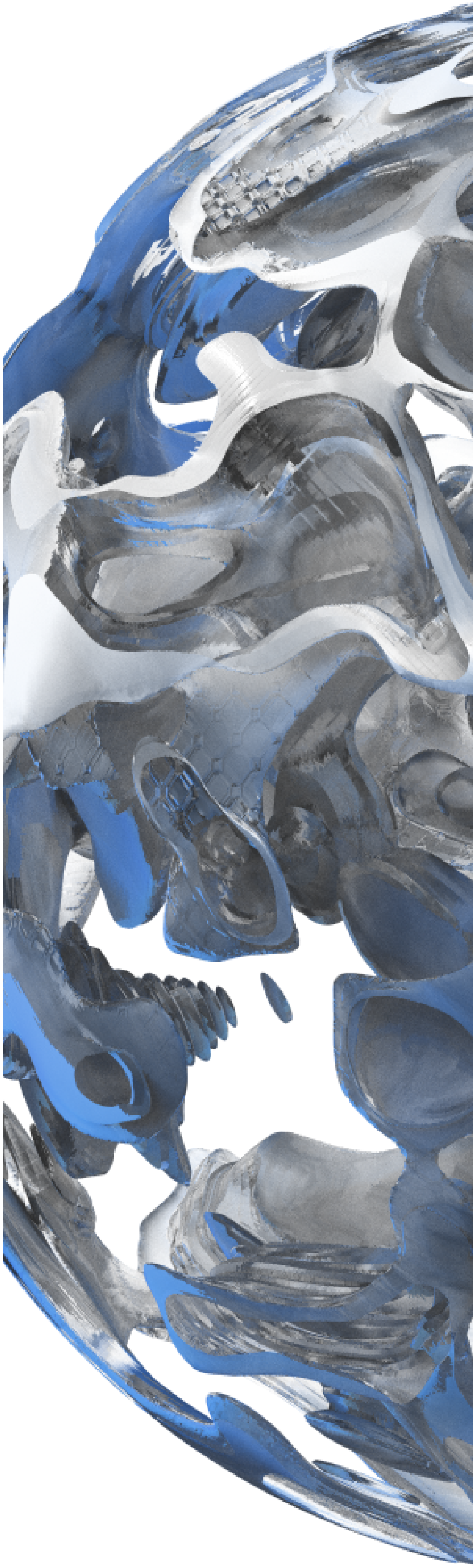


The VTK-m User's Guide

VTK-m version 1.5

Kenneth Moreland

With contributions from:
Hank Childs, Nickolas Davis, Mark Kim,
James Kress, Matthew Letter, La-ti Lo,
Robert Maynard, Sujin Philip, David Pugmire,
Allison Vacanti, Abhishek Yenpure,
and the VTK-m community

October 16, 2019

http://m.vtk.org
http://kitware.com

Published by Kitware Inc. ©2019

This project has been funded in whole or in part with Federal funds from the Department of Energy, including from Sandia National Laboratories, Los Alamos National Laboratory, Advanced Simulation and Computing, and Oak Ridge National Laboratory.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND 2019-12638 B

Printed and produced in the United States of America.

CONTRIBUTORS

This book includes contributions from the VTK-m community including the VTK-m development team and the user community. We would like to thank the following people for their significant contributions to this text:

Nickolas Davis and **Matthew Letter** for their help keeping the user's guide up to date with the VTK-m source code.

Sujin Philip, **Robert Maynard**, **James Kress**, **Abhishek Yenpure**, **Mark Kim**, and **Hank Childs** for their descriptions of numerous filters.

Allison Vacanti for her documentation of several VTK-m features in Sections 26.10 and 26.11 and select filters.

David Pugmire for his documentation of partitioned data sets (Section 7.5) and select filters.

Abhishek Yenpure and **La-ti Lo** for their documentation of locator structures (Chapter 30).

ABOUT THE COVER

The cover image is a visualization of the temperature field computed by the Nek5000 thermal hydraulics simulator. In the simulation twin inlets pump air into a box with a temperature difference between the 2 inlets. The visualization is provided by Matthew Larsen at Lawrence Livermore National Laboratory.

The interior cover image represents seismic wave propagation through the Earth. The visualization is provided by Matthew Larsen at Lawrence Livermore National Laboratory.

The cover design was done by Steve Jordan.

Join the VTK-m Community at http://m.vtk.org

CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

Part I

Getting Started

CHAPTER

ONE

INTRODUCTION

High-performance computing relies on ever finer threading. Advances in processor technology include ever greater numbers of cores, hyperthreading, accelerators with integrated blocks of cores, and special vectorized instructions, all of which require more software parallelism to achieve peak performance. Traditional visualization solutions cannot support this extreme level of concurrency. Extreme scale systems require a new programming model and a fundamental change in how we design algorithms. To address these issues we created VTK-m: the visualization toolkit for multi-/many-core architectures.

VTK-m supports a number of algorithms and the ability to design further algorithms through a top-down design with an emphasis on extreme parallelism. VTK-m also provides support for finding and building links across topologies, making it possible to perform operations that determine manifold surfaces, interpolate generated values, and find adjacencies. Although VTK-m provides a simplified high-level interface for programming, its template-based code removes the overhead of abstraction.

VTK-m simplifies the development of parallel scientific visualization algorithms by providing a framework of supporting functionality that allows developers to focus on visualization operations. Consider the listings in Figure 1.1 that compares the size of the implementation for the Marching Cubes algorithm in VTK-m with the equivalent reference implementation in the CUDA software development kit. Because VTK-m internally manages the parallel distribution of work and data, the VTK-m implementation is shorter and easier to maintain. Additionally, VTK-m provides data abstractions not provided by other libraries that make code written in VTK-m more versatile.

1.1 How to Use This Guide

This user's guide is organized into 5 parts to help guide novice to advanced users and to provide a convenient reference. Part I, Getting Started, provides a brief overview of using VTK-m. This part provides instructions on building VTK-m and some simple examples of using VTK-m. Users new to VTK-m are well served to read through Part I first to become acquainted with the basic concepts.

The remaining parts, which provide detailed documentation of increasing complexity, have chapters that do not need to be read in detail. Readers will likely find it useful to skip to specific topics of interest.

Part II, Using VTK-m, dives deeper into the VTK-m library. It provides much more detail on the concepts introduced in Part I and introduces new topics helpful to people who use VTK-m's existing algorithms.

Part III, Developing Algorithms, documents how to use VTK-m's framework to develop new or custom visualization algorithms. In this part we dive into the inner workings of filters and introduce the concept of a *worklet*, which is the base unit used to write a device-portable algorithm in VTK-m. Part III also documents many supporting functions that are helpful in implementing visualization algorithms.

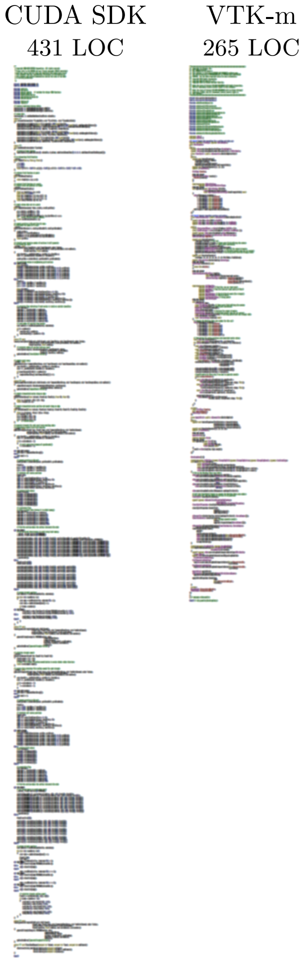


Figure 1.1: Comparison of the Marching Cubes algorithm in VTK-m and the reference implementation in the CUDA SDK. Implementations in VTK-m are simpler, shorter, more general, and easier to maintain. (Lines of code (LOC) measurements come from cloc.)

Part IV, Advanced Development, explores in more detail how VTK-m manages memory and devices. This information describes how to adapt VTK-m to custom data structures and new devices.

Part V, Core Development, exposes the inner workings of VTK-m. These concepts allow you to design new algorithmic structures not already available in VTK-m.

1.2 Conventions Used in This Guide

When documenting the VTK-m API, the following conventions are used.

- Filenames are printed in a sans serif font.
- C++ code is printed in a `monospace font`.
- Macros and namespaces from VTK-m are printed in `red`.

- Identifiers from VTK-m are printed in `blue`.
- Signatures, described in Chapter 17, and the tags used in them are printed in `green`.

This guide provides actual code samples throughout its discussions to demonstrate their use. These examples are all valid code that can be compiled and used although it is often the case that code snippets are provided. In such cases, the code must be placed in a larger context.

Did you know?

In this guide we periodically use these ***Did you know?*** *boxes to provide additional information related to the topic at hand.*

Common Errors

Common Errors *blocks are used to highlight some of the common problems or complications you might encounter when dealing with the topic of discussion.*

CHAPTER

TWO

BUILD AND INSTALL VTK-M

Before we begin describing how to develop with VTK-m, we have a brief overview of how to build VTK-m, optionally install it on your system, and start your own programs that use VTK-m.

2.1 Getting VTK-m

VTK-m is an open source software product where the code is made freely available. To get the latest released version of VTK-m, go to the VTK-m releases page:

> http://m.vtk.org/index.php/VTK-m_Releases

From there with your favorite browser you may download the source code from any of the recent VTK-m releases in a variety of different archive files such as zip or tar gzip.

For access to the most recent work, the VTK-m development team provides public anonymous read access to their main source code repository. The main VTK-m repository on a GitLab instance hosted at Kitware, Inc. The repository can be browsed from its project web page:

> https://gitlab.kitware.com/vtk/vtk-m

We leave access to the githosted repository as an exercise for the user. Those interested in git access for the purpose of contributing to VTK-m should consult the CONTRIBUTING guidelines documented in the source code.[1]

2.2 Configure VTK-m

VTK-m uses a cross-platform configuration tool named CMake to simplify the configuration and building across many supported platforms. CMake is available from many package distribution systems and can also be downloaded for many platforms from http://cmake.org.

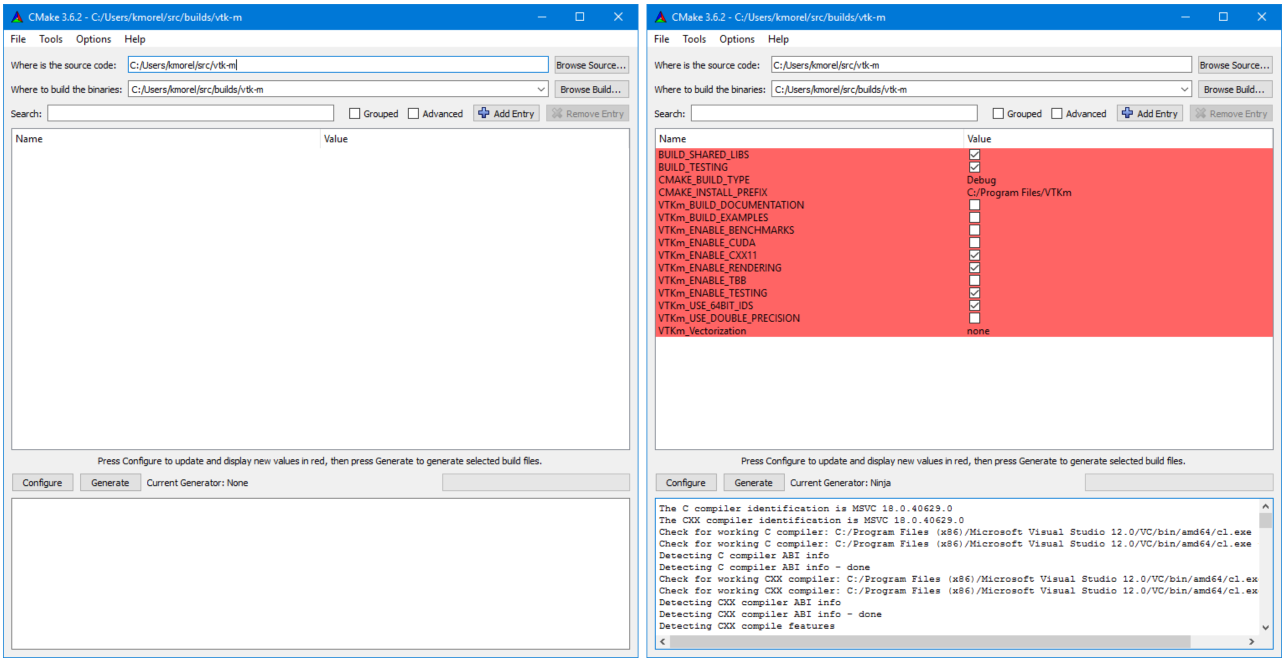
Most distributions of CMake come with a convenient GUI application (cmake-gui) that allows you to browse all of the available configuration variables and run the configuration. Many distributions also come with an alternative terminal-based version (ccmake), which is helpful when accessing remote systems where creating GUI windows is difficult.

[1] https://gitlab.kitware.com/vtk/vtk-m/blob/master/CONTRIBUTING.md

One helpful feature of CMake is that it allows you to establish a build directory separate from the source directory, and the VTK-m project requires that separation. Thus, when you run CMake for the first time, you want to set the build directory to a new empty directory and the source to the downloaded or cloned files. The following example shows the steps for the case where the VTK-m source is cloned from the git repository. (If you extracted files from an archive downloaded from the VTK-m web page, the instructions are the same from the second line down.)

Example 2.1: Running CMake on downloaded VTK-m source (Unix commands).

```
tar xvzf ~/Downloads/vtk-m-v1.5.0.tar.gz
mkdir vtkm-build
cd vtkm-build
cmake-gui ../vtk-m-v1.5.0
```



CMake 3.6.2 - C:/Users/kmorel/src/builds/vtk-m

Where is the source code: C:/Users/kmorel/src/vtk-m

Where to build the binaries: C:/Users/kmorel/src/builds/vtk-m

Search:

□ Grouped □ Advanced

Name	Value

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

CMake 3.6.2 - C:/Users/kmorel/src/builds/vtk-m

Where is the source code: C:/Users/kmorel/src/vtk-m

Where to build the binaries: C:/Users/kmorel/src/builds/vtk-m

Search:

□ Grouped □ Advanced

Name	Value
BUILD_SHARED_LIBS	☑
BUILD_TESTING	☑
CMAKE_BUILD_TYPE	Debug
CMAKE_INSTALL_PREFIX	C:/Program Files/VTKm
VTKm_BUILD_DOCUMENTATION	□
VTKm_BUILD_EXAMPLES	□
VTKm_ENABLE_BENCHMARKS	□
VTKm_ENABLE_CUDA	□
VTKm_ENABLE_CXX11	☑
VTKm_ENABLE_RENDERING	☑
VTKm_ENABLE_TBB	□
VTKm_ENABLE_TESTING	☑
VTKm_USE_64BIT_IDS	☑
VTKm_USE_DOUBLE_PRECISION	□
VTKm_Vectorization	none

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Ninja

```
The C compiler identification is MSVC 18.0.40629.0
The CXX compiler identification is MSVC 18.0.40629.0
Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64/cl.exe
Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64/cl.exe
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64/cl.ex
Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64/cl.ex
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Detecting CXX compile features
```

Figure 2.1: The CMake GUI configuring the VTK-m project. At left is the initial blank configuration. At right is the state after a configure pass.

The first time the CMake GUI runs, it initially comes up blank as shown at left in Figure 2.1. Verify that the source and build directories are correct (located at the top of the GUI) and then click the "Configure" button near the bottom. The first time you run configure, CMake brings up a dialog box asking what generator you want for the project. This allows you to select what build system or IDE to use (e.g. make, ninja, Visual Studio). Once you click "Finish," CMake will perform its first configuration. Don't worry if CMake gives an error about an error in this first configuration process.

Common Errors

Most options in CMake can be reconfigured at any time, but not the compiler and build system used. These must be set the first time configure is run and cannot be subsequently changed. If you want to change the compiler or the project file types, you will need to delete everything in the build directory and start over.

After the first configuration, the CMake GUI will provide several configuration options as shown in Figure 2.1 on the right. You now have a chance to modify the configuration of VTK-m, which allows you to modify both the behavior of the compiled VTK-m code as well as find components on your system. Using the CMake GUI is usually an iterative process where you set configuration options and re-run "Configure." Each time you configure, CMake might find new options, which are shown in red in the GUI.

It is often the case during this iterative configuration process that configuration errors occur. This can occur after a new option is enabled but CMake does not automatically find the necessary libraries to make that feature possible. For example, to enable TBB support, you may have to first enable building TBB, configure for TBB support, and then tell CMake where the TBB include directories and libraries are.

Once you have set all desired configuration variables and resolved any CMake errors, click the "Generate" button. This will create the build files (such as makefiles or project files depending on the generator chosen at the beginning). You can then close the CMake GUI.

There are a great number of configuration parameters available when running CMake on VTK-m. The following list contains the most common configuration parameters.

BUILD_SHARED_LIBS Determines whether static or shared libraries are built.

CMAKE_BUILD_TYPE Selects groups of compiler options from categories like Debug and Release. Debug builds are, obviously, easier to debug, but they run *much* slower than Release builds. Use Release builds whenever releasing production software or doing performance tests.

CMAKE_INSTALL_PREFIX The root directory to place files when building the install target.

VTKm_ENABLE_EXAMPLES The VTK-m repository comes with an examples directory. This macro determines whether they are built.

VTKm_ENABLE_BENCHMARKS If on, the VTK-m build includes several benchmark programs. The benchmarks are regression tests for performance.

VTKm_ENABLE_CUDA Determines whether VTK-m is built to run on CUDA GPU devices.

VTKm_CUDA_Architecture Specifies what GPU architecture(s) to build CUDA for. The options include native, fermi, kepler, maxwell, pascal, and volta.

VTKm_ENABLE_OPENMP Determines whether VTK-m is built to run on multi-core devices using OpenMP pragmas provided by the C++ compiler.

VTKm_ENABLE_RENDERING Determines whether to build the rendering library.

VTKm_ENABLE_TBB Determines whether VTK-m is built to run on multi-core x86 devices using the Intel Threading Building Blocks library.

VTKm_ENABLE_LOGGING If on, the VTK-m library will output log messages to std-out/std-err. This can be particuarly useful when debugging your source code or understanding what is going on in VTK-m

VTKm_ENABLE_TESTING If on, the VTK-m build includes building many test programs. The VTK-m source includes hundreds of regression tests to ensure quality during development.

VTKm_USE_64BIT_IDS If on, then VTK-m will be compiled to use 64-bit integers to index arrays and other lists. If off, then VTK-m will use 32-bit integers. 32-bit integers take less memory but could cause failures on larger data.

VTKm_USE_DOUBLE_PRECISION If on, then VTK-m will use double precision (64-bit) floating point numbers for calculations where the precision type is not otherwise specified. If off, then single precision (32-bit) floating point numbers are used. Regardless of this setting, VTK-m's templates will accept either type.

2.3 Building VTK-m

Once CMake successfully configures VTK-m and generates the files for the build system, you are ready to build VTK-m. As stated earlier, CMake supports generating configuration files for several different types of build tools. Make and ninja are common build tools, but CMake also supports building project files for several different types of integrated development environments such as Microsoft Visual Studio and Apple XCode.

The VTK-m libraries and test files are compiled when the default build is invoked. For example, if Makefiles were generated, the build is invoked by calling make in the build directory. Expanding on Example 2.1

Example 2.2: Using make to build VTK-m.

```
tar xvzf ~/Downloads/vtk-m-v1.5.0.tar.gz
mkdir vtkm-build
cd vtkm-build
cmake-gui ../vtk-m-v1.5.0
make -j
make install
```

Did you know?

The Makefiles and other project files generated by CMake support parallel builds, which run multiple compile steps simultaneously. On computers that have multiple processing cores (as do almost all modern computers), this can significantly speed up the overall compile. Some build systems require a special flag to engage parallel compiles. For example, make requires the -j flag to start parallel builds as demonstrated in Example 2.2.

Did you know?

Example 2.2 assumes that a make build system was generated, which is the default on most system. However, CMake supports many more build systems, which use different commands to run the build. If you are not sure what the appropriate build command is, you can run cmake --build to allow CMake to start the build using whatever build system is being used.

Common Errors

CMake allows you to switch between several types of builds including default, Debug, and Release. Programs and libraries compiled as release builds can run much *faster than those from other types of builds. Thus, it is important to perform Release builds of all software released for production or where runtime is a concern. Some integrated development environments such as Microsoft Visual Studio allow you to specify the different build types within the build system. But for other build programs, like make, you have to specify the build type in the CMAKE_BUILD_TYPE CMake configuration variable, which is described in Section 2.2.*

CMake creates several build "targets" that specify the group of things to build. The default target builds all of VTK-m's libraries as well as tests, examples, and benchmarks if enabled. The test target executes each of the VTK-m regression tests and verifies they complete successfully on the system. The install target copies the subset of files required to use VTK-m to a common installation directory. The install target may need to be run as an administrator user if the installation directory is a system directory.

> Did you know?
>
> *VTK-m contains a significant amount of regression tests. If you are not concerned with testing a build on a given system, you can turn off building the testing, benchmarks, and examples using the CMake configuration variables described in Section 2.2. This can shorten the VTK-m compile time.*

2.4 Linking to VTK-m

Ultimately, the value of VTK-m is the ability to link it into external projects that you write. The header files and libraries installed with VTK-m are typical, and thus you can link VTK-m into a software project using any type of build system. However, VTK-m comes with several CMake configuration files that simplify linking VTK-m into another project that is also managed by CMake. Thus, the documentation in this section is specifically for finding and configuring VTK-m for CMake projects.

VTK-m can be configured from an external project using the `find_package` CMake function. The behavior and use of this function is well described in the CMake documentation. The first argument to `find_package` is the name of the package, which in this case is `VTKm`. CMake configures this package by looking for a file named VTKmConfig.cmake, which will be located in the lib/cmake/vtkm-1.5 directory of the install or build of VTK-m. The configurable CMake variable CMAKE_PREFIX_PATH can be set to to the build or install directory, or VTKm_DIR can be set to the directory that contains this file.

Example 2.3: Loading VTK-m configuration from an external CMake project.

```
find_package(VTKm REQUIRED)
```

> Did you know?
>
> *The CMake* `find_package` *function also supports several features not discussed here including specifying a minimum or exact version of VTK-m and turning off some of the status messages. See the CMake documentation for more details.*

When you load the VTK-m package in CMake, several libraries are defined. Projects building with VTK-m components should link against one or more of these libraries as appropriate, typically with the target_link_libraries command.

Example 2.4: Linking VTK-m code into an external program.

```
find_package(VTKm REQUIRED)

add_executable(myprog myprog.cxx)
target_link_libraries(myprog vtkm_filter)
```

Several library targets are provided, but most projects will need to link in one or more of the following.

vtkm_cont Contains the base objects used to control VTK-m. This library should always be linked in.

vtkm_filter Contains VTK-m's pre-built filters including but not limited to CellAverage, CleanGrid, Contour, ExternalFaces, and PointAverage. Applications that are looking to use VTK-m filters will need to link to this library.

vtkm_rendering Contains VTK-m's rendering components. This library is only available if VTKm_ENABLE_RENDERING is set to true.

vtkm_source Contains VTK-m's pre-built dataset generators including but not limited to Wavelet, Tangle, and Oscillator. Most applications will not need to link to this library.

Did you know?

The "libraries" made available in the VTK-m do more than add a library to the linker line. These libraries are actually defined as external targets that establish several compiler flags, like include file directories. Many CMake packages require you to set up other target options to compile correctly, but for VTK-m it is sufficient to simply link against the library.

Common Errors

Because the VTK-m CMake libraries do more than set the link line, correcting the link libraries can do more than fix link problems. For example, if you are getting compile errors about not finding VTK-m header files, then you probably need to link to one of VTK-m's libraries to fix the problem rather than try to add the include directories yourself.

The following is a list of all the CMake variables defined when the `find_package` function completes.

VTKm_FOUND Set to true if the VTK-m CMake package is successfully loaded. If `find_package` was not called with the `REQUIRED` option, then this variable should be checked before attempting to use VTK-m.

VTKm_VERSION The version number of the loaded VTK-m package. The package also sets VTKm_VERSION_MAJOR, VTKm_VERSION_MINOR, and VTKm_VERSION_PATCH to get the individual components of the version. There is also a VTKm_VERSION_FULL that is augmented with a partial git SHA to identify snapshots in between releases.

VTKm_ENABLE_CUDA Set to true if VTK-m was compiled for CUDA.

VTKm_ENABLE_OPENMP Set to true if VTK-m was compiled for OpenMP.

VTKm_ENABLE_TBB Set to true if VTK-m was compiled for TBB.

VTKm_ENABLE_RENDERING Set to true if the VTK-m rendering library was compiled.

VTKm_ENABLE_MPI Set to true if VTK-m was compiled with MPI support.

These package variables can be used to query whether optional components are supported before they are used in your CMake configuration.

Example 2.5: Using an optional component of VTK-m.

```cmake
find_package(VTKm REQUIRED)

if (NOT VTKm_ENABLE_RENDERING)
  message(SEND_ERROR "VTK-m must be built with rendering on.")
endif()

add_executable(myprog myprog.cxx)
target_link_libraries(myprog vtkm_cont vtkm_rendering)
```

CHAPTER

THREE

QUICK START

In this chapter we go through the steps to create a simple program that uses VTK-m. This "hello world" example presents only the bare minimum of features available. The remainder of this book documents dives into much greater detail.

We will call the example program we are building VTKmQuickStart. It will demonstrate reading data from a file, processing the data with a filter, and rendering an image of the data. Readers who are less interested in an explanation and are more interested in browsing some code can skip to Section 3.5 on page 17.

3.1 Initialize

The first step to using VTK-m is to initialize the library. Although initializing VTK-m is *optional*, it is recommend to allow VTK-m to configure devices and logging. Initialization is done by calling the `vtkm::cont::Initialize` function. The `Initialize` function is defined in the vtkm/cont/Initialize.h header file.

`Initialize` takes the `argc` and `argv` arguments that are passed to the `main` function of your program, find any command line arguments relevant to VTK-m, and remove them from the list to make further command line argument processing easier.

Example 3.1: Initializing VTK-m.

```
int main(int argc, char* argv[])
{
  vtkm::cont::Initialize(argc, argv);
```

`Initialize` has many options to customize command line argument processing. See Chapter 6 for more details.

Did you know?

Don't have access to `argc` *and* `argv`*? No problem. You can call* `vtkm::cont::Initialize` *with no arguments.*

3.2 Reading a File

VTK-m comes with a simple I/O library that can read and write files in VTK legacy format. These files have a ".vtk" extension.

VTK legacy files can be read using the vtkm::io::reader::VTKDataSetReader object, which is declared in the vtkm/io/reader/VTKDataSetReader.h header file. The object is constructed with a string specifying the filename (which for this example we will get from the command line). The data is then read in by calling the VTKDataSetReader::ReadDataSet method.

Example 3.2: Reading data from a VTK legacy file.

```
vtkm::io::reader::VTKDataSetReader reader(argv[1]);
vtkm::cont::DataSet inData = reader.ReadDataSet();
```

The ReadDataSet method returns the data in a vtkm::cont::DataSet object. The structure and features of a DataSet object is described in Chapter 7. For the purposes of this quick start, we will treat DataSet as a mostly opaque object that gets passed to and from operations in VTK-m.

More information about VTK-m's file readers and writers can be found in Chapter 8.

3.3 Running a Filter

Algorithms in VTK-m are encapsulated in units called *filters*. A filter takes in a DataSet, processes it, and returns a new DataSet. The returned DataSet often, but not always, contains data inherited from the source data.

VTK-m comes with many filters, which are documented in Chapter 9. For this example, we will demonstrate the use of the vtkm::filter::MeshQuality filter, which is defined in the vtkm/filter/MeshQuality.h header file. The MeshQuality filter will compute for each cell in the input data will compute a quantity representing some metric of the cell's shape. Several metrics are available, and in this example we will find the area of each cell.

Like all filters, MeshQuality contains an Execute method that takes an input DataSet and produces an output DataSet. It also has several methods used to set up the parameters of the execution. Section 9.1.18 provides details on all the options of MeshQuality. Suffice it to say that in this example we instruct the filter to find the area of each cell, which it will output to a field named "area."

Example 3.3: Running a filter.

```
vtkm::filter::MeshQuality cellArea(vtkm::filter::CellMetric::AREA);
vtkm::cont::DataSet outData = cellArea.Execute(inData);
```

3.4 Rendering an Image

Although it is possible to leverage external rendering systems, VTK-m comes with its own self-contained image rendering algorithms. These rendering classes are completely implemented with the parallel features provided by VTK-m, so using rendering in VTK-m does not require any complex library dependencies.

Even a simple rendering scene requires setting up several parameters to establish what is to be featured in the image including what data should be rendered, how that data should be represented, where objects should be placed in space, and the qualities of the image to generate. Consequently, setting up rendering in VTK-m involves many steps. Chapter 10 goes into much detail on the ways in which a rendering scene is specified. For now, we just briefly present some boilerplate to achieve a simple rendering.

Example 3.4: Rendering data.

```
vtkm::rendering::Actor actor(outData.GetCellSet(),
                             outData.GetCoordinateSystem(),
                             outData.GetField("area"));

```

```
 5    vtkm::rendering::Scene scene;
 6    scene.AddActor(actor);
 7
 8    vtkm::rendering::MapperRayTracer mapper;
 9
10    vtkm::rendering::CanvasRayTracer canvas(1280, 1024);
11
12    vtkm::rendering::View3D view(scene, mapper, canvas);
13    view.Initialize();
14
15    view.Paint();
16
17    view.SaveAs("image.ppm");
```

The first step in setting up a render is to create a *scene*. A scene comprises some number of *actors*, which represent some data to be rendered in some location in space. In our case we only have one DataSet to render, so we simply create a single actor and add it to a scene as shown in lines 1–6.

The second step in setting up a render is to create a *view*. The view comprises the aforementioned scene, a *mapper*, which describes how the data are to be rendered, and a *canvas*, which holds the image buffer and other rendering context. The view is created in line 12. Before it is used, the view must be initialized (line 13).

Once this set up is complete, the image generation can finally be performed by calling `Paint` on the view object (line 15). However, the rendering done by VTK-m's rendering classes is performed offscreen, which means that the result does not appear on your computer's monitor. The easiest way to see the image is to save it to an image file using the `SaveAs` method (line 17).

3.5 The Full Example

Putting together the examples from Sections 3.1 to 3.4, here is a complete program for reading, processing, and rendering data with VTK-m.

Example 3.5: Simple example of using VTK-m.

```
#include <vtkm/cont/Initialize.h>

#include <vtkm/io/reader/VTKDataSetReader.h>

#include <vtkm/filter/MeshQuality.h>

#include <vtkm/rendering/Actor.h>
#include <vtkm/rendering/CanvasRayTracer.h>
#include <vtkm/rendering/MapperRayTracer.h>
#include <vtkm/rendering/Scene.h>
#include <vtkm/rendering/View3D.h>

int main(int argc, char* argv[])
{
  vtkm::cont::Initialize(argc, argv);

  if (argc != 2)
  {
    std::cerr << "USAGE: " << argv[0] << " <file.vtk>" << std::endl;
    return 1;
  }

  // Read in a file specified in the first command line argument.
  vtkm::io::reader::VTKDataSetReader reader(argv[1]);
  vtkm::cont::DataSet inData = reader.ReadDataSet();

  // Run the data through the elevation filter.
  vtkm::filter::MeshQuality cellArea(vtkm::filter::CellMetric::AREA);
  vtkm::cont::DataSet outData = cellArea.Execute(inData);

  // Render an image and write it out to a file.
  vtkm::rendering::Actor actor(outData.GetCellSet(),
                               outData.GetCoordinateSystem(),
                               outData.GetField("area"));

  vtkm::rendering::Scene scene;
  scene.AddActor(actor);

  vtkm::rendering::MapperRayTracer mapper;

  vtkm::rendering::CanvasRayTracer canvas(1280, 1024);

  vtkm::rendering::View3D view(scene, mapper, canvas);
  view.Initialize();

  view.Paint();

  view.SaveAs("image.ppm");

  return 0;
}
```

3.6 Build Configuration

Now that we have the program listed in Example 3.5, we still need to compile it with the appropriate compilers and flags. By far the easiest way to compile VTK-m code is to use CMake. CMake command that can be used to link code to VTK-m is discussed in Section 2.4. The following example provides a minimal CMakeLists.txt required to build this program.

Example 3.6: CMakeLists.txt to build a program using VTK-m.

```
cmake_minimum_required(VERSION 3.13)
project(VTKmQuickStart CXX)

find_package(VTKm REQUIRED)

add_executable(VTKmQuickStart VTKmQuickStart.cxx)
target_link_libraries(VTKmQuickStart vtkm_filter vtkm_rendering)
```

The first two lines contain boilerplat for any CMakeLists.txt file. They all should declare the minimum CMake version required (for backward compatability) and have a `project` command to declare which languages are used.

The remainder of the commands find the VTK-m library, declare the program begin compiled, and link the program to the VTK-m library. These steps are described in detail in Section 2.4.

Part II

Using VTK-m

CHAPTER

FOUR

BASE TYPES

It is common for a framework to define its own types. Even the C++ standard template library defines its own base types like `std::size_t` and `std::pair`. VTK-m is no exception.

In fact VTK-m provides a great many base types. It is the general coding standard of VTK-m to not directly use the base C types like `int` and `float` and instead to use types declared in VTK-m. The rational is to precisely declare the representation of each variable to prevent future troubles.

Consider that you are programming something and you need to declare an integer variable. You would declare this variable as `int`, right? Well, maybe. In C++, the declaration `int` does not simply mean "an integer." `int` means something much more specific than that. If you were to look up the C++11 standard, you would find that `int` is an integer represented in 32 bits with a two's complement signed representation. In fact, a C++ compiler has no less than 8 standard integer types.[1]

So, `int` is nowhere near as general as the code might make it seem, and treating it as such could lead to trouble. For example, consider the MPI standard, which, back in the 1990's, implicitly selected `int` for its indexing needs. Fast forward to today where there is a need to reference buffers with more than 2 billion elements, but the standard is stuck with a data type that cannot represent sizes that big.[2]

Consequently, we feel that with VTK-m it is best to declare the intention of a variable with its declaration, which should help both prevent errors and future proof code. All the types presented in this chapter are declared in vtkm/Types.h, which is typically included either directly or indirectly by all source using VTK-m.

4.1 Floating Point Types

VTK-m declares 2 types to hold floating point numbers: `vtkm::Float32` and `vtkm::Float64`. These, of course, represent floating point numbers with 32-bits and 64-bits of precision, respectively. These should be used when the precision of a floating point number is predetermined.

When the precision of a floating point number is not predetermined, operations usually have to be overloaded or templated to work with multiple precisions. In cases where a precision must be set, but no particular precision is specified, `vtkm::FloatDefault` should be used. `vtkm::FloatDefault` will be set to either `vtkm::Float32` or `vtkm::Float64` depending on whether the CMake option VTKM_USE_DOUBLE_PRECISION was set when VTK-m was compiled, as discussed in Section 2.2. Using `vtkm::FloatDefault` makes it easier for users to trade off precision and speed.

[1] I intentionally use the phrase "no less than" for our pedantic readers. One could argue that `char` and `bool` are treated distinctly by the compiler even if their representations match either `signed char` or `unsigned char`. Furthermore, many modern C++ compilers have extensions for less universally accepted types like 128-bit integers.

[2] To be fair, it is *possible* to represent buffers this large in MPI, but it is extraordinarily awkward to do so.

4.2 Integer Types

The most common use of an integer in VTK-m is to index arrays and other like storage mechanisms. For most indexing purposes, the `vtkm::Id` type should be used. (The width of `vtkm::Id` is determined by the VTKM_USE_64BIT_IDS CMake option.)

VTK-m also has a secondary index type named `vtkm::IdComponent`, which is smaller and typically used for indexing groups of components within a thread. For example, if you had an array of 3D points, you would use `vtkm::Id` to reference each point, and you would use `vtkm::IdComponent` to reference the respective x, y, and z components.

Did you know?

The VTK-m index types, `vtkm::Id` *and* `vtkm::IdComponent` *use signed integers. This breaks with the convention of other common index types like the C++ standard template library* `std::size_t`*, which use unsigned integers. Unsigned integers make sense for indices as a valid index is always 0 or greater. However, doing things like iterating in a for loop backward, representing relative indices, and representing invalid values is much easier with signed integers. Thus, VTK-m chooses to use a signed integer for indexing.*

VTK-m also has types to declare an integer of a specific width and sign. The types `vtkm::Int8`, `vtkm::Int16`, `vtkm::Int32`, and `vtkm::Int64` specify signed integers of 1, 2, 4, and 8 bits, respectively. Likewise, the types `vtkm::UInt8`, `vtkm::UInt16`, `vtkm::UInt32`, and `vtkm::UInt64` specify unsigned integers of 1, 2, 4, and 8 bits, respectively.

4.3 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides a collection of base types to represent these short vectors, which are collectively referred to as `Vec` types.

`vtkm::Vec2f`, `vtkm::Vec3f`, and `vtkm::Vec4f` specify floating point `Vec`s of 2, 3, and 4 components, respectively. The precision of the floating point numbers follows that of `vtkm::FloatDefault` (which, from what is said in Section 4.1, is specified by the VTKM_USE_DOUBLE_PRECISION compile option). Components of these and other `Vec` types can be references through the `[ ]` operator, much like a C array. `Vec`s also support basic arithmetic operators so that they can be used much like their scalar-value counterparts.

Example 4.1: Simple use of `Vec` objects.

```
vtkm::Vec2f A(1);          // A is (1, 1)
A[1] = 3;                  // A is (1, 3) now
vtkm::Vec2f B = { 4, 5 }; // B is (4, 5)
vtkm::Vec2f C = A + B;     // C is (5, 8)
vtkm::FloatDefault manhattanDistance = C[0] + C[1];
```

You can also specify the precision for each of these vector types by appending the bit size of each component. For example, `vtkm::Vec3f_32` and `vtkm::Vec3f_64` represent 3-component floating point vectors with each component being 32 bits and 64 bits respectively. Note that the precision number refers to the precision of each component, not the vector as a whole. So `vtkm::Vec3f_32` contains 3 32-bit (4-byte) floating point components, which means the entire `vtkm::Vec3f_32` requires 96 bits (12 bytes).

To help with indexing 2-, 3-, and 4- dimensional arrays, VTK-m provides the types vtkm::Id2, vtkm::Id3, and vtkm::Id4, which are Vecs of type vtkm::Id. Likewise, VTK-m provides vtkm::IdComponent2, vtkm::IdComponent3, and vtkm::IdComponent4.

VTK-m also provides types for Vecs of integers of all varieties described in Section 4.2. vtkm::Vec2i, vtkm::Vec3i, and vtkm::Vec4i are vectors of signed integers whereas vtkm::Vec2ui, vtkm::Vec3ui, and vtkm::Vec4ui are vectors of unsigned integers. All of these sport components of a width equal to vtkm::Id. The width can be specified by appending the desired number of bits in the same way as the floating point Vecs. For example, vtkm::Vec4ui_8 is a Vec of 4 unsigned bytes.

These types really just scratch the surface of the Vec types available in VTK-m and the things that can be done with them. See Chapter 19 for more information on Vec types and what can be done with them.

CHAPTER

FIVE

VTK-M VERSION

As the VTK-m code evolves, changes to the interface and behavior will inevitably happen. Consequently, code that links into VTK-m might need a specific version of VTK-m or changes its behavior based on what version of VTK-m it is using. To facilitate this, VTK-m software is managed with a versioning system and advertises its version in multiple ways. As with many software products, VTK-m has three version numbers: major, minor, and patch. The major version represents significant changes in the VTK-m implementation and interface. Changes in the major version include backward incompatible changes. The minor version represents added functionality. Generally, changes in the minor version to not introduce changes to the API (although the early 1.X versions of VTK-m violate this). The patch version represents fixes provided after a release occurs. Patch versions represent minimal change and do not add features.

If you are writing a software package that is managed by CMake and load VTK-m with the `find_package` command as described in Section 2.4, then you can query the VTK-m version directly in the CMake configuration. When you load VTK-m with `find_package`, CMake sets the variables VTKm_VERSION_MAJOR, VTKm_VERSION_MINOR, and VTKm_VERSION_PATCH to the major, minor, and patch versions, respectively. Additionally, VTKm_VERSION is set to the “major.minor” version number and VTKm_VERSION_FULL is set to the “major.minor.patch” version number. If the current version of VTK-m is actually a development version that is in between releases of VTK-m, then and abbreviated SHA of the git commit is also included as part of VTKm_VERSION_FULL.

Did you know?

If you have a specific version of VTK-m required for your software, you can also use the version option to the `find_package` *CMake command. The* `find_package` *command takes an optional version argument that causes the command to fail if the wrong version of the package is found.*

It is also possible to query the VTK-m version directly in your code through preprocessor macros. The vtkm/Version.h header file defines the following preprocessor macros to identify the VTK-m version. `VTKM_VERSION_MAJOR`, `VTKM_VERSION_MINOR`, and `VTKM_VERSION_PATCH` are set to integer numbers representing the major, minor, and patch versions, respectively. Additionally, `VTKM_VERSION` is set to the “major.minor” version number as a string and `VTKM_VERSION_FULL` is set to the “major.minor.patch” version number (also as a string). If the current version of VTK-m is actually a development version that is in between releases of VTK-m, then and abbreviated SHA of the git commit is also included as part of `VTKM_VERSION_FULL`.

Common Errors

Note that the CMake variables all begin with `VTKm_` *(lowercase "m") whereas the preprocessor macros begin with* `VTKM_` *(all uppercase). This follows the respective conventions of CMake variables and preprocessor macros.*

Note that vtkm/Version.h does not include any other VTK-m header files. This gives your code a chance to load, query, and react to the VTK-m version before loading any VTK-m code proper.

INITIALIZATION

When it comes to running VTK-m code, there are a few ways in which various facilities, such as logging and device connections, can be initialized. The preferred method of initializing these features is to run the vtkm::cont::Initialize function. Although it is not strictly necessary to call Initialize, it is recommended to set up state and check for available devices.

Initialize can be called without any arguments, in which case VTK-m will be initialized with defaults. But it can also optionally take the argc and argv arguments to the main function to parse some options that control the state of VTK-m. VTK-m accepts arguments that, for example, configure the compute device to use or establish logging levels. Any arguments that are handled by VTK-m are removed from the argc/argv list so that your program can then respond to the remaining arguments.

Initialize takes an optional third argument that specifies some options on the behavior of the argument parsing. The options are specified as a bit-wise "or" of fields specified in the vtkm::cont::InitializeOptions enum. The available initialize options are

None Placeholder used when no options are enabled. This is the value used when the third argument to Initialize is not provided.

RequireDevice Issue an error if the device argument is not specified.

DefaultAnyDevice If no device is specified, treat it as if the user gave "--device=Any". This means that DeviceAdapterTagUndefined will never be return in the result.

AddHelp Add a help option. If "-h" or "--help" is provided, prints a usage statement. Of course, the usage statement will only print out arguments processed by VTK-m, which is why help is not given by default. A string with usage help is returned from Initialize so that the calling program can provide VTK-m's help in its own usage statement.

ErrorOnBadOption If an unknown option is encountered, the program terminates with an error. If this option is not provided, any unknown options are returned in argv. If this option is used, it is a good idea to use AddHelp as well.

ErrorOnBadArgument If an extra argument is encountered, the program terminates with an error. If this option is not provided, any unknown arguments are returned in argv.

Strict If supplied, Initialize treats its own arguments as the only ones supported by the application and provides an error if not followed exactly. This is a convenience option that is a combination of ErrorOnBadOption, ErrorOnBadArgument, and AddHelp.

As stated earlier, vtkm::cont::Initialize removes parsed options from the argc/argv passed to it so that the calling program can further respond to command line arguments. Additionally, Initialize returns an vtkm::cont::InitializeResult object that contains the following information.

`Device` A `vtkm::cont::DeviceAdapterId` that represents the device specified by the command line arguments. (See Chapter 12 for details on how VTK-m represents devices.) If no device is specified in the command line options, `vtkm::cont::DeviceAdapterTagUndefined` is returned (unless the `DefaultAnyDevice` option is given, in which case `vtkm::cont::DeviceAdapterTagAny` is returned).

Example 6.1: Calling `Initialize`.

```cpp
#include <vtkm/cont/Initialize.h>

int main(int argc, char** argv)
{
  vtkm::cont::InitializeOptions options =
    vtkm::cont::InitializeOptions::ErrorOnBadArgument |
    vtkm::cont::InitializeOptions::DefaultAnyDevice;
  vtkm::cont::InitializeResult config = vtkm::cont::Initialize(argc, argv, options);

  if (argc != 2)
  {
    std::cerr << "USAGE: " << argv[0] << " [options] filename" << std::endl;
    std::cerr << "Available options are:" << std::endl;
    std::cerr << config.Usage << std::endl;
    return 1;
  }
  std::string filename = argv[1];

  // Do something cool with VTK-m
  // ...

  return 0;
}
```

CHAPTER

SEVEN

DATA SETS

A *data set*, implemented with the vtkm::cont::DataSet class, contains and manages the geometric data structures that VTK-m operates on. A data set comprises the following 3 data structures.

Cell Set A cell set describes topological connections. A cell set defines some number of points in space and how they connect to form cells, filled regions of space. A data set has exactly one cell set.

Field A field describes numerical data associated with the topological elements in a cell set. The field is represented as an array, and each entry in the field array corresponds to a topological element (point, edge, face, or cell). Together the cell set topology and discrete data values in the field provide an interpolated function throughout the volume of space covered by the data set. A cell set can have any number of fields.

Coordinate System A coordinate system is a special field that describes the physical location of the points in a data set. Although it is most common for a data set to contain a single coordinate system, VTK-m supports data sets with no coordinate system such as abstract data structures like graphs that might not have positions in a space. DataSet also supports multiple coordinate systems for data that have multiple representations for position. For example, geospatial data could simultaneously have coordinate systems defined by 3D position, latitude-longitude, and any number of 2D projections.

In addition to the base vtkm::cont::DataSet, VTK-m provides vtkm::cont::PartitionedDataSet to represent data partitioned into multiple domains. A PartitionedDataSet is implemented as a collection of DataSet objects. Partitioned data sets are described later in Section 7.5.

7.1 Building Data Sets

Before we go into detail on the cell sets, fields, and coordinate systems that make up a data set in VTK-m, let us first discuss how to build a data set. One simple way to build a data set is to load data from a file using the vtkm::io module. Reading files is discussed in detail in Chapter 8.

This section describes building data sets of different types using a set of classes named DataSetBuilder*, which provide a convenience layer on top of vtkm::cont::DataSet to make it easier to create data sets.

Did you know?

To simplify the introduction of DataSets, this section uses the simplest mechanisms. In many cases this involves loading data in a std::vector and passing that to VTK-m, which usually causes the data to be copied. This is not the most efficient method to load data into VTK-m. Although it is sufficient for small *data or data that come from a "slow" source, such as a file, it might be a bottleneck for large data generated by another library. It is possible to adapt VTK-m's* `DataSet` *to externally defined data. This is done by wrapping existing data into what is called* `ArrayHandle`*, but this is a more advanced topic that will not be addressed in this chapter.* `ArrayHandle`*s are introduced in Chapter 16 and more adaptive techniques are described in later chapters.*

7.1.1 Creating Uniform Grids

Uniform grids are meshes that have a regular array structure with points uniformly spaced parallel to the axes. Uniform grids are also sometimes called regular grids or images.

The `vtkm::cont::DataSetBuilderUniform` class can be used to easily create 2- or 3-dimensional uniform grids. `DataSetBuilderUniform` has several versions of a method named `Create` that takes the number of points in each dimension, the origin, and the spacing. The origin is the location of the first point of the data (in the lower left corner), and the spacing is the distance between points in the x, y, and z directions. The `Create` methods also take an optional name for the coordinate system and an optional name for the cell set.

The following example creates a `vtkm::cont::DataSet` containing a uniform grid of 101 × 101 × 26 points.

Example 7.1: Creating a uniform grid.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet = dataSetBuilder.Create(vtkm::Id3(101, 101, 26));
```

If not specified, the origin will be at the coordinates $(0,0,0)$ and the spacing will be 1 in each direction. Thus, in the previous example the width, height, and depth of the mesh in physical space will be 100, 100, and 25, respectively, and the mesh will be centered at $(50,50,12.5)$. Let us say we actually want a mesh of the same dimensions, but we want the z direction to be stretched out so that the mesh will be the same size in each direction, and we want the mesh centered at the origin.

Example 7.2: Creating a uniform grid with custom origin and spacing.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet =
  dataSetBuilder.Create(vtkm::Id3(101, 101, 26),
                        vtkm::Vec3f(-50.0, -50.0, -50.0),
                        vtkm::Vec3f(1.0, 1.0, 4.0));
```

7.1.2 Creating Rectilinear Grids

A rectilinear grid is similar to a uniform grid except that a rectilinear grid can adjust the spacing between adjacent grid points. This allows the rectilinear grid to have tighter sampling in some areas of space, but the points are still constrained to be aligned with the axes and each other. The irregular spacing of a rectilinear grid is specified by providing a separate array each for the x, y, and z coordinates.

The `vtkm::cont::DataSetBuilderRectilinear` class can be used to easily create 2- or 3-dimensional rectilinear grids. `DataSetBuilderRectilinear` has several versions of a method named `Create` that takes these coordinate arrays and builds a `vtkm::cont::DataSet` out of them. The arrays can be supplied as either standard C arrays or as `std::vector` objects, in which case the data in the arrays are copied into the `DataSet`. These arrays can also be passed as `ArrayHandle` objects (introduced later in this book), in which case the data are shallow copied.

The following example creates a vtkm::cont::DataSet containing a rectilinear grid with 201 × 201 × 101 points with different irregular spacing along each axis.

Example 7.3: Creating a rectilinear grid.

```
// Make x coordinates range from -4 to 4 with tighter spacing near 0.
std::vector<vtkm::Float32> xCoordinates;
for (vtkm::Float32 x = -2.0f; x <= 2.0f; x += 0.02f)
{
  xCoordinates.push_back(vtkm::CopySign(x * x, x));
}

// Make y coordinates range from 0 to 2 with tighter spacing near 2.
std::vector<vtkm::Float32> yCoordinates;
for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
{
  yCoordinates.push_back(vtkm::Sqrt(y));
}

// Make z coordinates rangefrom -1 to 1 with even spacing.
std::vector<vtkm::Float32> zCoordinates;
for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
{
  zCoordinates.push_back(z);
}

vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;

vtkm::cont::DataSet dataSet =
  dataSetBuilder.Create(xCoordinates, yCoordinates, zCoordinates);
```

7.1.3 Creating Explicit Meshes

An explicit mesh is an arbitrary collection of cells with arbitrary connections. It can have multiple different types of cells. Explicit meshes are also known as unstructured grids. Explicit meshes can contain cells of different shapes. The shapes that VTK-m currently supports are listed in Figure 7.1.

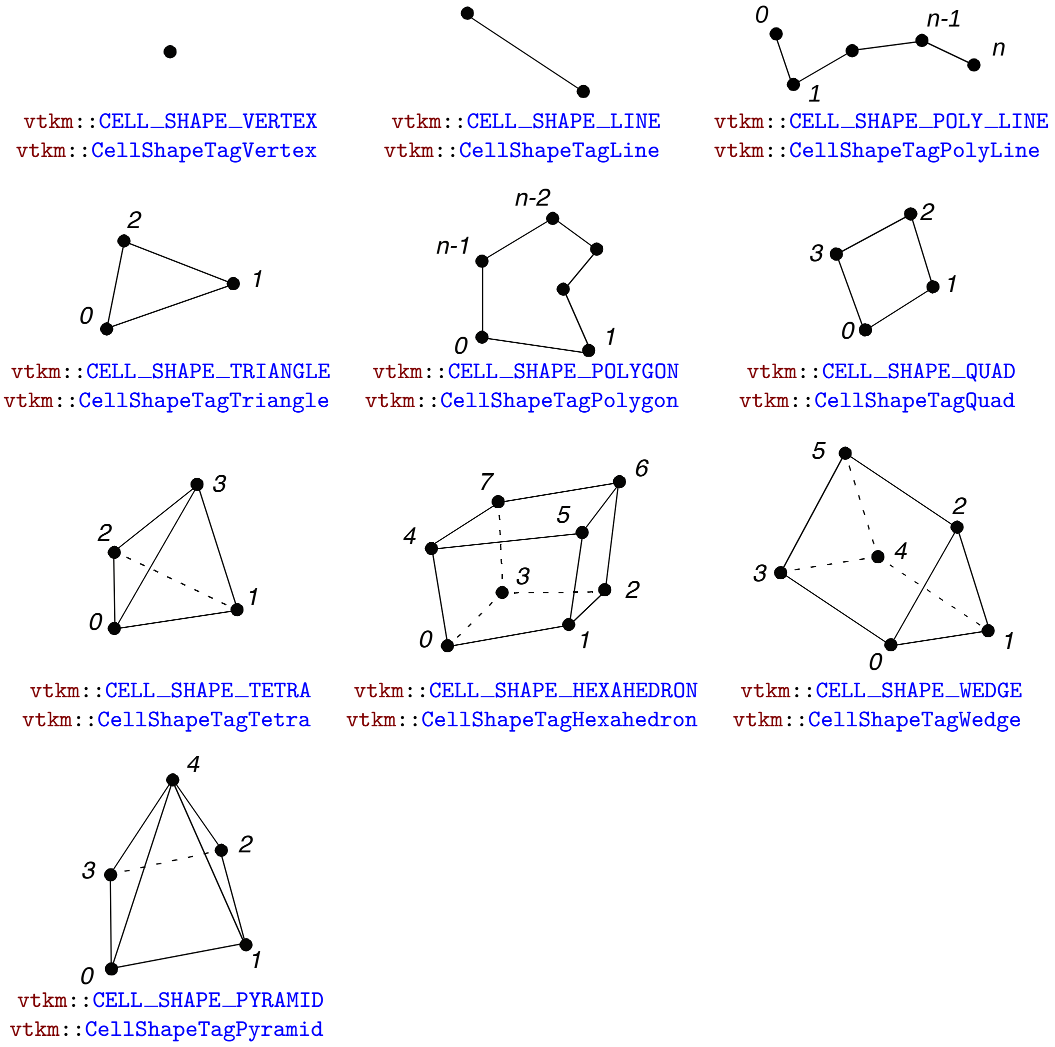
The cells of an explicit mesh are defined with the following 3 arrays, which are depicted graphically in Figure 7.2.

Shapes An array of ids identifying the shape of the cell. Each value is a vtkm::UInt8 and should be set to one of the vtkm ::CELL_SHAPE_* constants. The shapes and their identifiers are shown in Figure 7.1. The size of this array is equal to the number of cells in the set.

Connectivity An array that lists all the points that comprise each cell. Each entry in the array is a vtkm::Id giving the point id associated with a vertex of a cell. The points for each cell are given in a prescribed order for each shape, which is also shown in Figure 7.1. The point indices are stored consecutively from the first cell to the last.

Offsets An array of vtkm::Id s pointing to the index in the connectivity array where the points for a particular cell starts. The size of this array is equal to the number of cells in the set plus 1. The first entry is expected to be 0 (since the connectivity of the first cell is at the start of the connectivity array). The last entry, which does not correspond to any cell, should be the size of the connectivity array.

One important item that is missing from this list of arrays is a count of the number of indices associated with each cell. This is not explicitly represented in VTK-m's mesh structure because it can be implicitly derived from the offsets array by subtracting consecutive entries. However, it is usually the case when building an explicit



vtkm::CELL_SHAPE_VERTEX
vtkm::CellShapeTagVertex

vtkm::CELL_SHAPE_LINE
vtkm::CellShapeTagLine

vtkm::CELL_SHAPE_POLY_LINE
vtkm::CellShapeTagPolyLine

vtkm::CELL_SHAPE_TRIANGLE
vtkm::CellShapeTagTriangle

vtkm::CELL_SHAPE_POLYGON
vtkm::CellShapeTagPolygon

vtkm::CELL_SHAPE_QUAD
vtkm::CellShapeTagQuad

vtkm::CELL_SHAPE_TETRA
vtkm::CellShapeTagTetra

vtkm::CELL_SHAPE_HEXAHEDRON
vtkm::CellShapeTagHexahedron

vtkm::CELL_SHAPE_WEDGE
vtkm::CellShapeTagWedge

vtkm::CELL_SHAPE_PYRAMID
vtkm::CellShapeTagPyramid

Figure 7.1: Basic Cell Shapes

mesh that you will have an array of these counts rather than the offsets. It is for this reason that VTK-m contains mechanisms to build an explicit data set with a "num indices" arrays rather than an offsets array.

The vtkm::cont::DataSetBuilderExplicit class can be used to create data sets with explicit meshes. DataSetBuilderExplicit has several versions of a method named Create. Generally, these methods take the shapes, number of indices, and connectivity arrays as well as an array of point coordinates. These arrays can be given in std::vector objects, and the data are copied into the DataSet created.

The following example creates a mesh like the one shown in Figure 7.2.

Example 7.4: Creating an explicit mesh with DataSetBuilderExplicit.

```
// Array of point coordinates.
std::vector<vtkm::Vec3f_32> pointCoordinates;
pointCoordinates.push_back(vtkm::Vec3f_32(1.1f, 0.0f, 0.0f));
pointCoordinates.push_back(vtkm::Vec3f_32(0.2f, 0.4f, 0.0f));
pointCoordinates.push_back(vtkm::Vec3f_32(0.9f, 0.6f, 0.0f));
pointCoordinates.push_back(vtkm::Vec3f_32(1.4f, 0.5f, 0.0f));
```

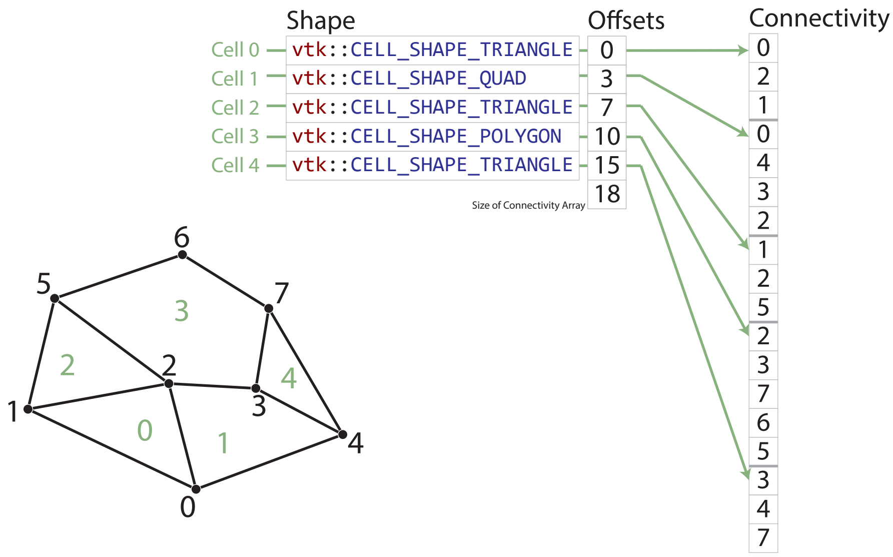


Figure 7.2: An example explicit mesh.

```cpp
  pointCoordinates.push_back(vtkm::Vec3f_32(1.8f, 0.3f, 0.0f));
  pointCoordinates.push_back(vtkm::Vec3f_32(0.4f, 1.0f, 0.0f));
  pointCoordinates.push_back(vtkm::Vec3f_32(1.0f, 1.2f, 0.0f));
  pointCoordinates.push_back(vtkm::Vec3f_32(1.5f, 0.9f, 0.0f));

  // Array of shapes.
  std::vector<vtkm::UInt8> shapes;
  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
  shapes.push_back(vtkm::CELL_SHAPE_QUAD);
  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
  shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);

  // Array of number of indices per cell.
  std::vector<vtkm::IdComponent> numIndices;
  numIndices.push_back(3);
  numIndices.push_back(4);
  numIndices.push_back(3);
  numIndices.push_back(5);
  numIndices.push_back(3);

  // Connectivity array.
  std::vector<vtkm::Id> connectivity;
  connectivity.push_back(0); // Cell 0
  connectivity.push_back(2);
  connectivity.push_back(1);
  connectivity.push_back(0); // Cell 1
  connectivity.push_back(4);
  connectivity.push_back(3);
  connectivity.push_back(2);
  connectivity.push_back(1); // Cell 2
  connectivity.push_back(2);
  connectivity.push_back(5);
  connectivity.push_back(2); // Cell 3
  connectivity.push_back(3);
  connectivity.push_back(7);
  connectivity.push_back(6);
  connectivity.push_back(5);
  connectivity.push_back(3); // Cell 4
  connectivity.push_back(4);
  connectivity.push_back(7);

  // Copy these arrays into a DataSet.
  vtkm::cont::DataSetBuilderExplicit dataSetBuilder;

  vtkm::cont::DataSet dataSet =
    dataSetBuilder.Create(pointCoordinates, shapes, numIndices, connectivity);
```

Often it is awkward to build your own arrays and then pass them to `DataSetBuilderExplicit`. There also exists an alternate builder class named `vtkm::cont::DataSetBuilderExplicitIterative` that allows you to specify each cell and point one at a time rather than all at once. This is done by calling one of the versions of `AddPoint` and one of the versions of `AddCell` for each point and cell, respectively. The next example also builds the mesh shown in Figure 7.2 except this time using `DataSetBuilderExplicitIterative`.

Example 7.5: Creating an explicit mesh with `DataSetBuilderExplicitIterative`.

```
  vtkm::cont::DataSetBuilderExplicitIterative dataSetBuilder;

  dataSetBuilder.AddPoint(1.1, 0.0, 0.0);
  dataSetBuilder.AddPoint(0.2, 0.4, 0.0);
  dataSetBuilder.AddPoint(0.9, 0.6, 0.0);
  dataSetBuilder.AddPoint(1.4, 0.5, 0.0);
  dataSetBuilder.AddPoint(1.8, 0.3, 0.0);
  dataSetBuilder.AddPoint(0.4, 1.0, 0.0);
  dataSetBuilder.AddPoint(1.0, 1.2, 0.0);
  dataSetBuilder.AddPoint(1.5, 0.9, 0.0);

  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
  dataSetBuilder.AddCellPoint(0);
  dataSetBuilder.AddCellPoint(2);
  dataSetBuilder.AddCellPoint(1);

  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_QUAD);
  dataSetBuilder.AddCellPoint(0);
  dataSetBuilder.AddCellPoint(4);
  dataSetBuilder.AddCellPoint(3);
  dataSetBuilder.AddCellPoint(2);

  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
  dataSetBuilder.AddCellPoint(1);
  dataSetBuilder.AddCellPoint(2);
  dataSetBuilder.AddCellPoint(5);

  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_POLYGON);
  dataSetBuilder.AddCellPoint(2);
  dataSetBuilder.AddCellPoint(3);
  dataSetBuilder.AddCellPoint(7);
  dataSetBuilder.AddCellPoint(6);
  dataSetBuilder.AddCellPoint(5);

  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
  dataSetBuilder.AddCellPoint(3);
  dataSetBuilder.AddCellPoint(4);
  dataSetBuilder.AddCellPoint(7);

  vtkm::cont::DataSet dataSet = dataSetBuilder.Create();
```

7.1.4 Add Fields

In addition to creating the geometric structure of a data set, it is usually important to add fields to the data. Fields describe numerical data associated with the topological elements in a cell. They often represent a physical quantity (such as temperature, mass, or volume fraction) but can also represent other information (such as indices or classifications).

The easiest way to define fields in a data set is to use the `vtkm::cont::DataSetFieldAdd` class. This class works on `DataSet`s of any type. It has methods named `AddPointField` and `AddCellField` that define a field for either points or cells. Every field must have an associated field name.

Both `AddPointField` and `AddCellField` are overloaded to accept arrays of data in different structures. Field arrays can be passed as standard C arrays or as `std::vector`s, in which case the data are copied. Field arrays can also be passed in a `ArrayHandle` (introduced later in this book), in which case the data are not copied.

The following (somewhat contrived) example defines fields for a uniform grid that identify which points and cells are on the boundary of the mesh.

Example 7.6: Adding fields to a `DataSet`.

```
// Make a simple structured data set.
const vtkm::Id3 pointDimensions(20, 20, 10);
const vtkm::Id3 cellDimensions = pointDimensions - vtkm::Id3(1, 1, 1);
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointDimensions);

// This is the helper object to add fields to a data set.
vtkm::cont::DataSetFieldAdd dataSetFieldAdd;

// Create a field that identifies points on the boundary.
std::vector<vtkm::UInt8> boundaryPoints;
for (vtkm::Id zIndex = 0; zIndex < pointDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < pointDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < pointDimensions[0]; xIndex++)
    {
      if ((xIndex == 0) || (xIndex == pointDimensions[0] - 1) || (yIndex == 0) ||
          (yIndex == pointDimensions[1] - 1) || (zIndex == 0) ||
          (zIndex == pointDimensions[2] - 1))
      {
        boundaryPoints.push_back(1);
      }
      else
      {
        boundaryPoints.push_back(0);
      }
    }
  }
}

dataSetFieldAdd.AddPointField(dataSet, "boundary_points", boundaryPoints);

// Create a field that identifies cells on the boundary.
std::vector<vtkm::UInt8> boundaryCells;
for (vtkm::Id zIndex = 0; zIndex < cellDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < cellDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < cellDimensions[0]; xIndex++)
    {
      if ((xIndex == 0) || (xIndex == cellDimensions[0] - 1) || (yIndex == 0) ||
          (yIndex == cellDimensions[1] - 1) || (zIndex == 0) ||
          (zIndex == cellDimensions[2] - 1))
      {
        boundaryCells.push_back(1);
      }
      else
      {
        boundaryCells.push_back(0);
      }
    }
  }
}

dataSetFieldAdd.AddCellField(dataSet, "boundary_cells", boundaryCells);
```

7.2 Cell Sets

A cell set determines the topological structure of the data in a data set. Fundamentally, any cell set is a collection of cells, which typically (but not always) represent some region in space. 3D cells are made up of points, edges, and faces. (2D cells have only points and edges, and 1D cells have only points.) Figure 7.3 shows the relationship between a cell's shape and these topological elements. The arrangement of these points, edges, and faces is defined by the *shape* of the cell, which prescribes a specific ordering of each. The basic cell shapes provided by VTK-m are discussed in detail in Section 25.1 starting on page 193.

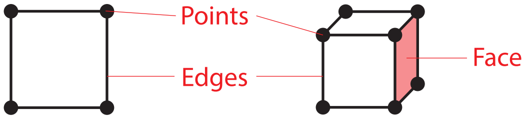


Figure 7.3: The relationship between a cell shape and its topological elements (points, edges, and faces).

There are multiple ways to express the connections of a cell set, each with different benefits and restrictions. These different cell set types are managed by different cell set classes in VTK-m. All VTK-m cell set classes inherit from `vtkm::cont::CellSet`. The two basic types of cell sets are structured and explicit, and there are several variations of these types.

7.2.1 Structured Cell Sets

A `vtkm::cont::CellSetStructured` defines a 1-, 2-, or 3-dimensional grid of points with lines, quadrilaterals, or hexahedra, respectively, connecting them. The topology of a `CellSetStructured` is specified by simply providing the dimensions, which is the number of points in the i, j, and k directions of the grid of points. The number of points is implicitly $i \times j \times k$ and the number of cells is implicitly $(i-1) \times (j-1) \times (k-1)$ (for 3D grids). Figure 7.4 demonstrates this arrangement.

The big advantage of using `vtkm::cont::CellSetStructured` to define a cell set is that it is very space efficient because the entire topology can be defined by the three integers specifying the dimensions. Also algorithms can be optimized for `CellSetStructured`'s regular nature. However, `CellSetStructured`'s strictly regular grid structure also limits its applicability. A structured cell set can only be a dense grid of lines, quadrilaterals, or hexahedra. It cannot represent irregular data well.

Many data models in other software packages, such as the one for VTK, make a distinction between uniform, rectilinear, and curvilinear grids. VTK-m's cell sets do not. All three of these grid types are represented by `CellSetStructured`. This is because in a VTK-m data set the cell set and the coordinate system are defined

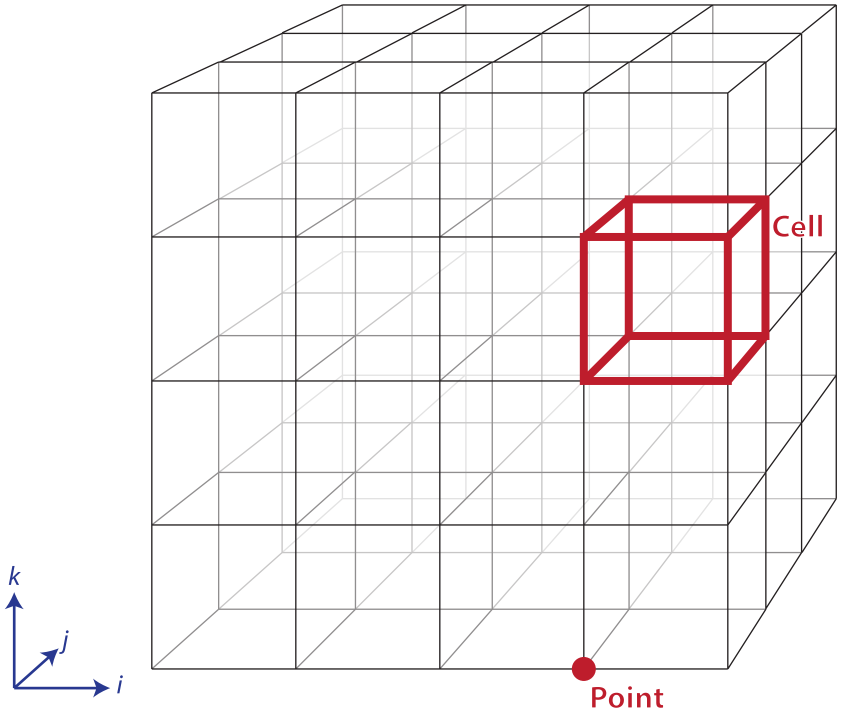


Figure 7.4: The arrangement of points and cells in a 3D structured grid.

independently and used interchangeably. A structured cell set with uniform point coordinates makes a uniform grid. A structured cell set with point coordinates defined irregularly along coordinate axes makes a rectilinear grid. And a structured cell set with arbitrary point coordinates makes a curvilinear grid. The point coordinates are defined by the data set's coordinate system, which is discussed in Section 7.4 starting on page 41.

7.2.2 Explicit Cell Sets

A `vtkm::cont::CellSetExplicit` defines an irregular collection of cells. The cells can be of different types and connected in arbitrary ways. The types of cell sets are listed in Figure 7.5. This is done by explicitly providing for each cell a sequence of points that defines the cell.

An explicit cell set is defined with a minimum of three arrays. The first array identifies the shape of each cell. (Identifiers for cell shapes are shown in Figure 7.5.) The second array has a sequence of point indices that make up each cell. The third array identifies an offset into the second array where the point indices for each cell is found plus an extra entry at the end set to the size of the second array. Figure 7.6 shows a simple example of an explicit cell set.

An explicit cell set can also identify the number of indices defined for each cell by subtracting consecutive entries in the offsets array. It is often the case when creating a `CellSetExplicit` that you have an array containing the number of indices rather than the offsets. Such an array can be converted to an offsets array that can be used with `CellSetExplicit` by using the `vtkm::cont::ConvertNumIndicesToOffsets` convenience function.

`vtkm::cont::ExplicitCellSet` is a powerful representation for a cell set because it can represent an arbitrary collection of cells. However, because all connections must be explicitly defined, `ExplicitCellSet` requires a significant amount of memory to represent the topology.

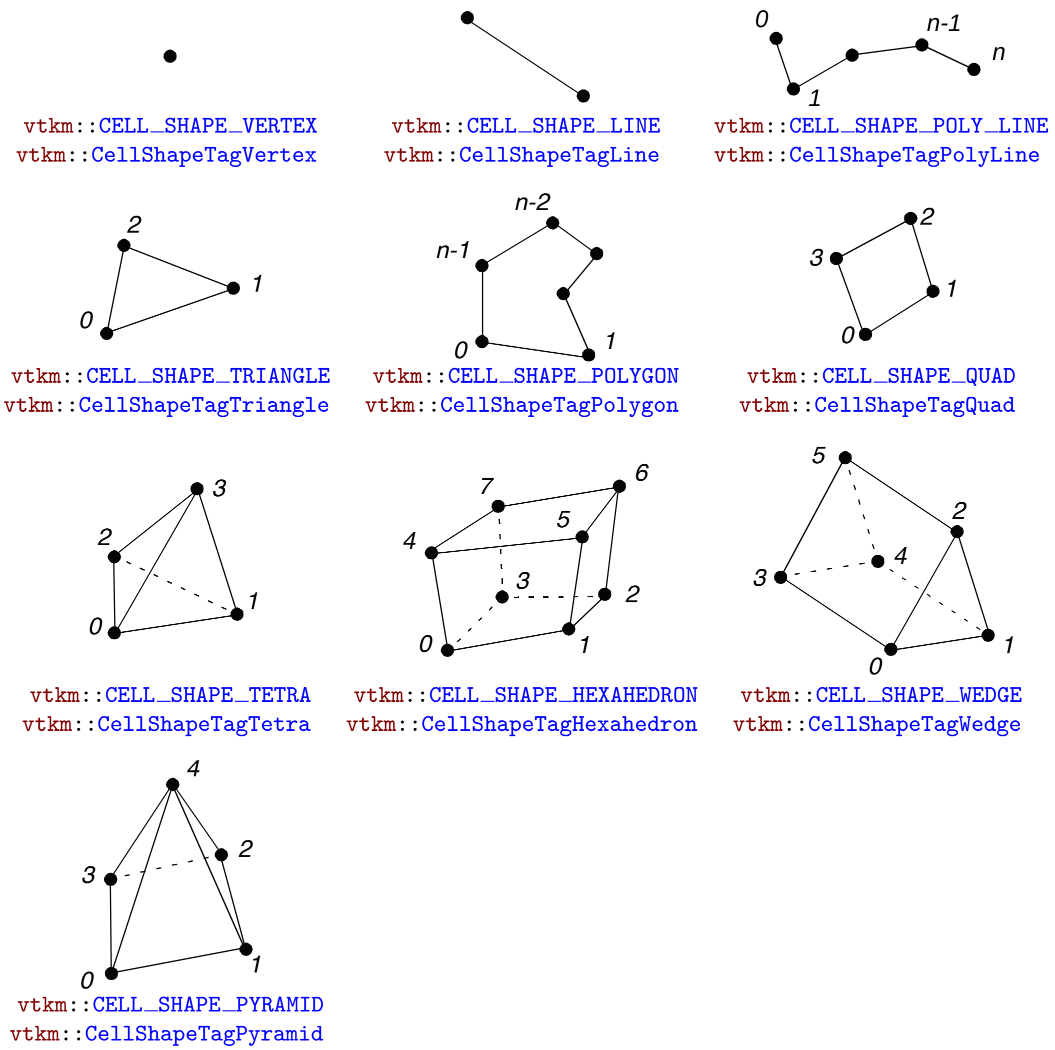


Figure 7.5: Basic Cell Shapes in a CellSetExplicit.

An important specialization of an explicit cell set is vtkm::cont::CellSetSingleType. CellSetSingleType is an explicit cell set constrained to contain cells that all have the same shape and all have the same number of points. So for example if you are creating a surface that you know will contain only triangles, CellSetSingleType is a good representation for these data.

Using CellSetSingleType saves memory because the array of cell shapes and the array of point counts no longer need to be stored. CellSetSingleType also allows VTK-m to skip some processing and other storage required for general explicit cell sets.

7.2.3 Cell Set Permutations

A vtkm::cont::CellSetPermutation rearranges the cells of one cell set to create another cell set. This restructuring of cells is not done by copying data to a new structure. Rather, CellSetPermutation establishes a look-up from one cell structure to another. Cells are permuted on the fly while algorithms are run.

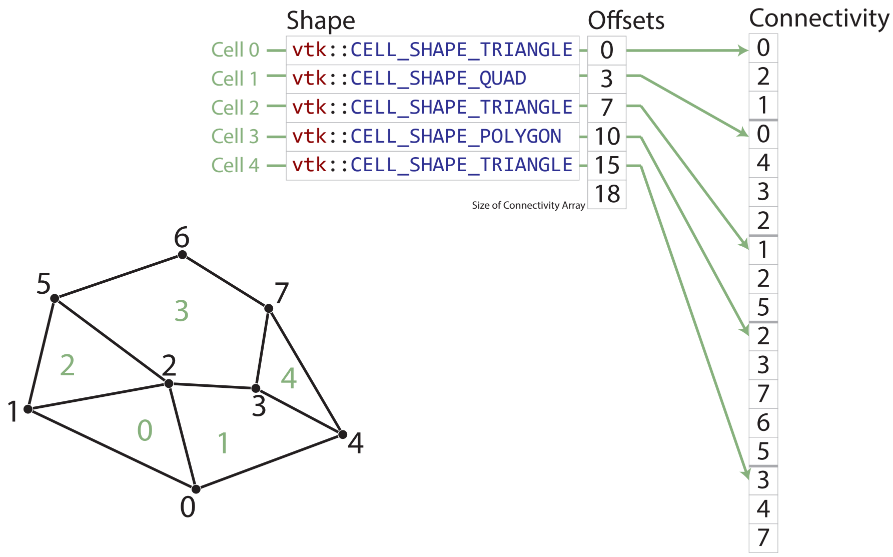


Figure 7.6: Example of cells in a CellSetExplicit and the arrays that define them.

A CellSetPermutation is established by providing a mapping array that for every cell index provides the equivalent cell index in the cell set being permuted. CellSetPermutation is most often used to mask out cells in a data set so that algorithms will skip over those cells when running.

Did you know?

Although CellSetPermutation can mask cells, it cannot mask points. All points from the original cell set are available in the permuted cell set regardless of whether they are used.

The following example uses vtkm::cont::CellSetPermutation with a counting array to expose every tenth cell. This provides a simple way to subsample a data set.

Example 7.7: Subsampling a data set with CellSetPermutation.

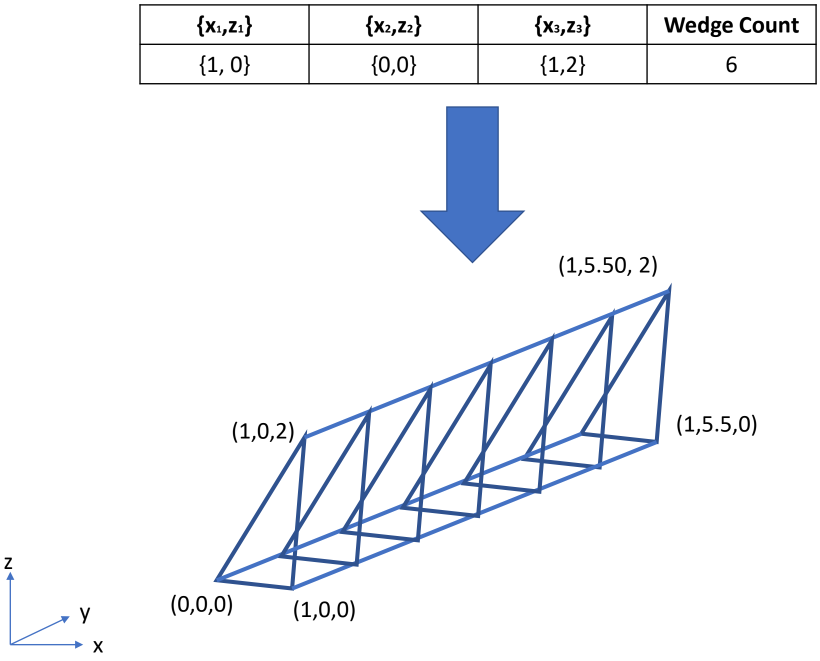
```
// Create a simple data set.
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet originalDataSet = dataSetBuilder.Create(vtkm::Id3(33, 33, 26));
vtkm::cont::CellSetStructured<3> originalCellSet;
originalDataSet.GetCellSet().CopyTo(originalCellSet);

// Create a permutation array for the cells. Each value in the array refers
// to a cell in the original cell set. This particular array selects every
// 10th cell.
vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);

// Create a permutation of that cell set containing only every 10th cell.
vtkm::cont::CellSetPermutation<vtkm::cont::CellSetStructured<3>,
                               vtkm::cont::ArrayHandleCounting<vtkm::Id>>
  permutedCellSet(permutationArray, originalCellSet);
```

7.2.4 Cell Set Extrude

A vtkm::cont::CellSetExtrude defines a 3-dimensional extruded mesh representation from 2-dimensional coordinates in the XZ-plane. This is done by providing 2-dimensional coordinates, the number of planes to extrude along the Y-axis, and whether the resulting wedge cellset representation should be a torus or a cylinder.



$\{x_1,z_1\}$	$\{x_2,z_2\}$	$\{x_3,z_3\}$	Wedge Count
{1, 0}	{0,0}	{1,2}	6

Figure 7.7: An example of an extruded wedge from XZ-plane coordinates. Six wedges are extracted from three XZ-plane points.

The extruded mesh is advantageous because it is represented on-the-fly as required, so no additional memory is required. In contrast other forms of cell sets, such as vtkm::cont::CellSetExplicit, need to be explicitly constructed by replicating the vertices and cells. Figure 7.7 shows an example of six wedges extruded from three 2-dimensional coordinates.

7.3 Fields

A field on a data set provides a value on every point in space on the mesh. Fields are often used to describe physical properties such as pressure, temperature, mass, velocity, and much more. Fields are represented in a VTK-m data set as an array where each value is associated with a particular element type of a mesh (such as points or cells). This association of field values to mesh elements and the structure of the cell set determines how the field is interpolated throughout the space of the mesh.

Fields are manged by the vtkm::cont::Field class. The Field object internally holds a reference to an array in a type-agnostic way. Filters and other VTK-m units will determine the type of the array and pull it out of the Field.

Field has a convenience method named GetRange that finds the range of values stored in the field array. The returned value of GetRange is an ArrayHandle containing vtkm::Range values. The ArrayHandle will have as many values as components in the field. So, for example, calling GetRange on a scalar field will return an ArrayHandle with exactly 1 entry in it. Calling GetRange on a field of 3D vectors will return an ArrayHandle with exactly 3 entries corresponding to each of the components in the range. Details on how to get data from an ArrayHandle them is given in Chapter 27.

7.4 Coordinate Systems

A coordinate system determines the location of a mesh's elements in space. The spatial location is described by providing a 3D vector at each point that gives the coordinates there. The point coordinates can then be interpolated throughout the mesh.

Coordinate systems are managed by the vtkm::cont::CoordinateSystem class. In actuality, a coordinate system is just a field with a special meaning, and so the CoordinateSystem class inherits from the Field class. CoordinateSystem constrains the field to be associated with points and typically has 3D floating point vectors for values.

In addition to all the methods provided by the Field superclass, the CoordinateSystem also provides a GetBounds convenience method that returns a vtkm::Bounds object giving the spatial bounds of the coordinate system.

It is typical for a DataSet to have one coordinate system defined, but it is possible to define multiple coordinate systems. This is helpful when there are multiple ways to express coordinates. For example, positions in geographic may be expressed as Cartesian coordinates or as latitude-longitude coordinates. Both are valid and useful in different ways.

It is also valid to have a DataSet with no coordinate system. This is useful when the structure is not rooted in physical space. For example, if the cell set is representing a graph structure, there might not be any physical space that has meaning for the graph.

7.5 Partitioned Data Sets

A partitioned data set, implemented with vtkm::cont::PartitionedDataSet, comprises a set of vtkm::cont::DataSet objects. The PartitionedDataSet interface allows for adding, replacing, and querying DataSets in its list with the following methods.

GetNumberOfPartitions Returns the number of partitions stored in the PartitionedDataSet.

GetPartition Returns the DataSet at a given index.

GetPartitions Returns all of the DataSets stored in the PartitionedDataSet in a std::vector.

AppendPartition Adds a given DataSet to the end of the list of partitions.

AppendPartitions Given a list of DataSet objects, appends this list to the end of the list of partitions. This list can be given as a std::vector or it can be an initializer list (declared in { } curly braces).

InsertPartition Given an index and a DataSet, places the DataSet at the given index and pushes the remaining partitions after it.

ReplacePartition Given an index and a DataSet, replaces the partition at that index with the new DataSet.

GetField Retrieves a vtkm::cont::Field object from the DataSet at a given index.

The following example creates a vtkm::cont::PartitionedDataSet containing two uniform grid data sets.

Example 7.8: Creating a PartitionedDataSet.

```
// Create two uniform data sets
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet1 = dataSetBuilder.Create(vtkm::Id3(10, 10, 10));
vtkm::cont::DataSet dataSet2 = dataSetBuilder.Create(vtkm::Id3(30, 30, 30));

// Add the datasets to a multi block
vtkm::cont::PartitionedDataSet partitionedData;
partitionedData.AppendPartitions({ dataSet1, dataSet2 });
```

It is always possible to retrieve the independent blocks in a PartitionedDataSet, from which you can iterate and get information about the data. However, VTK-m provides several helper functions to collect metadata information about the collection as a whole.

vtkm::cont::BoundsCompute Queries the bounds of all the DataSets contained in the given PartitionedDataSet and returns a vtkm::Bounds object encompassing the conglomerate data.

vtkm::cont::BoundsGlobalCompute An MPI version of BoundsCompute that also finds the bounds around the conglomerate data across all processes. All MPI processes must call this method.

vtkm::cont::FieldRangeCompute Given a PartitionedDataSet, the name of a field, and (optionally) an association of the field, returns the minimum and maximum value of that field over all the contained blocks. The result is returned in a ArrayHandle of vtkm::Range objects in the same manner as the vtkm::cont::Field::GetRange method (see Section 7.3).

vtkm::cont::FieldRangeGlobalCompute An MPI version of FieldRangeCompute that also finds the field ranges over all blocks on all processes. All MPI processes must call this method.

The following example illustrates a spatial bounds query and a field range query on a vtkm::cont::PartitionedDataSet.

Example 7.9: Queries on a PartitionedDataSet.

```
// Get the bounds of a multi-block data set
vtkm::Bounds bounds = vtkm::cont::BoundsCompute(partitionedData);

// Get the overall min/max of a field named "cellvar"
vtkm::cont::ArrayHandle<vtkm::Range> cellvarRanges =
  vtkm::cont::FieldRangeCompute(partitionedData, "cellvar");

// Assuming the "cellvar" field has scalar values, then cellvarRanges has one entry
vtkm::Range cellvarRange = cellvarRanges.GetPortalConstControl().Get(0);
```

Did you know?

The aforementioned functions for querying a PartitionedDataSet *object also work on* DataSet *objects. This is particularly useful with the* BoundsGlobalCompute *and* FieldRangeGlobalCompute *to manage distributed parallel objects.*

Filters can be executed on PartitionedDataSet objects in a similar way they are executed on DataSet objects. In both cases, the Execute method is called on the filter giving data object as an argument.

Example 7.10: Applying a filter to multi block data.

```
vtkm::filter::CellAverage cellAverage;
cellAverage.SetActiveField("pointvar", vtkm::cont::Field::Association::POINTS);

vtkm::cont::PartitionedDataSet results = cellAverage.Execute(partitionedData);
```

CHAPTER

EIGHT

FILE I/O

Before VTK-m can be used to process data, data need to be loaded into the system. VTK-m comes with a basic file I/O package to get started developing very quickly. All the file I/O classes are declared under the `vtkm::io` namespace.

> **Did you know?**
>
> *Files are just one of many ways to get data in and out of VTK-m. In later chapters we explore ways to define VTK-m data structures of increasing power and complexity. In particular, Section 7.1 describes how to build VTK-m data set objects and Section 36.3 documents how to adapt data structures defined in other libraries to be used directly in VTK-m.*

8.1 Readers

All reader classes provided by VTK-m are located in the `vtkm::io::reader` namespace. The general interface for each reader class is to accept a filename in the constructor and to provide a `ReadDataSet` method to load the data from disk.

The data in the file are returned in a `vtkm::cont::DataSet` object as described in Chapter 7, but it is sufficient to known that a `DataSet` can be passed around readers, writers, filters, and rendering units.

8.1.1 Legacy VTK File Reader

Legacy VTK files are a simple open format for storing visualization data. These files typically have a `.vtk` extension. Legacy VTK files are popular because they are simple to create and read and are consequently supported by a large number of tools. The format of legacy VTK files is well documented in *The VTK User's Guide*.[1] Legacy VTK files can also be read and written with tools like ParaView and VisIt.

Legacy VTK files can be read using the `vtkm::io::reader::VTKDataSetReader` class. The constructor for this class takes a string containing the filename. The `ReadDataSet` method reads the data from the previously indicated file and returns a `vtkm::cont::DataSet` object, which can be used with filters and rendering.

Example 8.1: Reading a legacy VTK file.

[1] A free excerpt describing the file format is available at http://www.vtk.org/Wiki/File:VTK-File-Formats.pdf.

```
#include <vtkm/io/reader/VTKDataSetReader.h>

vtkm::cont::DataSet OpenDataFromVTKFile()
{
  vtkm::io::reader::VTKDataSetReader reader("data.vtk");

  return reader.ReadDataSet();
}
```

8.2 Writers

All writer classes provided by VTK-m are located in the vtkm::io::writer namespace. The general interface for each writer class is to accept a filename in the constructor and to provide a WriteDataSet method to save data to the disk. The WriteDataSet method takes a vtkm::cont::DataSet object as an argument, which contains the data to write to the file.

8.2.1 Legacy VTK File Writer

Legacy VTK files can be written using the vtkm::io::writer::VTKDataSetWriter class. The constructor for this class takes a string containing the filename. The WriteDataSet method takes a vtkm::cont::DataSet object and writes its data to the previously indicated file.

Example 8.2: Writing a legacy VTK file.

```
#include <vtkm/io/writer/VTKDataSetWriter.h>

void SaveDataAsVTKFile(vtkm::cont::DataSet data)
{
  vtkm::io::writer::VTKDataSetWriter writer("data.vtk");

  writer.WriteDataSet(data);
}
```

CHAPTER

NINE

RUNNING FILTERS

Filters are functional units that take data as input and write new data as output. Filters operate on vtkm::cont::DataSet objects, which are described in Chapter7.

> Did you know?
>
> *The structure of filters in VTK-m is significantly simpler than their counterparts in VTK. VTK filters are arranged in a dataflow network (a.k.a. a visualization pipeline) and execution management is handled automatically. In contrast, VTK-m filters are simple imperative units, which are simply called with input data and return output data.*

VTK-m comes with several filters ready for use, and in this chapter we will give a brief overview of these filters. All VTK-m filters are currently defined in the vtkm::filter namespace. We group filters based on the type of operation that they do and the shared interfaces that they have. Later Part III describes the necessary steps in creating new filters in VTK-m.

Different filters will be used in different ways, but the basic operation of all filters is to instantiate the filter class, set the state parameters on the filter object, and then call the filter's Execute method. The Execute method takes a vtkm::cont::DataSet and returns a new DataSet, which contains the modified data. The Execute method can alternately take a vtkm::cont::PartitionedDataSet object, which is a composite of DataSet objects. In this case Execute will return another PartitionedDataSet object.

The following example provides a simple demonstration of using a filter. It specifically uses the point elevation filter to estimate the air pressure at each point based on its elevation.

Example 9.1: Using PointElevation, which is a field filter.

```
VTKM_CONT
vtkm::cont::DataSet ComputeAirPressure(vtkm::cont::DataSet dataSet)
{
  vtkm::filter::PointElevation elevationFilter;

  // Use the elevation filter to estimate atmospheric pressure based on the
  // height of the point coordinates. Atmospheric pressure is 101325 Pa at
  // sea level and drops about 12 Pa per meter.
  elevationFilter.SetLowPoint(0.0, 0.0, 0.0);
  elevationFilter.SetHighPoint(0.0, 0.0, 2000.0);
  elevationFilter.SetRange(101325.0, 77325.0);

  elevationFilter.SetUseCoordinateSystemAsField(true);

  elevationFilter.SetOutputFieldName("pressure");
16
17    vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
18
19    return result;
20  }
```

We see that this example follows the previously described proceedure of constructing the filter (line 4), setting the state parameters (lines 9–15), and finally executing the filter on a DataSet (line 17).

Every vtkm::cont::DataSet object contains a list of *fields*, which describe some numerical value associated with different parts of the data set in space. Fields often represent physical properties such as temperature, pressure, or velocity. Fields are identified with string names. There are also special fields called coordinate systems that describe the location of points in space. Field are mentioned here because they are often used as input data to the filter's operation and filters often generate new fields in the output. This is the case in Example 9.1. In line 13 the coordinate system is set as the input field and in line 15 the name to use for the generated output field is selected.

9.1 Provided Filters

VTK-m comes with the implementation of many filters.

9.1.1 Cell Average

vtkm::filter::CellAverage is the cell average filter. It will take a data set with a collection of cells and a field defined on the points of the data set and create a new field defined on the cells. The values of this new derived field are computed by averaging the values of the input field at all the incident points. This is a simple way to convert a point field to a cell field.

The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName` method.

CellAverage provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.2 Clean Grid

vtkm::filter::CleanGrid is a filter that converts a cell set to an explicit representation and potentially removes redundant or unused data. It does this by iterating over all cells in the data set, and for each one creating the explicit cell representation that is stored in the output. (Explicit cell sets are described in Section 7.2.2.) One benefit of using CleanGrid is that it can optionally remove unused points and combine coincident points. Another benefit is that the resulting cell set will be of a known specific type.

Common Errors

The result of vtkm::filter::CleanGrid *is not necessarily smaller, memory-wise, than its input. For example, "cleaning" a data set with a structured topology will actually result in a data set that requires much more memory to store an explicit topology.*

CleanGrid provides the following methods.

SetCompactPointFields/GetCompactPointFields Sets a Boolean flag that determines whether unused points are removed from the output. If true (the default), then the output data set will have a new coordinate system containing only those points being used by the cell set, and the indices of the cells will be adjusted to the new ordering of points.

SetMergePoints/GetMergePoints Sets a Boolean flag that determines whether points coincident in space are merged into a single point. If true (the default), then the output data set will have a new coordinate system containing containing only points that are unique in space, and the indices of the cells will be adjusted to the new set of points. The tolerance parameters control the proximity used for points to be considered coincident.

SetTolerance/GetTolerance Defines the tolerance used when determining whether two points are considered coincident. Because floating point parameters have limited precision, point coordinates that are essentially the same might not be bit-wise exactly the same. Thus, the CleanGrid filter has the ability to find and merge points that are close but perhaps not exact. The default tolerance is 10^{-6}.

SetToleranceIsAbsolute/GetToleranceIsAbsolute Sets a Boolean flag that determines whether the tolerance parameter should be considered relative to the size of the data set. If false (the default), then the tolerance is multiplied by the length of the diagonal of the bounds of the data being processed. If true, then the tolerance value is used as is.

SetRemoveDegenerateCellsGetRemoveDegenerateCells Sets a Boolean flag that determines whether degenerate cells should be removed. If true (the default), then the CleanGrid filter will look for repeated points in cells and, if the repeated points cause the cell to drop dimensionality, the cell is removed. This is particularly useful when point merging is on as this operation can create degenerate cells.

SetFastMerge/GetFastMerge Sets a Boolean flag that determines whether to use a faster but less accurate method for finding coincident points. If true (the default), some corners are cut when computing coincident points. This will make the point merge step go faster but the tolerance will not be strictly followed. If false, then extra steps will be taken to ensure that all points within tolerance are merged and that only points within tolerance are merged. This flag has no effect if point merging is off.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.3 Clip with Field

Clipping is an operation that removes regions from the data set based on a user-provided value or function. The `vtkm::filter::ClipWithField` filter takes a clip value as an argument and removes regions where a named scalar field is below (or above) that value. (A companion filter that discards a region of the data based on an implicit function is described in Section 9.1.4.)

The result of `ClipWithField` is a volume. If a cell has field values at its vertices that are all below the specified value, then it will be discarded entirely. Likewise, if a cell has field values at its vertices that are all above the specified value, then it will be retained in its entirety. If a cell has some vertices with field values below the specified value and some above, then the cell will be split into the portions above the value (which will be retained) and the portions below the value (which will be discarded).

This operation is sometimes called an *isovolume* because it extracts the volume of a mesh that is inside the iso-region of a scalar. This is in contrast to an *isosurface* (also known as a *contour*), which extracts only the surface of that iso-value. (See Section 9.1.7 for extracting an isosurface.) `ClipWithField` is also similar to a threshold operation, which extracts cells based on the value of field. The difference is that threshold will either keep or remove entire cells based on the field values whereas clip with carve cells that straddle the valid regions. (See section 9.1.24 for threshold extraction.)

`ClipWithField` provides the following methods.

`SetClipValue/GetClipValue` Specifies the field value for the clip operation. Regions where the active field is less than this value are clipped away from each input cell.

`SetInvertClip` Specifies if the result for the clip filter should be inverted. If set to false (the default), regions where the active field is less than the specified clip value are removed. If set to true, regions where the active field is more than the specified clip value are removed.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.2: Using `ClipWithField`.

```
// Create an instance of a clip filter that discards all regions with scalar
// value less than 25.
vtkm::filter::ClipWithField clip;
```

```
clip.SetClipValue(25.0);
clip.SetActiveField("pointvar");

// Execute the clip filter
vtkm::cont::DataSet outData = clip.Execute(inData);
```

9.1.4 Clip with Implicit Function

Clipping is an operation that removes regions from the data set based on a user-provided value or function. The vtkm::filter::ClipWithImplicitFunction takes an implicit function as an argument. See Chapter 14 for more detail. ClipWithImplicitFunction discards regions of the original data set according to the values of the implicit function. (A companion filter that discards a region of the data based on the value of a scalar field is described in Section 9.1.3.)

The result of ClipWithImplicitFunction is a volume. If a cell has its vertices positioned all outside the implicit function, then it will be discarded entirely. Likewise, if a cell its vertices all inside the implicit function, then it will be retained in its entirety. If a cell has some vertices inside the implicit function and some outside, then the cell will be split into the portions inside (which will be retained) and the portions outside (which will be discarded).

ClipWithImplicitFunction provides the following methods.

SetImplicitFunction/GetImplicitFunction Specifies the implicit function to be used to perform the clip operation. The filter does not directly take a vtkm::ImplicitFunction but rather an ImplicitFunction wrapped inside of a vtkm::cont::ImplicitFunctionHandle. The ImplicitFunctionHandle manages the use of the virtual methods in ImplicitFunction on different devices, which may be using different memory spaces or require different processor instructions. An ImplicitFunctionHandle is easily created with the vtkm::cont::make_ImplicitFunctionHandle function.

SetInvertClip Specifies whether the result of the clip filter should be inverted. If set to false (the default), all regions where the implicit function is negative will be removed. If set to true, all regions where the implicit function is positive will be removed.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

In the example provided below the vtkm::Sphere implicit function is used. This function evaluates to a negative value if points from the original dataset occur within the sphere, evaluates to 0 if the points occur on the surface of the sphere, and evaluates to a positive value if the points occur outside the sphere.

Example 9.3: Using ClipWithImplicitFunction.

```
// Parameters needed for implicit function
vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);

// Create an instance of a clip filter with this implicit function.
vtkm::filter::ClipWithImplicitFunction clip;
clip.SetImplicitFunction(
  vtkm::cont::make_ImplicitFunctionHandle(implicitFunction));
```

```

// By default, ClipWithImplicitFunction will remove everything inside the sphere.
// Set the invert clip flag to keep the inside of the sphere and remove everything
// else.
clip.SetInvertClip(true);

// Execute the clip filter
vtkm::cont::DataSet outData = clip.Execute(inData);
```

9.1.5 Connected Components

Connected components in a mesh are groups of mesh elements that are connected together in some way. For example, if two cells are neighbors, then they are in the same component. Likewise, a cell is also in the same component as its neighbor's neighbors as well as their neighbors and so on. Connected components help identify when features in a simulation fragment or meld.

VTK-m provides two types of connected components filters. The first filter follows topological connections to find cells that are literally connected together. The second filter takes a structured cell set and a field that classifies each cell and finds connected components where all the cells have the same field value.

Cell Connectivity

The `vtkm::filter::CellSetConnectivity` filter finds groups of cells that are connected together through their topology. Two cells are considered connected if they share an edge. `CellSetConnectivity` identifies some number of components and assigns each component a unique integer.

The result of the filter is a cell field of type `vtkm::Id`. Each entry in the cell field will be a number that identifies to which component the cell belongs. By default, this output cell field is named "component". Although an input field can be specified, it is ignored.

`CellSetConnectivity` provides the following methods.

`SetActiveField`/`GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField`/`GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem`/`GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName`/`GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass`/`GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Image Field

The `vtkm::filter::ImageConnectivity` filter finds groups of points that have the same field value and are connected together through their topology. Any point is considered to be connected to its Moore neighborhood:

8 neighboring points for 2D and 27 neighboring points for 3D. As the name implies, `ImageConnectivity` only works on data with a structured cell set. You will get an error if you use any other type of cell set.

The active field passed to the filter must be associated with the points.

The result of the filter is a point field of type `vtkm::Id`. Each entry in the point field will be a number that identifies to which component the cell belongs. By default, this output point field is named "component".

`ImageConnectivity` provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.6 Coordinate System Transforms

VTK-m provides multiple filters to translate between different coordiante systems.

Cylindrical Coordinate System Transform

`vtkm::filter::CylindricalCoordinateSystemTransform` is a coordinate system transformation filter. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a cartesian coordinate system to a cylindrical coordinate system. The order for cylindrical coordinates is (R, θ, Z)

The default name for the output field is "cylindricalCoordinateSystemTransform", but that can be overridden as always using the `SetOutputFieldName` method.

In addition to the standard `SetOutputFieldName` and `Execute` methods, `CylindricalCoordinateSystemTransform` provides the following methods.

`SetCartesianToCylindrical` This method specifies a transformation from cartesian to cylindrical coordinates.

`SetCylindricalToCartesian` This method specifies a transformation from cylindrical to cartesian coordinates.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Spherical Coordinate System Transform

`vtkm::filter::SphericalCoordinateSystemTransform` is a coordinate system transformation filter. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a cartesian coordinate system to a spherical coordinate system. The order for spherical coordinates is (R,θ,ϕ)

The default name for the output field is "sphericalCoordinateSystemTransform", but that can be overridden as always using the `SetOutputFieldName` method.

In addition the standard `SetOutputFieldName` and `Execute` methods, `CylindricalCoordinateSystemTransform` provides the following methods.

`SetCartesianToSpherical` This method specifies a transformation from cartesian to spherical coordinates.

`SetSphericalToCartesian` This method specifies a transformation from spherical to cartesian coordinates.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.7 Contour

Contouring is one of the most fundamental filters in scientific visualization. A contour is the locus where a field is equal to a particular value. A topographic map showing curves of various elevations often used when hiking in hilly regions is an example of contours of an elevation field in 2 dimensions. Extended to 3 dimensions, a contour gives a surface. Thus, a contour is often called an *isosurface*. The contouring/isosurface algorithm is implemented by `vtkm::filter::Contour`.

`Contour` provides the following methods.

`SetIsoValue/GetIsoValue` Specifies the value on which to extract the contour. The contour will be the surface where the field (provided to `Execute`) is equal to this value.

`SetMergeDuplicatePoints/GetMergeDuplicatePoints` Specifies whether coincident points in the data set should be merged. Because the contour filter (like all filters in VTK-m) runs in parallel, parallel threads can (and often do) create duplicate versions of points. When this flag is set to true, a secondary operation will find all duplicated points and combine them together.

`SetGenerateNormals/GetGenerateNormals` Specifies whether to generate normal vectors for the surface. Normals are used in shading calculations during rendering and can make the surface appear more smooth. By default, the generated normals are based on the gradient of the field being contoured and can be quite expensive to compute. A faster method is available that computes the normals based on the faces of the isosurface mesh, but the normals do not look as good as the gradient based normals. Fast normals can be enabled using the flags described bellow.

`SetComputeFastNormalsForStructured/GetComputeFastNormalsForStructured` Specifies whether to use the fast method of normals computation for Structured data sets. This is only valid if the generate normals flag is set.

`SetComputeFastNormalsForUnstructured/GetComputeFastNormalsForUnstructured` Specifies whether to use the fast method of normals computation for unstructured data sets. This is only valid if the generate normals flag is set.

`SetNormalArrayName/GetNormalArrayName` Specifies the name used for the normals field if it is being created.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.4: Using `Contour`, which is a data set with field filter.

```
vtkm::filter::Contour contour;

contour.SetActiveField("pointvar");
contour.SetIsoValue(10.0);

vtkm::cont::DataSet isosurface = contour.Execute(inData);
```

9.1.8 Cross Product

`vtkm::filter::CrossProduct` computes the cross product of two vector fields for every element in the input data set. The cross product filter computes (PrimaryField × SecondaryField), where both the primary and secondary field are specified using methods on the `CrossProduct` class. The cross product computation works for both point and cell centered vector fields.

`CrossProduct` provides the following methods.

SetPrimaryField/GetPrimaryFieldName Specifies the name of the field to use as input for the primary (first) value of the cross product.

SetUseCoordinateSystemAsPrimaryField/GetUseCoordinateSystemAsPrimaryField Specifies a Boolean flag that determines whether to use point coordinates as the primary input field. Set to false by default. When true, the name for the primary field is ignored.

SetPrimaryCoordinateSystem/GetPrimaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the primary input field. The default index is 0, which is the first coordinate system.

SetSecondaryField/GetSecondaryFieldName Specifies the name of the field to use as input for the secondary (second) value of the cross product.

SetUseCoordinateSystemAsSecondaryField/GetUseCoordinateSystemAsSecondaryField Specifies a Boolean flag that determines whether to use point coordinates as the secondary input field. Set to false by default. When true, the name for the secondary field is ignored.

SetSecondaryCoordinateSystem/GetSecondaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the secondary input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.9 Dot Product

vtkm::filter::DotProduct computes the dot product of two vector fields for every element in the input data set. The dot product filter computes (PrimaryField · SecondaryField), where both the primary and secondary field are specified using methods on the DotProduct class. The dot product computation works for both point and cell centered vector fields.

DotProduct provides the following methods.

SetPrimaryField/GetPrimaryFieldName Specifies the name of the field to use as input for the primary (first) value of the dot product.

SetUseCoordinateSystemAsPrimaryField/GetUseCoordinateSystemAsPrimaryField Specifies a Boolean flag that determines whether to use point coordinates as the primary input field. Set to false by default. When true, the name for the primary field is ignored.

SetPrimaryCoordinateSystem/GetPrimaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the primary input field. The default index is 0, which is the first coordinate system.

SetSecondaryField/GetSecondaryFieldName Specifies the name of the field to use as input for the secondary (second) value of the dot product.

SetUseCoordinateSystemAsSecondaryField/GetUseCoordinateSystemAsSecondaryField Specifies a Boolean flag that determines whether to use point coordinates as the secondary input field. Set to false by default. When true, the name for the secondary field is ignored.

SetSecondaryCoordinateSystem/GetSecondaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the secondary input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.10 External Faces

vtkm::filter::ExternalFaces is a filter that extracts all the external faces from a polyhedral data set. An external face is any face that is on the boundary of a mesh. Thus, if there is a hole in a volume, the boundary of that hole will be considered external. More formally, an external face is one that belongs to only one cell in a mesh.

ExternalFaces provides the following methods.

SetCompactPoints/GetCompactPoints Specifies whether point fields should be compacted. If on, the filter will remove from the output all points that are not used in the resulting surface. If off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

SetPassPolyData/GetPassPolyData Specifies how polygonal data (polygons, lines, and vertices) will be handled. If on (the default), these cells will be passed to the output. If off, these cells will be removed from the output. (Because they have less than 3 topological dimensions, they are not considered to have any "faces.")

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.11 Extract Structured

vtkm::filter::ExtractStructured is a filter that extracts a volume of interest (VOI) from a structured data set. In addition the filter is able to subsample the VOI while doing the extraction.

The output of this filter is a structured dataset. The filter treats input data of any topological dimension (i.e., point, line, plane, or volume) and can generate output data of any topological dimension.

Typical applications of this filter are to extract a slice from a volume for image processing, subsampling large volumes to reduce data size, or extracting regions of a volume with interesting data.

ExtractStructured provides the following methods.

`SetVOI/GetVOI` Specifies what volume of interest (VOI) should be extracted by the filter. By default the VOI is the entire input.

`SetSampleRate/GetSampleRate` Specifies the sample rate of the VOI. Supports sub-sampling on a per dimension basis.

`SetIncludeBoundary/GetIncludeBoundary` Specifies if the VOI is inclusive or exclusive on the boundary of the VOI.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.12 Field to Colors

`vtkm::filter::FieldToColors` takes a field in a data set, looks up each value in a color table, and writes the resulting colors to a new field. The color to be used for each field value is specified using a `vtkm::cont::ColorTable` object. `ColorTable` objects are also used with VTK-m's rendering module and are described in Section 10.8.

`FieldToColors` has three modes it can use to select how it should treat the input field.

`FieldToColors::SCALAR` Treat the field as a scalar field. It is an error to a field of any type that cannot be directly converted to a basic floating point number (such as a vector).

`FieldToColors::MAGNITUDE` Given a vector field, take the magnitude of each field value before looking it up in the color table.

`FieldToColors::COMPONENT` Select a particular component of the vectors in a field to map to colors.

Additionally, `FieldToColors` has different modes in which it can represent colors in its output.

`FieldToColors::RGB` Output colors are represented as RGB values with each component represented by an unsigned byte. Specifically, these are `vtkm::Vec3ui_8` values.

`FieldToColors::RGBA` Output colors are represented as RGBA values with each component represented by an unsigned byte. Specifically, these are `vtkm::Vec4ui_8` values.

`FieldToColors` provides the following methods.

`SetColorTable/GetColorTable` Specifies the `vtkm::cont::ColorTable` object to use to map field values to colors.

`SetMappingMode/GetMappingMode` Specifies the input mapping mode. The value is one of the `FieldToColors::SCALAR`, `FieldToColors::MAGNITUDE`, or `FieldToColors::COMPONENT` selectors described previously.

`SetMappingToScalar` Sets the input mapping mode to scalar. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::SCALAR )`.

`SetMappingToMagnitude` Sets the input mapping mode to vector. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::MAGNITUDE )`.

`SetMappingToComponent` Sets the input mapping mode to component. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::COMPONENT )`.

`IsMappingScalar` Returns true if the input mapping mode is scalar (`FieldToColors`SCALAR).

`IsMappingMagnitude` Returns true if the input mapping mode is magnitude (`FieldToColors::MAGNITUDE`).

`IsMappingComponent` Returns true if the input mapping mode is component (`FieldToColors::COMPONENT`).

`SetMappingComponent/GetMappingComponent` Specifies the component of the vector to use in the mapping. This only has an effect if the input mapping mode is set to `FieldToColors::COMPONENT`.

`SetOutputMode/GetOutputMode` Specifies the output representation of colors. The value is one of the `FieldToColors::RGB` or `FieldToColors::RGBA` selectors described previously.

`SetOutputToRGB` Sets the output representation to 8-bit RGB. Shortcut for `SetOutputMode(vtkm::filter::FieldToColors::RGB )`.

`SetOutputToRGBA` Sets the output representation to 8-bit RGBA. Shortcut for `SetOutputMode(vtkm::filter::FieldToColors::RGBA )`.

`IsOutputRGB` Returns true if the output representation is 8-bit RGB (`FieldToColors::RGB`).

`IsOutputRGBA` Returns true if the output representation is 8-bit RGBA (`FieldToColors::RGBA`).

`SetNumberOfSamplingPoints/GetNumberOfSamplingPoints` Specifies how many samples to use when looking up color values. The implementation of `FieldToColors` first builds an array of color samples to quickly look up colors for particular values. The size of this lookup array can be adjusted with this parameter. By default, an array of 256 colors is used.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.13 Ghost Cell Classification

vtkm::filter::GhostCellClassify adds a cell centered field to the input data set that marks each cell as either vtkm::CellClassification::NORMAL or vtkm::CellClassification::GHOST. The outer layer of cells are marked as GHOST, and the remainder are marked as *CellClassificationNORMAL. This filter only supports uniform and rectilinear data sets. The default field is "vtkmGhostCells".

GhostCellClassify provides the following methods.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.14 Ghost Cell Removal

vtkm::filter::GhostCellRemove is a filter that is used to remove cells from a data set according to a cell centered field that is provided to the filter. The default field used for removal is "vtkmGhostCells". The field is of type vtkm::UInt8, and represents a bit-field to classify each cell. By default, if the input is a structured data set the filter will attempt to output a structured data set. If this is not possible, an explict data set is produced. The field specified for cell removal is not passed to the output.

GhostCellRemove provides the following methods.

RemoveAllGhost Remove all cells where the value is a ghost cell (i.e. vtkm::CellClassification::GHOST).

RemoveByType Remove cells specified by the vtkm::UInt8 using a bitwise "and" operation with the type field. The values in vtkm::CellClassification can be combined with a logical "or" operation to specify the type. Current values of vtkm::CellClassification include: NORMAL, GHOST, and INVALID.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.15 Gradients

vtkm::filter::Gradients computes the gradient of a point based input field for every element in the input data set. The gradient computation can either generate cell center based gradients, which are fast but less accurate, or more accurate but slower point based gradients. The default for the filter is output as cell centered gradients, but can be changed by using the SetComputePointGradient method. The default name for the output fields is "Gradients", but that can be overriden as always using the SetOutputFieldName method.

Gradients provides the following methods.

SetComputePointGradient/GetComputePointGradient Specifies whether we are computing point or cell based gradients. The output field(s) of this filter will be point based if this is enabled.

SetComputeDivergence/GetComputeDivergence Specifies whether the divergence field will be generated. By default the name of the array will be "Divergence" but can be changed by using SetDivergenceName. The field will be a cell field unless ComputePointGradient is enabled. The input array must have 3 components in order to compute this. The default is off.

SetComputeVorticity/GetComputeVorticity Specifies whether the vorticity field will be generated. By default the name of the array will be "Vorticity" but can be changed by using SetVorticityName. The field will be a cell field unless ComputePointGradient is enabled. The input array must have 3 components in order to compute this. The default is off.

SetComputeQCriterion/GetComputeQCriterion Specifies whether the Q-Criterion field will be generated. By default the name of the array will be "QCriterion" but can be changed by using SetQCriterionName. The field will be a cell field unless ComputePointGradient is enabled. The input array must have 3 components in order to compute this. The default is off.

SetComputeGradient/GetComputeGradient Specifies whether the actual gradient field is written to the output. When processing fields that have 3 components it is desirable to compute information such as Divergence, Vorticity, or Q-Criterion without incurring the cost of also having to write out the 3x3 gradient result. The default is on.

SetColumnMajorOrdering/SetRowMajorOrdering When processing input fields that have 3 components, the output will be a a 3x3 gradient. By default VTK-m outputs all matrix like arrays in Row Major ordering (C-Ordering). The ordering can be changed when integrating with libraries like VTK or with FORTRAN codes that use Column Major ordering. The default is Row Major. This setting is only relevant for 3 component input fields when SetComputeGradient is enabled.

SetDivergenceName/GetDivergenceName Specifies the output cell normals field name. The default is "Divergence".

SetVorticityName/GetVorticityName Specifies the output Vorticity field name. The default is "Vorticity"

SetQCriterionName/GetQCriterionName Specifies the output Q-Criterion field name. The default is "QCriterion".

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.16 Histogram

`vtkm::filter::Histogram` computes a histogram of a given field.

The default number of bins in the output histogram is 10, but that can be overridden using the `SetNumberOfBins` method.

The default name for the output fields is "histogram". The name can be overridden as always using the `SetOutputFieldName` method.

`Histogram` provides the following methods.

`SetRange/GetRange` Specifies an explicit range to use to generate the histogram. If no range is set the fields global range is used.

`GetBinDelta` Get the size of each bin from the last computed field.

`GetComputedRange` Get the computed local range of the histogram from the last computed field.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.17 Lagrangian Coherent Structures

Lagrangian coherent structures (LCS) are distinct structures present in a flow filed that have a major influence over nearby trajectories over some interval of time. Some of these structures may be sources, sinks, saddles, or vortices in the flow field. Identifying Lagrangian coherent structures is part of advanced flow analysis and is an important part of studying flow fields. These structures can be studied by calculating the finite time Lyapunov exponent (FTLE) for a flow field at various locations, usually over a regular grid encompassing the entire flow field. If the provided input dataset is structured, then by default the points in this dataset will be used as seeds for advection.

The `vtkm::filter::LagrangianStructures` filter is used to compute the FTLE of a flow field. `LagrangianStructures` has the following methods.

SetStepSize/GetStepSize Set or retrieve the step size for a single advection step for the particles used to calculate the FTLE.

SetNumberOfSteps/GetNumberOfSteps Set or retrieve the maximum number of steps a particle is allowed to traverse while calculating the FTLE field.

SetAdvectionTime/GetAdvectionTime Set or retrieve the time interval of advection. The FTLE field is calculated over some finite time, and the advection time determines that interval of time used.

SetUseAuxiliaryGrid/GetUseAuxiliaryGrid Set or retrieve the flag to use auxiliary grids. When this flag is off (the default), then the points of the mesh representing the vector field are advected and used for computing the FTLE. However, if the mesh is too coarse, the FTLE will likely be inaccurate. Or if the mesh is unstructured the FTLE may be less efficient to compute. When this flag is on, an auxiliary grid of uniformly spaced points is used for the FTLE computation.

SetAuxiliaryGridDimensions/GetAuxiliaryGridDimensions Set or retrieve the dimensions of the auxiliary grid for FTLE calculation. Seeds for advection will be placed along the points of this auxiliary grid. This option has no effect unless the UseAuxiliaryGrid option is on.

SetUseFlowMapOutput/GetUseFlowMapOutput Set or retrieve the flag to use flow maps instead of advection. If the start and end points for FTLE calculation are known already, advection is an unnecessary step. This flag allows users to bypass advection, and instead use a precalculated flow map. By default this option is off.

SetFlowMapOutput/GetFlowMapOutput Set or retrieve the array representing the flow map output to be used for FTLE calculation.

SetOutputFieldName/GetOutputFieldName Set or retrieve the name of the output field in the dataset returned by the LagrangianStructures filter. By default, the field will be names "FTLE".

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.18 Mesh Quality Metrics

vtkm::filter::MeshQuality is a filter that can calculate various metrics for evaluating mesh quality. It generates a new field as output, based on the cells of an input data set. The metrics for this filter come from the Verdict library, and full mathematical descriptions for each metric can be found in the Verdict documentation.[1]

The following table lists the supported mesh quality metrics. The constructor for vtkm::filter::MeshQuality takes an argument from the enum vtkm::filter::CellMetric, which specifies the desired metric to calculate. The possible values for the CellMetric enum are listed in the following table. Also listed in the table is the default name for the field being created, which can be overridden using the SetOutputFieldName method. Also, because most metrics only work on select types of cells, the table specifies on which cell types each metric works using the abbreviations "Tri" for triangles, "Quad" for quadrilaterals, "Tet" for tetrahedrons, "Pyr" for pyramids, "Wed" for wedges, and "Hex" for hexahedrons.

[1] https://www.csimsoft.com/download?file=Documents/sand20071751.pdf

Constructor Argument	Default Output Name	Supported Cell Types	Brief Description
CellMetric::AREA	area	Tri, Quad	Area of a 2D cell
CellMetric::ASPECT_GAMMA	aspectGamma	Tet	Compare root-mean-square edge length to volume
CellMetric::ASPECT_RATIO	aspectRatio	Tri, Quad, Tet, Hex	Ratio involving longest edge and circumradius
CellMetric::CONDITION	condition	Tri, Quad, Tet, Hex	Condition number of weighted Jacobian matrix
CellMetric::DIAGONAL_RATIO	diagonalRatio	Quad, Hex	Ratio of minimum and maximum diagonals
CellMetric::DIMENSION	dimension	Hex	Designed specifically for Sandia's Pronto code
CellMetric::JACOBIAN	jacobian	Quad, Tet, Hex	Minimum determinant of Jacobian matrix, over corners and cell center
CellMetric::MAX_ANGLE	maxAngle	Tri, Quad	Maximum angle within cell, in degrees
CellMetric::MAX_DIAGONAL	maxDiagonal	Hex	Length of maximum diagonal in cell
CellMetric::MIN_ANGLE	minAngle	Tri, Quad	Minimum angle within cell, in degrees
CellMetric::MIN_DIAGONAL	minDiagonal	Hex	Length of minimum diagonal in cell
CellMetric::ODDY	oddy	Quad, Hex	Maximum deviation of a metric tensor from an identity matrix, over all corners and cell center
CellMetric::-RELATIVE_SIZE_SQUARED	relativeSize Squared	Tri, Quad, Tet, Hex	The ratio of the area/volume of this cell compared to mesh average
CellMetric::SCALED_JACOBIAN	scaledJacobian	Tri, Quad, Tet, Hex	Derived from the Jacobian metric, with normalization involving edge length

Constructor Argument	Default Output Name	Supported Cell Types	Brief Description
CellMetric::SHAPE	shape	Tri, Quad, Tet, Hex	Varies by shape type, including incorporating Jacobian or Condition Consult Verdict manual
CellMetric::SHAPE_AND_SIZE	shapeAndSize	Tri, Quad, Tet, Hex	Shape metric multiplied by relative size squared metric
CellMetric::SHEAR	shear	Quad, Hex	Min. value of Jacobian at each corner, divided by length of adjacent edges
CellMetric::SKEW	skew	Quad, Hex	Maximum angle between principle axes
CellMetric::STRETCH	stretch	Quad, Hex	Ratio of minimum edges and maximum diagonal, normalized for unit cube
CellMetric::TAPER	taper	Quad	Maximum ratio of cross-derivative with associated principal axis
CellMetric::VOLUME	volume	Tet, Pyr, Wed, Hex	Volume of a 3D cell
CellMetric::WARPAGE	warpage	Quad	Captures angles between diagonals

Finally, `MeshQuality` provides the following methods.

`SetOutputFieldName`/`GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass`/`GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.19 Point Average

`vtkm::filter::PointAverage` is the point average filter. It will take a data set with a collection of cells and a field defined on the cells of the data set and create a new field defined on the points. The values of this new derived field are computed by averaging the values of the input field at all the incident cells. This is a simple way to convert a cell field to a point field.

The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName` method.

`PointAverage` provides the following methods.

`SetActiveField`/`GetActiveFieldName` Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.20 Point Elevation

vtkm::filter::PointElevation computes the "elevation" of a field of point coordinates in space. The filter will take a data set and a field of 3 dimensional vectors and compute the distance along a line defined by a low point and a high point. Any point in the plane touching the low point and perpendicular to the line is set to the minimum range value in the elevation whereas any point in the plane touching the high point and perpendicular to the line is set to the maximum range value. All other values are interpolated linearly between these two planes. This filter is commonly used to compute the elevation of points in some direction, but can be repurposed for a variety of measures. Example 9.1 gives a demonstration of the elevation filter.

The default name for the output field is "elevation", but that can be overridden as always using the SetOutputFieldName method.

PointElevation provides the following methods.

SetLowPoint/SetHighPoint This pair of methods is used to set the low and high points, respectively, of the elevation. Each method takes three floating point numbers specifying the x, y, and z components of the low or high point.

SetRange Sets the range of values to use for the output field. This method takes two floating point numbers specifying the low and high values, respectively.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.21 Point Transform

vtkm::filter::PointTransform is the point transform filter. The filter will take a data set and a field of 3 dimensional vectors and perform the specified point transform operation. Multiple point transformations can be accomplished by subsequent calls to the filter and specifying the result of the previous transform as the input field.

The default name for the output field is “transform”, but that can be overridden as always using the SetOutputFieldName method. By default, produced field replaces the coordinate system.

PointTransform provides the following methods.

SetTranslation This method translates, or moves, each point in the input field by a given direction. This method takes either a three component vector of floats, or the x, y, z translation values separately.

SetRotation This method is used to rotate the input field about a given axis. This method takes a single floating point number to specify the degrees of rotation and either a vector representing the rotation axis, or the x, y, z axis components separately.

SetRotationX This method is used to rotate the input field about the x axis. This method takes a single floating point number to specify the degrees of rotation.

SetRotationY This method is used to rotate the input field about the y axis. This method takes a single floating point number to specify the degrees of rotation.

SetRotationZ This method is used to rotate the input field about the $z0$ axis. This method takes a single floating point number to specify the degrees of rotation.

SetScale This method is used to scale the input field. This method takes either a single float to scale each vector component of the field equally, or the x, y, z scaling values as separate floats, or a three component vector.

SetTransform This is a generic transform method. This method takes a $4x4$ matrix and applies this to the input field.

SetChangeCoordinateSystem/GetChangeCoordinateSystem When this flag is on, the default, the coordiante system in the output DataSet is replaced with the transformed point coordinates .

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.22 Stream Tracing

Stream tracing is a visualization technique used to characterize the structure of flow. The flow itself is defined by a vector field of velocities. Stream tracing works by following the path taken by the flow. There are multiple ways in which to represent flow in this manner, and consequently VTK-m contains several filters that trace streams in different ways.

Streamlines

Streamlines are a powerful technique for the visualization of flow fields. A streamline is a curve that is parallel to the velocity vector of the flow field. Individual streamlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field.

`vtkm::filter::Streamline` provides the following methods.

`SetSeeds` Specifies the seed locations for the streamlines. Each seed is advected in the vector field to generate one streamline for each seed.

`SetStepSize` Specifies the step size used for the numerical integrator (4th order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each streamline.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.5: Using `Streamline`, which is a data set with field filter.

```cpp
vtkm::filter::Streamline streamlines;

// Specify the seeds.
vtkm::cont::ArrayHandle<vtkm::Vec3f> seedArray;
seedArray.Allocate(2);
seedArray.GetPortalControl().Set(0, vtkm::Vec3f(0, 0, 0));
seedArray.GetPortalControl().Set(1, vtkm::Vec3f(1, 1, 1));

streamlines.SetActiveField("vectorvar");
streamlines.SetStepSize(0.1f);
streamlines.SetNumberOfSteps(100);
streamlines.SetSeeds(seedArray);

vtkm::cont::DataSet output = streamlines.Execute(inData);
```

Stream Surface

A *stream surface* is defined as a continous sourface that is everywhere tangent to a specified vector field. `vtkm::filter::StreamSurface` computes a stream surface from a set of input points and the vector field of the input data set. The stream surface is created by creating streamlines from each input point and then connecting adjacent streamlines with a series of triangles.

`vtkm::filter::StreamSurface` provides the following methods.

`SetSeeds` Specifies the seed locations for the edge of the stream surface.

`SetStepSize` Specifies the step size used for the numerical integrator (4th order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each seed.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.6: Using `StreamSurface`, which is a data set with field filter.

```cpp
vtkm::filter::StreamSurface streamSurface;

// Specify the seeds.
vtkm::cont::ArrayHandle<vtkm::Vec3f> seedArray;
seedArray.Allocate(2);
seedArray.GetPortalControl().Set(0, vtkm::Vec3f(0, 0, 0));
seedArray.GetPortalControl().Set(1, vtkm::Vec3f(1, 1, 1));

streamSurface.SetActiveField("vectorvar");
streamSurface.SetStepSize(0.1f);
streamSurface.SetNumberOfSteps(100);
streamSurface.SetSeeds(seedArray);

vtkm::cont::DataSet output = streamSurface.Execute(inData);
```

Pathlines

Pathlines are the analog to Streamlines for time varying vector fields. Individual pathlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field. This filter requires two data sets as input. The data set passed into the filter is termed “Previous” and the “Next” data set is specified to the filter using a method.

`vtkm::filter::Pathline` provides the following methods.

`SetPreviousTime` Specifies time value for the input data set.

`SetNextTime` Specifies time value for the next data set.

`SetNextDataSet` Specifies the data set for the next time step.

`SetSeeds` Specifies the seed locations for the pathlines. Each seed is advected in the vector field to generate one streamline for each seed.

`SetStepSize` Specifies the step size used for the numerical integrator (4th order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each pathline.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.7: Using `Pathline`, which is a data set with field filter.

```
vtkm::filter::Pathline pathlines;

// Specify the seeds.
vtkm::cont::ArrayHandle<vtkm::Vec3f> seedArray;
seedArray.Allocate(2);
seedArray.GetPortalControl().Set(0, vtkm::Vec3f(0, 0, 0));
seedArray.GetPortalControl().Set(1, vtkm::Vec3f(1, 1, 1));

pathlines.SetActiveField("vectorvar");
pathlines.SetStepSize(0.1f);
pathlines.SetNumberOfSteps(100);
pathlines.SetSeeds(seedArray);
pathlines.SetPreviousTime(0.0f);
pathlines.SetNextTime(1.0f);
pathlines.SetNextDataSet(inData2);

vtkm::cont::DataSet pathlineCurves = pathlines.Execute(inData1);
```

9.1.23 Surface Normals

`vtkm::filter::SurfaceNormals` computes the surface normals of a polygonal data set at its points and/or cells. The filter takes a data set as input and by default, uses the active coordinate system to compute the normals. Optionally, a coordinate system or a point field of 3d vectors can be explicitly provided to the `Execute` method. The point and cell normals may be oriented to a point outside of the manifold surface by setting the auto orient normals option (`SetAutoOrientNormals`), or they may point inward by also setting flip normals (`SetFlipNormals`) to true. Triangle vertices will be wound counter-clockwise around the cell normals when the consistency option (`SetConsistency`) is enabled. For non-polygonal cells, a zeroed vector is assigned. The point normals are computed by averaging the cell normals of the incident cells of each point.

The default name for the output fields is “Normals”, but that can be overridden using the `SetCellNormalsName` and `SetPointNormalsName` methods. The filter will also respect the name in `SetOutputFieldName` if neither of the others are set.

`SurfaceNormals` provides the following methods.

`SetGenerateCellNormals/GetGenerateCellNormals` Specifies whether the cell normals should be generated. This is off by default.

`SetGeneratePointNormals/GetGeneratePointNormals` Specifies whether the point normals should be generated. This is on by default.

`SetNormalizeCellNormals/GetNormalizeCellNormals` Specifies whether cell normals should be normalized (made unit length). This is on by default. The intended use case of this flag is for faster, approximate point normals generation by skipping the normalization of the face normals. Note that when set to false, the result cell normals will not be unit length normals and the point normals will be different.

`SetAutoOrientNormals/GetAutoOrientNormals` If true, any generated point and/or cell normals will be oriented to point outwards from the surface. This requires a closed manifold surface or else the behavior is undefined. This is off by default.

`SetFlipNormals/GetFlipNormals` When AutoOrientNormals is true, this option will reverse the point and cell normals to point inward. This is off by default.

`SetConsistency/GetConsistency` When GenerateCellNormals is true, this option will ensure that the triangle vertices in the output dataset are wound counter-clockwise around the generated cell normal. This only affects triangles. This is on by default.

`SetCellNormalsName/GetCellNormalsName` Specifies the output cell normals field name. If no cell or point normal name is specified, “Normals” is used.

`SetPointNormalsName/GetPointNormalsName` Specifies the output point normals field name. If no cell or point normal name is specified, “Normals” is used.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.24 Threshold

A threshold operation removes topology elements from a data set that do not meet a specified criterion. The `vtkm::filter::Threshold` filter removes all cells where the field (provided to `Execute`) is not between a range of values.

Note that `Threshold` either passes an entire cell or discards an entire cell. This can consequently lead to jagged surfaces at the interface of the threshold caused by the shape of cells that jut inside or outside the removed region. See Section 9.1.3 for a clipping filter that will clip off a smooth region of the mesh.

`Threshold` provides the following methods.

`SetLowerThreshold/GetLowerThreshold` Specifies the lower scalar value. Any cells where the scalar field is less than this value are removed.

`SetUpperThresholdGetUpperThreshold` Specifies the upper scalar value. Any cells where the scalar field is more than this value are removed.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.25 Tube

`vtkm::filter::Tube` generates a tube around each line and polyline in the input data set. The radius, number of sides, and end capping can be specified for each tube. The orientation of the geometry of the tube are computed automatically using a heuristic to minimize the twisting along the input data set.

`Tube` provides the following methods.

`SetRadius` Specifies the radius of the tube.

`SetNumberOfSides` Specifies the number of sides for the tube geometry.

`SetCapping` Specifies if the ends of the tube should be capped.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.8: Using Tube, which is a data set with field filter.

```
vtkm::filter::Tube tubeFilter;

tubeFilter.SetRadius(0.5f);
tubeFilter.SetNumberOfSides(7);
tubeFilter.SetCapping(true);

vtkm::cont::DataSet output = tubeFilter.Execute(inData);
```

9.1.26 Vector Magnitude

vtkm::filter::VectorMagnitude takes a field comprising vectors and computes the magnitude for each vector. The vector field is selected as usual with the SetActiveField method. The default name for the output field is "magnitude", but that can be overridden as always using the SetOutputFieldName method.

VectorMagnitude provides the following methods.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.27 Vertex Clustering

vtkm::filter::VertexClustering is a filter that simplifies a polygonal mesh. It does so by dividing space into a uniform grid of bin and then merges together all points located in the same bin. The smaller the dimensions of this binning grid, the fewer polygons will be in the output cells and the coarser the representation. This surface simplification is an important operation to support level of detail (LOD) rendering in visualization applications.

VertexClustering provides the following methods.

SetNumberOfDivisions/GetNumberOfDimensions Specifies the dimensions of the uniform grid that establishes the bins used for clustering. Setting smaller numbers of dimensions produces a smaller output, but with a coarser representation of the surface. The dimensions are provided as a vtkm::Id3.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.9: Using `VertexClustering`.

```
vtkm::filter::VertexClustering vertexClustering;

vertexClustering.SetNumberOfDivisions(vtkm::Id3(128, 128, 128));

vtkm::cont::DataSet simplifiedSurface = vertexClustering.Execute(originalSurface);
```

9.1.28 Warp Scalar

`vtkm::filter::WarpScalar` is a specialized point transformation filter. The filter transforms points by moving them based on a scalar field and a constant scale factor. This filter is useful for creating carpet plots.

The `WarpScalar` filter will take a data set, a normal field, a scalar field, and a constant scale factor. The coordinates will be scaled based on the scalar field and the scale factor. If no explicit normal field is provided the filter will search for a field named “normal”. If no explicit scalar field is provided the filter will search for a field named “scalarfactor”.

The default name for the output field is “warpscalar”, but that can be overridden as always using the `SetOutputFieldName` method.

In addition to the standard `SetOutputFieldName` and `Execute` methods, `WarpScalar` provides the following methods.

`SetNormalField` This method allows the user to select the name of the normal field. The normal field is the B field in the warp equation of $A+B \times scaleAmount \times scalarFactor$ (where A is the original position of the point).

`SetScalarFactorField` This method allows the user to select the name of the scale factor field. The scale factor field is the $scalarFactor$ field in the warp equation of $A+B \times scaleAmount \times scalarFactor$ (where A is the original position of the point).

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.29 Warp Vector

vtkm::filter::WarpVector is a specialized point transformation filter. The filter transforms points by moving them based on a vector field and a constant scale factor. This filter can be used to highlight interesting features such as flow or deformations.

The WarpScalar filter will take a data set, a vector field, and a constant scale factor. The coordinates will be scaled based on the vector field and the scale factor. If no explicit vector field is provided the filter will search for a field named "normal".

The default name for the output field is "warpvector", but that can be overridden as always using the SetOutputFieldName method.

In addition the standard SetOutputFieldName and Execute methods, WarpVector provides the following methods.

SetVectorField This method allows the user to select the name of the vector field. The vector field is the B field in the warp equation of $A+B$ (where A is the original position of the point).

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.30 ZFP Compression

vtkm::filter::ZFPCompressor takes a 1D, 2D, or 3D field and compresses the values using the compression algorithm ZFP. The field is selected as usual with the SetActiveField method. The rate of compression is set using SetRate. The default name for the output field is "compressed"

ZFPCompressor provides the following methods:

SetRate/GetRate Specifies the rate of compression.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

vtkm::filter::ZFPDecompressor takes a field of compressed values and decompresses into scalar values using the compression algorithm ZFP. The field is selected as usual with the SetActiveField method. The rate of compression is set using SetRate. The default name for the output field is “decompressed”

ZFPDecompressor provides the following methods:

SetRate Specifies the rate of compression.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetActiveField/GetActiveFieldName Specifies the name of the field to use as input.

SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.2 Advanced Field Management

Most filters work with fields as inputs and outputs to their algorithms. Although in the previous discussions of the filters we have seen examples of specifying fields, these examples have been kept brief in the interest of clarity. In this section we revisit how filters manage fields and provide more detailed documentation of the controls.

Note that not all of the discussion in this section applies to all the aforementioned filters. For example, not all filters have a specified input field. But where possible, the interface to the filter objects is kept consistent.

9.2.1 Input Fields

Many of VTK-m’s filters have a method named SetActiveField, which selects a field in the input data to use as the data for the filter’s algorithm. We have already seen how SetActiveField takes the name of the field as an argument. However, SetActiveField also takes an optional second argument that specifies which topological elements the field is associated with (such as points or cells). If specified, this argument is one of the following.

vtkm::cont::Field::ASSOC_ANY Any field regardless of the association. (This is the default if no association is given.)

vtkm::cont::Field::ASSOC_POINTS A field that applies to points. There is a separate field value attached to each point. Point fields usually represent samples of continuous data that can be reinterpolated through cells. Physical properties such as temperature, pressure, density, velocity, etc. are usually best represented in point fields. Data that deals with the points of the topology, such as displacement vectors, are also appropriate for point data.

vtkm::cont::Field::ASSOC_CELL_SET A field that applies to cells. There is a separate field value attached to each cell in a cell set. Cell fields usually represent values from an integration over the finite cells of the mesh. Integrated values like mass or volume are best represented in cell fields. Statistics about each cell like strain or cell quality are also appropriate for cell data.

vtkm::cont::Field::ASSOC_WHOLE_MESH A "global" field that applies to the whole mesh. These often contain summary or annotation information. An example of a whole mesh field could be the volume that the mesh covers.

Example 9.10: Setting a field's active filter with an association.

```
  filter.SetActiveField("pointvar", vtkm::cont::Field::Association::POINTS);
```

Common Errors

It is possible to have two fields with the same name that are only differentiatable by the association. That is, you could have a point field and a cell field with different data but the same name. Thus, it is best practice to specify the field association when possible. Likewise, it is poor practice to have two fields with the same name, particularly if the data are not equivalent in some way. It is often the case that fields are selected without an association.

It is also possible to set the active scalar field as a coordinate system of the data. A coordinate system essentially provides the spatial location of the points of the data and they have a special place in the vtkm::cont::DataSet structure. (See Section 7.4 for details on coordinate systems.) You can use a coordinate system as the active scalars by calling the SetUseCoordinateSystemAsField method with a true flag. Since a DataSet can have multiple coordinate systems, you can select the desired coordinate system with SetActiveCoordinateSystem. (By default, the first coordinate system will be used.)

9.2.2 Passing Fields from Input to Output

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. By default, the filter will automatically pass all fields from input to output (performing whatever transformations are necessary). You can control which fields are passed (and equivalently which are not) with the SetFieldsToPass methods of vtkm::filter::Filter.

There are multiple ways to to use Filter::SetFieldsToPass to control what fields are passed. If you want to turn off all fields so that none are passed, call SetFieldsToPass with vtkm::filter::FieldSelection::MODE_NONE.

Example 9.11: Turning off the passing of all fields when executing a filter.

```
    filter.SetFieldsToPass(vtkm::filter::FieldSelection::MODE_NONE);
```

If you want to pass one specific field, you can pass that field's name to SetFieldsToPass.

Example 9.12: Setting one field to pass by name.

```
filter.SetFieldsToPass("pointvar");
```

Or you can provide a list of fields to pass by giving SetFieldsToPass an initializer list of names.

Example 9.13: Using a list of fields for a filter to pass.

```
filter.SetFieldsToPass({ "pointvar", "cellvar" });
```

If you want to instead select a list of fields to *not* pass, you can add vtkm::filter::FieldSelection::MODE_EXCLUDE as an argument to SetFieldsToPass.

Example 9.14: Excluding a list of fields for a filter to pass.

```
filter.SetFieldsToPass({ "pointvar", "cellvar" },
                       vtkm::filter::FieldSelection::MODE_EXCLUDE);
```

Ultimately, Filter::SetFieldsToPass takes a vtkm::filter::FieldSelection object. You can create one directly to select (or exclude) specific fields and their associations.

Example 9.15: Using vtkm::filter::FieldSelection.

```
vtkm::filter::FieldSelection fieldSelection;
fieldSelection.AddField("scalars");
fieldSelection.AddField("cellvar", vtkm::cont::Field::Association::CELL_SET);

filter.SetFieldsToPass(fieldSelection);
```

It is also possible to specify field attributions directly to Filter::SetFieldsToPass. If you only have one field, you can just specify both the name and attribution. If you have multiple fields, you can provide an initializer list of std::pair or vtkm::Pair containing a std::string and a vtkm::cont::Field::AssociationEnum. In either case, you can add an optional last argument of vtkm::filter::FieldSelection::MODE_EXCLUDE to exclude the specified filters instead of selecting them.

Example 9.16: Selecting one field and its association for a filter to pass.

```
filter.SetFieldsToPass("pointvar", vtkm::cont::Field::Association::POINTS);
```

Example 9.17: Selecting a list of fields and their associations for a filter to pass.

```
filter.SetFieldsToPass(
  { vtkm::make_Pair("pointvar", vtkm::cont::Field::Association::POINTS),
    vtkm::make_Pair("cellvar", vtkm::cont::Field::Association::CELL_SET),
    vtkm::make_Pair("scalars", vtkm::cont::Field::Association::ANY) });
```

CHAPTER

TEN

RENDERING

Rendering, the generation of images from data, is a key component to visualization. To assist with rendering, VTK-m provides a rendering package to produce imagery from data, which is located in the vtkm::rendering namespace.

The rendering package in VTK-m is not intended to be a fully featured rendering system or library. Rather, it is a lightweight rendering package with two primary use cases:

1. New users getting started with VTK-m need a "quick and dirty" render method to see their visualization results.
2. In situ visualization that integrates VTK-m with a simulation or other data-generation system might need a lightweight rendering method.

Both of these use cases require just a basic rendering platform. Because VTK-m is designed to be integrated into larger systems, it does not aspire to have a fully featured rendering system.

Did you know?

VTK-m's big sister toolkit VTK is already integrated with VTK-m and has its own fully featured rendering system. If you need more rendering capabilities than what VTK-m provides, you can leverage VTK instead.

10.1 Scenes and Actors

The primary intent of the rendering package in VTK-m is to visually display the data that is loaded and processed. Data are represented in VTK-m by vtkm::cont::DataSet objects, which are described in Chapter 7. They are also the unit created from I/O operations (Chapter 8 and filters (Chapter 9).

To render a DataSet, the data are wrapped in a vtkm::rendering::Actor class. The Actor holds the components of the DataSet to render (a cell set, a coordinate system, and a field). A color table can also be optionally be specified, but a default color table will be specified otherwise.

Actors are collected together in an object called vtkm::rendering::Scene. An Actor is added to a Scene with the AddActor method. The following example demonstrates creating a Scene with one Actor.

Example 10.1: Creating an Actor and adding it to a Scene.

```
vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                             surfaceData.GetCoordinateSystem(),
                             surfaceData.GetField("RandomPointScalars"));

vtkm::rendering::Scene scene;
scene.AddActor(actor);
```

10.2 Canvas

A *canvas* is a unit that represents the image space that is the target of the rendering. The canvas' primary function is to manage the buffers that hold the working image data during the rendering. The canvas also manages the context and state of the rendering subsystem.

vtkm::rendering::Canvas is the base class of all canvas objects. Each type of rendering system has its own canvas subclass, but currently the only rendering system provided by VTK-m is the internal ray tracer. The canvas for the ray tracer is vtkm::rendering::CanvasRayTracer. CanvasRayTracer is typically constructed by giving the width and height of the image to render.

Example 10.2: Creating a canvas for rendering.

```
vtkm::rendering::CanvasRayTracer canvas(1920, 1080);
```

10.3 Mappers

A *mapper* is a unit that converts data (managed by an Actor) and issues commands to the rendering subsystem to generate images. All mappers in VTK-m are a subclass of vtkm::rendering::Mapper. Different rendering systems (as established by the Canvas) often require different mappers. Also, different mappers could render different types of data in different ways. For example, one mapper might render polygonal surfaces whereas another might render polyhedra as a translucent volume. Thus, a mapper should be picked to match both the rendering system of the Canvas and the data in the Actor.

The following mappers are provided by VTK-m.

vtkm::rendering::MapperRayTracer Uses VTK-m's built in ray tracing system to render the visible surface of a mesh. MapperRayTracer only works in conjunction with CanvasRayTracer.

vtkm::rendering::MapperCylinder Uses VTK-m's built in ray tracing system to render cylinders as lines of a mesh. MapperCylinder only works in conjunction with CanvasRayTracer.

vtkm::rendering::MapperPoint Uses VTK-m's built in ray tracing system to render the visible points/vertices of a mesh. MapperPoint only works in conjunction with CanvasRayTracer.

vtkm::rendering::MapperQuad Uses VTK-m's built in ray tracing system to render the visible quadrilaterals of a mesh. MapperQuad only works in conjunction with CanvasRayTracer.

vtkm::rendering::MapperVolume Uses VTK-m's built in ray tracing system to render polyhedra as a translucent volume. MapperVolume only works in conjunction with CanvasRayTracer.

vtkm::rendering::MapperWireframer Uses VTK-m's built in ray tracing system to render the cell edges (i.e. the "wireframe") of a mesh. MapperWireframer only works in conjunction with CanvasRayTracer.

10.4 Views

A *view* is a unit that collects all the structures needed to perform rendering. It contains everything needed to take a `Scene` (Section 10.1) and use a `Mapper` (Section 10.3) to render it onto a `Canvas` (Section 10.2). The view also annotates the image with spatial and scalar properties.

The base class for all views is `vtkm::rendering::View`. `View` is an abstract class, and you must choose one of the three provided subclasses, `vtkm::rendering::View3D`, `vtkm::rendering::View2D`, and `vtkm::rendering::View3D`, depending on the type of data being presented. All three view classes take a `Scene`, a `Mapper`, and a `Canvas` as arguments to their constructor.

Example 10.3: Constructing a `View`.

```
vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                             surfaceData.GetCoordinateSystem(),
                             surfaceData.GetField("RandomPointScalars"));

vtkm::rendering::Scene scene;
scene.AddActor(actor);

vtkm::rendering::MapperRayTracer mapper;
vtkm::rendering::CanvasRayTracer canvas(1920, 1080);

vtkm::rendering::View3D view(scene, mapper, canvas);
view.Initialize();
```

Once the `View` is created but before it is used to render, the `Initialize` method should be called. This is demonstrated in Example 10.3.

The `View` also maintains a *background color* (the color used in areas where nothing is drawn) and a *foreground color* (the color used for annotation elements). By default, the `View` has a black background and a white foreground. These can be set in the view's constructor, but it is a bit more readable to set them using the `View::SetBackground` and `View::SetForeground` methods. In either case, the colors are specified using the `vtkm::rendering::Color` helper class, which manages the red, green, and blue color channels as well as an optional alpha channel. These channel values are given as floating point values between 0 and 1.

Example 10.4: Changing the background and foreground colors of a `View`.

```
view.SetBackgroundColor(vtkm::rendering::Color(1.0f, 1.0f, 1.0f));
view.SetForegroundColor(vtkm::rendering::Color(0.0f, 0.0f, 0.0f));
```

Common Errors

Although the background and foreground colors are set independently, it will be difficult or impossible to see the annotation if there is not enough contrast between the background and foreground colors. Thus, when changing a `View`'s background color, it is always good practice to also change the foreground color.

Once the `View` is constructed, intialized, and set up, it is ready to render. This is done by calling the `View::Paint` method.

Example 10.5: Using `Canvas::Paint` in a display callback.

```
view.Paint();
```

Putting together Examples 10.3, 10.4, and 10.5, the final render of a view looks like that in Figure 10.1.

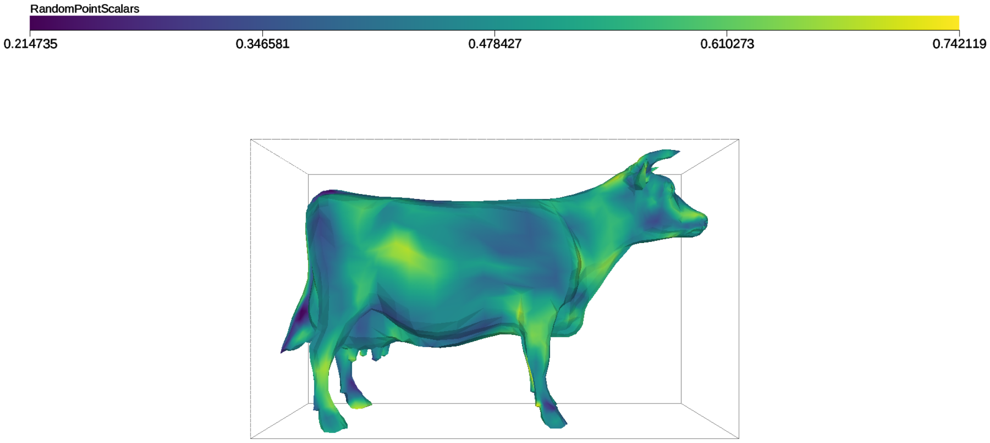


Figure 10.1: Example output of VTK-m's rendering system.

Of course, the vtkm::rendering::CanvasRayTracer created in 10.3 is an offscreen rendering buffer, so you cannot immediately see the image. When doing batch visualization, an easy way to output the image to a file for later viewing is with the View::SaveAs method. This method saves the file in the portable pixelmap (PPM) format.

Example 10.6: Saving the result of a render as an image file.

```
view.SaveAs("BasicRendering.ppm");
```

We visit doing interactive rendering in a GUI later in Section 10.7.

10.5 Changing Rendering Modes

Example 10.3 constructs the default mapper for ray tracing, which renders the data as an opaque solid. However, you can change the rendering mode by using one of the other mappers listed in Section 10.3. For example, say you just wanted to see a wireframe representation of your data. You can achieve this by using vtkm::rendering::MapperWireframer.

Example 10.7: Creating a mapper for a wireframe representation.

```
vtkm::rendering::MapperWireframer mapper;
vtkm::rendering::View3D view(scene, mapper, canvas);
```

Alternatively, perhaps you wish to render just the points of mesh. vtkm::rendering::MapperPoint renders the points as spheres and also optionally can scale the spheres based on field values.

Example 10.8: Creating a mapper for point representation.

```
vtkm::rendering::MapperPoint mapper;
mapper.UseVariableRadius(true);
mapper.SetRadiusDelta(10.0f);

vtkm::rendering::View3D view(scene, mapper, canvas);
```

These mappers respectively render the images shown in Figure 10.2. Other mappers, such as those that can render translucent volumes, are also available.

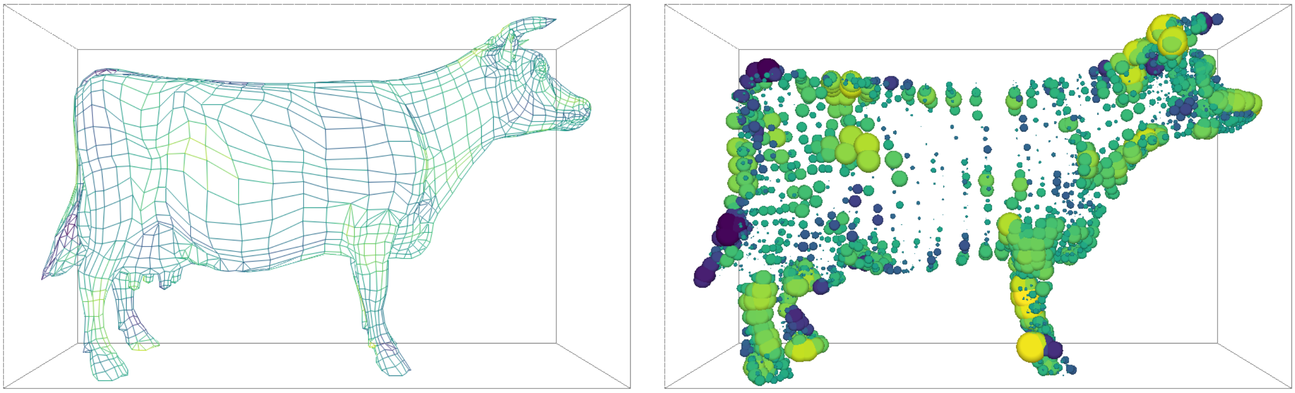


Figure 10.2: Examples of alternate rendering modes using different mappers. The left image is rendered with MapperWireframer. The right image is rendered with MapperPoint.

10.6 Manipulating the Camera

The vtkm::rendering::View uses an object called vtkm::rendering::Camera to describe the vantage point from which to draw the geometry. The camera can be retrieved from the View::GetCamera method. That retrieved camera can be directly manipulated or a new camera can be provided by calling View::SetCamera. In this section we discuss camera setups typical during view set up. Camera movement during interactive rendering is revisited in Section 10.7.2.

A Camera operates in one of two major modes: 2D mode or 3D mode. 2D mode is designed for looking at flat geometry (or close to flat geometry) that is parallel to the x-y plane. 3D mode provides the freedom to place the camera anywhere in 3D space. The different modes can be set with SetModeTo2D and SetModeTo3D, respectively. The interaction with the camera in these two modes is very different.

10.6.1 2D Camera Mode

The 2D camera is restricted to looking at some region of the x-y plane.

View Range

The vantage point of a 2D camera can be specified by simply giving the region in the x-y plane to look at. This region is specified by calling Camera::SetViewRange2D. This method takes the left, right, bottom, and top of the region to view. Typically these are set to the range of the geometry in world space as shown in Figure 10.3.

There are 3 overloaded versions of the SetViewRange2D method. The first version takes the 4 range values, left, right, bottom, and top, as separate arguments in that order. The second version takes two vtkm::Range objects specifying the range in the x and y directions, respectively. The third version trakes a single vtkm::Bounds object, which completely specifies the spatial range. (The range in z is ignored.) The Range and Bounds objects are documented later in Sections 19.4 and 19.5, respectively.

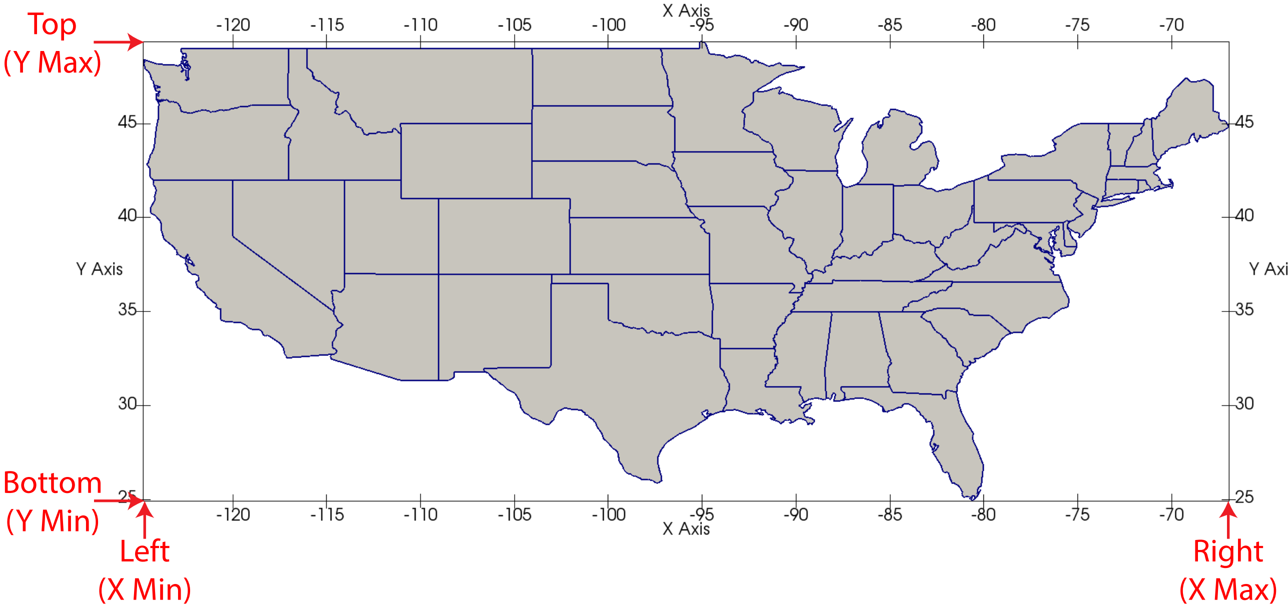


Figure 10.3: The view range bounds to give a Camera.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the Camera::Pan method. Pan takes two arguments: the amount to pan in x and the amount to pan in y.

The pan is given with respect to the projected space. So a pan of 1 in the x direction moves the camera to focus on the right edge of the image whereas a pan of -1 in the x direction moves the camera to focus on the left edge of the image.

Example 10.9: Panning the camera.

```
view.GetCamera().Pan(deltaX, deltaY);
```

Zoom

A camera zoom draws the geometry larger or smaller. A camera zoom is performed by calling the Camera::Zoom method. Zoom takes a single argument specifying the zoom factor. A positive number draws the geometry larger (zoom in), and larger zoom factor results in larger geometry. Likewise, a negative number draws the geometry smaller (zoom out). A zoom factor of 0 has no effect.

Example 10.10: Zooming the camera.

```
view.GetCamera().Zoom(zoomFactor);
```

10.6.2 3D Camera Mode

The 3D camera is a free-form camera that can be placed anywhere in 3D space and can look in any direction. The projection of the 3D camera is based on the pinhole camera model in which all viewing rays intersect a single point. This single point is the camera's position.

Position and Orientation

The position of the camera, which is the point where the observer is viewing the scene, can be set with the Camera::SetPosition method. The direction the camera is facing is specified by giving a position to focus on. This is called either the "look at" point or the focal point and is specified with the Camera::SetLookAt method. Figure 10.4 shows the relationship between the position and look at points.

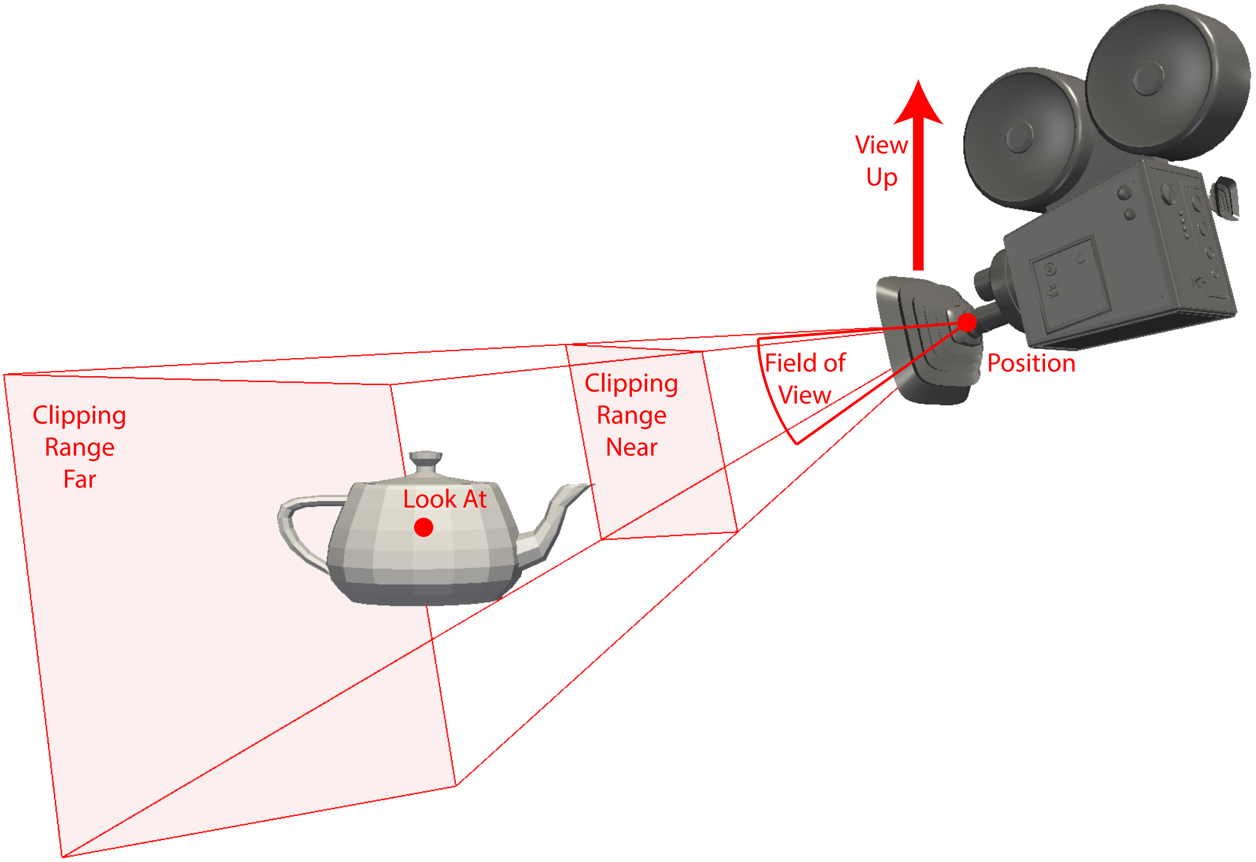


Figure 10.4: The position and orientation parameters for a Camera.

In addition to specifying the direction to point the camera, the camera must also know which direction is considered "up." This is specified with the view up vector using the Camera::SetViewUp method. The view up vector points from the camera position (in the center of the image) to the top of the image. The view up vector in relation to the camera position and orientation is shown in Figure 10.4.

Another important parameter for the camera is its field of view. The field of view specifies how wide of a region the camera can see. It is specified by giving the angle in degrees of the cone of visible region emanating from the pinhole of the camera to the Camera::SetFieldOfView method. The field of view angle in relation to the camera orientation is shown in Figure 10.4. A field of view angle of 60° usually works well.

Finally, the camera must specify a clipping region that defines the valid range of depths for the object. This is a pair of planes parallel to the image that all visible data must lie in. Each of these planes is defined simply by their distance to the camera position. The near clip plane is closer to the camera and must be in front of all geometry. The far clip plane is further from the camera and must be behind all geometry. The distance to both the near and far planes are specified with the `Camera::SetClippingRange` method. Figure 10.4 shows the clipping planes in relationship to the camera position and orientation.

Example 10.11: Directly setting `vtkm::rendering::Camera` position and orientation.

```
camera.SetPosition(vtkm::make_Vec(10.0, 6.0, 6.0));
camera.SetLookAt(vtkm::make_Vec(0.0, 0.0, 0.0));
camera.SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
camera.SetFieldOfView(60.0);
camera.SetClippingRange(0.1, 100.0);
```

Movement

In addition to specifically setting the position and orientation of the camera, `vtkm::rendering::Camera` contains several convenience methods that move the camera relative to its position and look at point.

Two such methods are elevation and azimuth, which move the camera around the sphere centered at the look at point. `Camera::Elevation` raises or lowers the camera. Positive values raise the camera up (in the direction of the view up vector) whereas negative values lower the camera down. `Camera::Azimuth` moves the camera around the look at point to the left or right. Positive values move the camera to the right whereas negative values move the camera to the left. Both `Elevation` and `Azimuth` specify the amount of rotation in terms of degrees. Figure 10.5 shows the relative movements of `Elevation` and `Azimuth`.

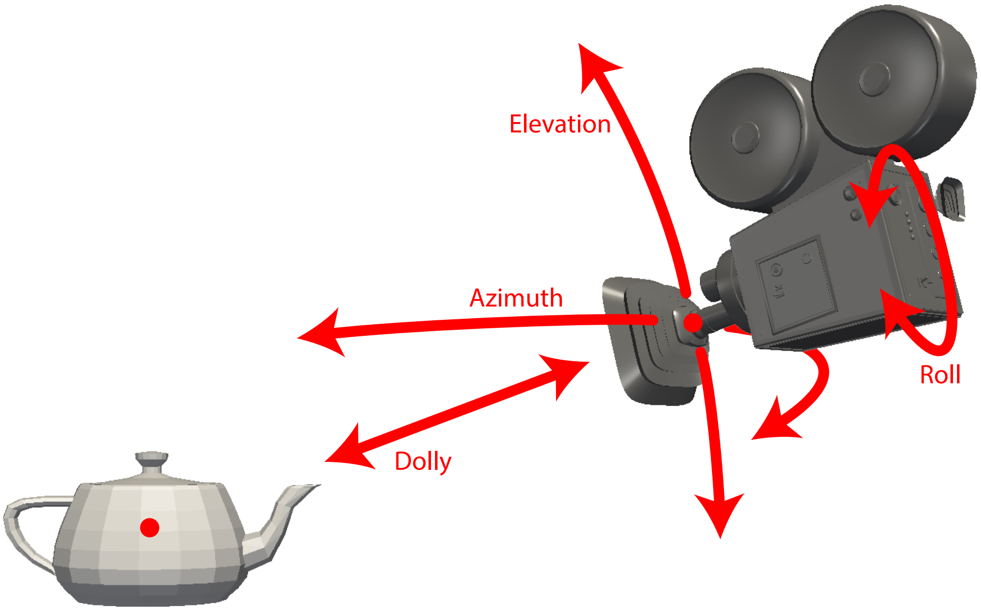


Figure 10.5: `Camera` movement functions relative to position and orientation.

Example 10.12: Moving the camera around the look at point.

```
view.GetCamera().Azimuth(45.0);
view.GetCamera().Elevation(45.0);
```

Common Errors

The Camera::Elevation and Camera::Azimuth methods change the position of the camera, but not the view up vector. This can cause some wild camera orientation changes when the direction of the camera view is near parallel to the view up vector, which often happens when the elevation is raised or lowered by about 90 degrees.

In addition to rotating the camera around the look at point, you can move the camera closer or further from the look at point. This is done with the Camera::Dolly method. The Dolly method takes a single value that is the factor to scale the distance between camera and look at point. Values greater than one move the camera away, values less than one move the camera closer. The direction of dolly movement is shown in Figure 10.5.

Finally, the Camera::Roll method rotates the camera around the viewing direction. It has the effect of rotating the rendered image. The Roll method takes a single value that is the angle to rotate in degrees. The direction of roll movement is shown in Figure 10.5.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the Camera::Pan method. Pan takes two arguments: the amount to pan in x and the amount to pan in y.

The pan is given with respect to the projected space. So a pan of 1 in the x direction moves the camera to focus on the right edge of the image whereas a pan of −1 in the x direction moves the camera to focus on the left edge of the image.

Example 10.13: Panning the camera.

```
view.GetCamera().Pan(deltaX, deltaY);
```

Zoom

A camera zoom draws the geometry larger or smaller. A camera zoom is performed by calling the Camera::Zoom method. Zoom takes a single argument specifying the zoom factor. A positive number draws the geometry larger (zoom in), and larger zoom factor results in larger geometry. Likewise, a negative number draws the geometry smaller (zoom out). A zoom factor of 0 has no effect.

Example 10.14: Zooming the camera.

```
view.GetCamera().Zoom(zoomFactor);
```

Reset

Setting a specific camera position and orientation can be frustrating, particularly when the size, shape, and location of the geometry is not known a priori. Typically this involves querying the data and finding a good camera orientation.

To make this process simpler, vtkm::rendering::Camera has a convenience method named Camera::ResetToBounds that automatically positions the camera based on the spatial bounds of the geometry. The most expedient method to find the spatial bounds of the geometry being rendered is to get the vtkm::rendering::Scene object and call GetSpatialBounds. The Scene object can be retrieved from the vtkm::rendering::View, which, as described in Section 10.4, is the central object for managing rendering.

Example 10.15: Resetting a Camera to view geometry.

```
void ResetCamera(vtkm::rendering::View& view)
{
  vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
  view.GetCamera().ResetToBounds(bounds);
}
```

The ResetToBounds method operates by placing the look at point in the center of the bounds and then placing the position of the camera relative to that look at point. The position is such that the view direction is the same as before the call to ResetToBounds and the distance between the camera position and look at point has the bounds roughly fill the rendered image. This behavior is a convenient way to update the camera to make the geometry most visible while still preserving the viewing position. If you want to reset the camera to a new viewing angle, it is best to set the camera to be pointing in the right direction and then calling ResetToBounds to adjust the position.

Example 10.16: Resetting a Camera to be axis aligned.

```
  view.GetCamera().SetPosition(vtkm::make_Vec(0.0, 0.0, 0.0));
  view.GetCamera().SetLookAt(vtkm::make_Vec(0.0, 0.0, -1.0));
  view.GetCamera().SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
  vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
  view.GetCamera().ResetToBounds(bounds);
```

10.7 Interactive Rendering

So far in our description of VTK-m's rendering capabilities we have talked about doing rendering of fixed scenes. However, an important use case of scientific visualization is to provide an interactive rendering system to explore data. In this case, you want to render into a GUI application that lets the user interact manipulate the view. The full design of a 3D visualization application is well outside the scope of this book, but we discuss in general terms what you need to plug VTK-m's rendering into such a system.

In this section we discuss two important concepts regarding interactive rendering. First, we need to write images into a GUI while they are being rendered. Second, we want to translate user interaction to camera movement.

10.7.1 Rendering Into a GUI

Before being able to show rendering to a user, we need a system rendering context in which to push the images. In this section we demonstrate the display of images using the OpenGL rendering system, which is common for scientific visualization applications. That said, you could also use other rendering systems like DirectX or even paste images into a blank widget.

Creating an OpenGL context varies depending on the OS platform you are using. If you do not already have an application you want to integrate with VTK-m's rendering, you may wish to start with graphics utility API such as GLUT or GLFW. The process of initializing an OpenGL context is not discussed here.

The process of rendering into an OpenGL context is straightforward. First call Paint on the View object to do the actual rendering. Second, get the image color data out of the View's Canvas object. This is done by calling Canvas::GetColorBuffer. This will return a vtkm::cont::ArrayHandle object containing the image's pixel color data. (ArrayHandles are discussed in detail in Chapter 16 and subsequent chapters.) A raw pointer can be pulled out of this ArrayHandle by calling GetStorage().GetBasePointer(). Third, the pixel color data are pasted into the OpenGL render context. There are multiple ways to do so, but the most straightforward way is to use the glDrawPixels function provided by OpenGL. Fourth, swap the OpenGL buffers. The method to swap OpenGL buffers varies by OS platform. The aforementioned graphics libraries GLUT and GLFW each provide a function for doing so.

Example 10.17: Rendering a `View` and pasting the result to an active OpenGL context.

```
view.Paint();

// Get the color buffer containing the rendered image.
vtkm::cont::ArrayHandle<vtkm::Vec4f_32> colorBuffer =
  view.GetCanvas().GetColorBuffer();

// Pull the C array out of the arrayhandle.
void* colorArray = colorBuffer.GetStorage().GetBasePointer();

// Write the C array to an OpenGL buffer.
glDrawPixels((GLint)view.GetCanvas().GetWidth(),
             (GLint)view.GetCanvas().GetHeight(),
             GL_RGBA,
             GL_FLOAT,
             colorArray);

// Swap the OpenGL buffers (system dependent).
```

10.7.2 Camera Movement

When interactively manipulating the camera in a windowing system, the camera is usually moved in response to mouse movements. Typically, mouse movements are detected through callbacks from the windowing system back to your application. Once again, the details on how this works depend on your windowing system. The assumption made in this section is that through the windowing system you will be able to track the x-y pixel location of the mouse cursor at the beginning of the movement and the end of the movement. Using these two pixel coordinates, as well as the current width and height of the render space, we can make several typical camera movements.

Common Errors

Pixel coordinates in VTK-m's rendering system originate in the lower-left corner of the image. However, windowing systems generally report mouse coordinates with the origin in the upper*-left corner. The upshot is that the y coordinates will have to be reversed when translating mouse coordinates to VTK-m image coordinates. This inverting is present in all the following examples.*

Rotate

A common and important mode of interaction with 3D views is to allow the user to rotate the object under inspection by dragging the mouse. To facilitate this type of interactive rotation, `vtkm::rendering::Camera` provides a convenience method named `TrackballRotate`. The `TrackballRotate` method takes a start and end position of the mouse on the image and rotates viewpoint as if the user grabbed a point on a sphere centered in the image at the start position and moved under the end position.

The `TrackballRotate` method is typically called from within a mouse movement callback. The callback must record the pixel position from the last event and the new pixel position of the mouse. Those pixel positions must be normalized to the range -1 to 1 where the position (-1,-1) refers to the lower left of the image and (1,1) refers to the upper right of the image. The following example demonstrates the typical operations used to establish rotations when dragging the mouse.

Example 10.18: Interactive rotations through mouse dragging with Camera::TrackballRotate.

```
void DoMouseRotate(vtkm::rendering::View& view,
                   vtkm::Id mouseStartX,
                   vtkm::Id mouseStartY,
                   vtkm::Id mouseEndX,
                   vtkm::Id mouseEndY)
{
  vtkm::Id screenWidth = view.GetCanvas().GetWidth();
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to
  // width/height, to normalized screen coordinates from -1 to 1. Note that y
  // screen coordinates are usually given from the top down whereas our
  // geometry transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
  vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  view.GetCamera().TrackballRotate(startX, startY, endX, endY);
}
```

Pan

Panning can be performed by calling Camera::Pan with the translation relative to the width and height of the canvas. For the translation to track the movement of the mouse cursor, simply scale the pixels the mouse has traveled by the width and height of the image.

Example 10.19: Pan the view based on mouse movements.

```
void DoMousePan(vtkm::rendering::View& view,
                vtkm::Id mouseStartX,
                vtkm::Id mouseStartY,
                vtkm::Id mouseEndX,
                vtkm::Id mouseEndY)
{
  vtkm::Id screenWidth = view.GetCanvas().GetWidth();
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to
  // width/height, to normalized screen coordinates from -1 to 1. Note that y
  // screen coordinates are usually given from the top down whereas our
  // geometry transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
  vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  vtkm::Float32 deltaX = endX - startX;
  vtkm::Float32 deltaY = endY - startY;

  view.GetCamera().Pan(deltaX, deltaY);
}
```

Zoom

Zooming can be performed by calling Camera::Zoom with a positive or negative zoom factor. When using Zoom to respond to mouse movements, a natural zoom will divide the distance traveled by the mouse pointer by the width or height of the screen as demonstrated in the following example.

Example 10.20: Zoom the view based on mouse movements.

```
void DoMouseZoom(vtkm::rendering::View& view,
                 vtkm::Id mouseStartY,
                 vtkm::Id mouseEndY)
{
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to height,
  // to normalized screen coordinates from -1 to 1. Note that y screen
  // coordinates are usually given from the top down whereas our geometry
  // transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  vtkm::Float32 zoomFactor = endY - startY;

  view.GetCamera().Zoom(zoomFactor);
}
```

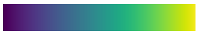
10.8 Color Tables

An important feature of VTK-m's rendering units is the ability to pseudocolor objects based on scalar data. This technique maps each scalar to a potentially unique color. This mapping from scalars to colors is defined by a vtkm::cont::ColorTable object. A ColorTable can be specified as an optional argument when constructing a vtkm::rendering::Actor. (Use of Actors is discussed in Section 10.1.)

Example 10.21: Specifying a ColorTable for an Actor.

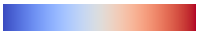
```
  vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                               surfaceData.GetCoordinateSystem(),
                               surfaceData.GetField("RandomPointScalars"),
                               vtkm::cont::ColorTable("inferno"));
```

The easiest way to create a ColorTable is to provide the name of one of the many predefined sets of color provided by VTK-m. A list of all available predefined color tables is provided below.



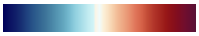
Viridis

Matplotlib Virdis, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white. This is the default color map.



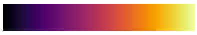
Cool to Warm

A color table designed to be perceptually even, to work well on shaded 3D surfaces, and to generally perform well across many uses.



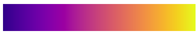
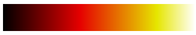
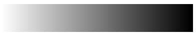
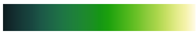
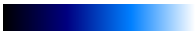
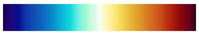
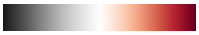
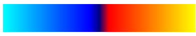
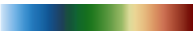
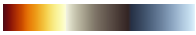
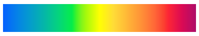
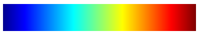
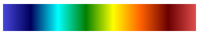
Cool to Warm Extended

This colormap is an expansion on cool to warm that moves through a wider range of hue and saturation. Useful if you are looking for a greater level of detail, but the darker colors at the end might interfere with 3D surfaces.



Inferno

Matplotlib Inferno, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.

	Plasma	Matplotlib Plasma, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.
	Black-Body Radiation	The colors are inspired by the wavelengths of light from black body radiation. The actual colors used are designed to be perceptually uniform.
	X Ray	Greyscale colormap useful for making volume renderings similar to what you would expect in an x-ray.
	Green	A sequential color map of green varied by saturation.
	Black - Blue - White	A sequential color map from black to blue to white.
	Blue to Orange	A double-ended (diverging) color table that goes from dark blues to a neutral white and then a dark orange at the other end.
	Gray to Red	A double-ended (diverging) color table with black/gray at the low end and orange/red at the high end.
	Cold and Hot	A double-ended color map with a black middle color and diverging values to either side. Colors go from red to yellow on the positive side and through blue on the negative side.
	Blue - Green - Orange	A three-part color map with blue at the low end, green in the middle, and orange at the high end.
	Yellow - Gray - Blue	A three-part color map with yellow at the low end, gray in the middle, and blue at the high end.
	Rainbow Uniform	A color table that spans the hues of a rainbow. There have been many scientific perceptual studies on the effectiveness of rainbow colors, and they uniformly found them to be ineffective. This color table modifies the hues to make them more perceptually uniform, which should improve the effectiveness of the colors. However, we still recommend the other color tables over this one.
	Jet	A rainbow color table that adds some darkness for greater perceptual resolution. The ends of the jet color table might be too dark for 3D surfaces.
	Rainbow Desaturated	All the badness of the rainbow color table with periodic dark points added, which can help identify rate of change.

CHAPTER

ELEVEN

ERROR HANDLING

VTK-m contains several mechanisms for checking and reporting error conditions.

11.1 Runtime Error Exceptions

VTK-m uses exceptions to report errors. All exceptions thrown by VTK-m will be a subclass of vtkm::cont::Error. For simple error reporting, it is possible to simply catch a vtkm::cont::Error and report the error message string reported by the Error::GetMessage method.

Example 11.1: Simple error reporting.

```
int main(int argc, char** argv)
{
  try
  {
    // Do something cool with VTK-m
    // ...
  }
  catch (const vtkm::cont::Error& error)
  {
    std::cout << error.GetMessage() << std::endl;
    return 1;
  }
  return 0;
}
```

There are several subclasses to vtkm::cont::Error. The specific subclass gives an indication of the type of error that occured when the exception was thrown. Catching one of these subclasses may help a program better recover from errors.

vtkm::cont::ErrorBadAllocation Thrown when there is a problem accessing or manipulating memory. Often this is thrown when an allocation fails because there is insufficient memory, but other memory access errors can cause this to be thrown as well.

vtkm::cont::ErrorBadType Thrown when VTK-m attempts to perform an operation on an object that is of an incompatible type.

vtkm::cont::ErrorBadValue Thrown when a VTK-m function or method encounters an invalid value that inhibits progress.

vtkm::cont::ErrorExecution Throw when an error is signaled in the execution environment for example when a worklet is being executed.

vtkm::cont::ErrorInternal Thrown when VTK-m detects an internal state that should never be reached. This error usually indicates a bug in VTK-m or, at best, VTK-m failed to detect an invalid input it should have.

vtkm::io::ErrorIO Thrown by a reader or writer when a file error is encountered.

11.2 Asserting Conditions

In addition to the aforementioned error signaling, the vtkm/Assert.h header file defines a macro named VTKM_ASSERT. This macro behaves the same as the POSIX assert macro. It takes a single argument that is a condition that is expected to be true. If it is not true, the program is halted and a message is printed. Asserts are useful debugging tools to ensure that software is behaving and being used as expected.

Example 11.2: Using VTKM_ASSERT.

```
template<typename T>
VTKM_CONT T GetArrayValue(vtkm::cont::ArrayHandle<T> arrayHandle, vtkm::Id index)
{
  VTKM_ASSERT(index >= 0);
  VTKM_ASSERT(index < arrayHandle.GetNumberOfValues());
```

Did you know?

Like the POSIX assert, if the NDEBUG macro is defined, then VTKM_ASSERT will become an empty expression. Typically NDEBUG is defined with a compiler flag (like -DNDEBUG) for release builds to better optimize the code. CMake will automatically add this flag for release builds.

Common Errors

A helpful warning provided by many compilers alerts you of unused variables. (This warning is commonly enabled on VTK-m regression test nightly builds.) If a function argument is used only in a VTKM_ASSERT, then it will be required for debug builds and be unused in release builds. To get around this problem, add a statement to the function of the form (void)*variableName*;*. This statement will have no effect on the code generated but will suppress the warning for release builds.*

11.3 Compile Time Checks

Because VTK-m makes heavy use of C++ templates, it is possible that these templates could be used with inappropriate types in the arguments. Using an unexpected type in a template can lead to very confusing errors, so it is better to catch such problems as early as possible. The VTKM_STATIC_ASSERT macro, defined in vtkm/StaticAssert.h makes this possible. This macro takes a constant expression that can be evaluated at compile time and verifies that the result is true.

In the following example, VTKM_STATIC_ASSERT and its sister macro VTKM_STATIC_ASSERT_MSG, which allows you to give a descriptive message for the failure, are used to implement checks on a templated function that is designed to work on any scalar type that is represented by 32 or more bits.

Example 11.3: Using VTKM_STATIC_ASSERT.

```
template<typename T>
VTKM_EXEC_CONT void MyMathFunction(T& value)
{
  VTKM_STATIC_ASSERT((std::is_same<typename vtkm::TypeTraits<T>::DimensionalityTag,
                                   vtkm::TypeTraitsScalarTag>::value));

  VTKM_STATIC_ASSERT_MSG(sizeof(T) >= 4,
                         "MyMathFunction needs types with at least 32 bits.");
```

Did you know?

In addition to the several trait template classes provided by VTK-m to introspect C++ types, the C++ standard type_traits *header file contains several helpful templates for general queries on types. Example 11.3 demonstrates the use of one such template:* `std::is_same`*.*

Common Errors

Many templates used to introspect types resolve to the tags `std::true_type` *and* `std::false_type` *rather than the constant values* `true` *and* `false` *that* `VTKM_STATIC_ASSERT` *expects. The* `std::true_type` *and* `std::false_type` *tags can be converted to the Boolean literal by adding* `::value` *to the end of them. Failing to do so will cause* `VTKM_STATIC_ASSERT` *to behave incorrectly. Example 11.3 demonstrates getting the Boolean literal from the result of* `std::is_same`*.*

CHAPTER

TWELVE

MANAGING DEVICES

Multiple vendors vie to provide accelerator-type processors. VTK-m endeavors to support as many such architectures as possible. Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a *device adapter*.

So far in Part II we have been writing code that runs on a local serial CPU. In those examples where we run a filter, VTK-m is launching parallel execution in the execution environment. Internally VTK-m uses a device adapter to manage this execution.

A build of VTK-m generally supports multiple device adapters. In this chapter we describe how to represent and manage devices.

12.1 Device Adapter Tag

A device adapter is identified by a *device adapter tag*. This tag, which is simply an empty struct type, is used as the template parameter for several classes in the VTK-m control environment and causes these classes to direct their work to a particular device. The following device adapter tags are available in VTK-m.

`vtkm::cont::DeviceAdapterTagSerial` Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available. This tag is defined in vtkm/cont/DeviceAdapterSerial.h.

`vtkm::cont::DeviceAdapterTagCuda` Uses a CUDA capable GPU device. For this device to work, VTK-m must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler. This tag is defined in vtkm/cont/cuda/DeviceAdapterCuda.h.

`vtkm::cont::DeviceAdapterTagOpenMP` Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. This tag is defined in vtkm/cont/openmp/DeviceAdapterOpenMP.h.

`vtkm::cont::DeviceAdapterTagTBB` Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use TBB and the executable must be linked to the TBB library. This tag is defined in vtkm/cont/tbb/DeviceAdapterTBB.h.

The following example uses the tag for the Intel Threading Building blocks device adapter to specify a specific device for VTK-m to use. (Details on specifying devices in VTK-m is provided in Section 12.3.)

Example 12.1: Specifying a device using a device adapter tag.

```
  vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagTBB{});
```

For classes and methods that have a template argument that is expected to be a device adapter tag, the tag type can be checked with the `VTKM_IS_DEVICE_ADAPTER_TAG` macro to verify the type is a valid device adapter tag. It is good practice to check unknown types with this macro to prevent further unexpected errors.

12.2 Device Adapter Id

Using a device adapter tag directly means that the type of device needs to be known at compile time. To store a device adapter type at run time, one can instead use `vtkm::cont::DeviceAdapterId`. `DeviceAdapterId` is a superclass to all the device adapter tags, and any device adapter tag can be "stored" in a `DeviceAdapterId`. Thus, it is more common for functions and classes to use `DeviceAdapterId` then to try to track a specific device with templated code.

In addition to the provided device adapter tags listed previously, a `DeviceAdapterId` can store some special device adapter tags that do not directly specify a specific device.

`vtkm::cont::DeviceAdapterTagAny` Used to specify that any device may be used for an operation. In practice this is limited to devices that are currently available.

`vtkm::cont::DeviceAdapterTagUndefined` Used to avoid specifying a device. Useful as a placeholder when a device can be specified but none is given.

Did you know?

Any device adapter tag can be used where a device adapter id is expected. Thus, you can use a device adapter tag whenever you want to specify a particular device and pass that to any method expecting a device id. Likewise, it is usually more convenient for classes and methods to manage device adapter ids rather than device adapter tag.

`DeviceAdapterId` contains several helpful methods to get runtime information about a particular device.

`GetName` A `static` method that returns a string description for the device adapter. The string is stored in a type named `vtkm::cont::DeviceAdapterNameType`, which is currently aliased to `std::string`. The device adapter name is useful for printing information about a device being used.

`GetId` A `static` method taking no arguments that returns a unique integer identifier for the device adapter as a `vtkm::Int8`.

`IsValueValid` A `static const bool` that is true if the implementation of the integer returned from `GetId` corresponds to a concrete device. So, for example, the `IsValueValid` flag for a `DeviceAdapterTagSerial` is true whereas the `IsValueValid` flag for a `DeviceAdapterTagAny` is false.

Did you know?

As a cheat, all device adapter tags actually inherit from the `vtkm::cont::DeviceAdapterId` *class. Thus, all of these methods can be called directly on a device adapter tag.*

Common Errors

Just because the DeviceAdapterId::IsValueValid *returns true that does not necessarily mean that this device is available to be run on. It simply means that the device is implemented in VTK-m. However, that device might not be compiled, or that device might not be available on the current running system, or that device might not be enabled. Use the device runtime tracker described in Section 12.3 to determine if a particular device can actually be used.*

12.3 Runtime Device Tracker

It is often the case that you are agnostic about what device VTK-m algorithms run so long as they complete correctly and as fast as possible. Thus, rather than directly specify a device adapter, you would like VTK-m to try using the best available device, and if that does not work try a different device. Because of this, there are many features in VTK-m that behave this way. For example, you may have noticed that running filters, as in the examples of Chapter 9, you do not need to specify a device; they choose a device for you.

However, even though we often would like VTK-m to choose a device for us, we still need a way to manage device preferences. VTK-m also needs a mechanism to record runtime information about what devices are available so that it does not have to continually try (and fail) to use devices that are not available at runtime. These needs are met with the vtkm::cont::RuntimeDeviceTracker class. RuntimeDeviceTracker maintains information about which devices can and should be run on. VTK-m maintains a RuntimeDeviceTracker for each thread your code is operating on. To get the runtime device for the current thread, use the vtkm::cont::GetRuntimeDeviceTracker method.

RuntimeDeviceTracker has the following methods.

CanRunOn Takes a device adapter tag and returns true if VTK-m was configured for the device and it has not yet been marked as disabled.

DisableDevice Takes a device adapter tag and marks that device to not be used. Future calls to CanRunOn for this device will return false until that device is reset.

ResetDevice Takes a vtkm::cont::DeviceAdapterTag and resets the state for that device to its default value. Each device defaults to on as long as VTK-m is configured to use that device and a basic runtime check finds a viable device.

Reset Resets all devices. This equivocally calls ResetDevice for all devices supported by VTK-m.

ForceDevice Takes a device adapter tag and enables that device. All other devices are disabled. This method throws a vtkm::cont::ErrorBadValue if the requested device cannot be enabled.

ReportAllocationFailure A device might have less working memory available than the main CPU. If this is the case, memory allocation errors are more likely to happen. This method is used to report a vtkm::cont::ErrorBadAllocation and disables the device for future execution.

ReportBadDeviceFailure It is possible that a device may throw a vtkm::cont::ErrorBadDevice failure caused by some erroneous device issue. If this occurs, it is possible to catch the vtkm::cont::ErrorBadDevice exception and pass it to ReportBadDeviceFailure along with the vtkm::cont::DeviceAdapterId to forcefully disable a device.

A RuntimeDeviceTracker can be used to specify which devices to consider for a particular operation. However, a better way to specify devices is to use the vtkm::cont::ScopedRuntimeDeviceTracker class. When a ScopedRuntimeDeviceTracker is constructed, it specifies a new set of devices for VTK-m to use. When the ScopedRuntimeDeviceTracker is destroyed as it leaves scope, it restores VTK-m's devices to those that existed when it was created.

The following example demonstrates how the ScopedRuntimeDeviceTracker is used to force the VTK-m operations that happen within a function to operate exclusively with the TBB device.

Example 12.2: Restricting which devices VTK-m uses per thread.

```
void ChangeDefaultRuntime()
{
  std::cout << "Checking changing default runtime." << std::endl;

  vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagTBB{});

  // VTK-m operations limited to serial devices here...

  // Devices restored as we leave scope.
}
```

In the previous example we forced VTK-m to use the TBB device. This is the default behavior of ScopedRuntimeDeviceTracker, but the constructor takes an optional second argument that is a value in the vtkm::cont::RuntimeDeviceTrackerMode to specify how modify the current device adapter list.

RuntimeDeviceTrackerMode::Force Replaces the current list of devices to try with the device specified to the ScopedRuntimeDeviceTracker. This has the effect of forcing VTK-m to use the provided device. This is the default behavior for the ScopedRuntimeDeviceTracker.

RuntimeDeviceTrackerMode::Enable Adds the provided device adapter to the list of devices to try.

RuntimeDeviceTrackerMode::Disable Removes the provided device adapter from the list of devices to try.

As a motivating example, let us say that we want to perform a deep copy of an array (described in Section 16.2). However, we do not want to do the copy on a CUDA device because we happen to know the data is not on that device and we do not want to spend the time to transfer the data to that device. We can use a vtkm::cont::ScopedRuntimeDeviceTracker to temporarily disable the CUDA device for this operation.

Example 12.3: Disabling a device with RuntimeDeviceTracker.

```
  vtkm::cont::ScopedRuntimeDeviceTracker tracker(
    vtkm::cont::DeviceAdapterTagCuda(),
    vtkm::cont::RuntimeDeviceTrackerMode::Disable);

  vtkm::cont::ArrayCopy(srcArray, destArray);
```

CHAPTER

THIRTEEN

TIMERS

It is often the case that you need to measure the time it takes for an operation to happen. This could be for performing measurements for algorithm study or it could be to dynamically adjust scheduling.

Performing timing in a multi-threaded environment can be tricky because operations happen asynchronously. To ensure that accurate timings can be made, VTK-m provides a `vtkm::cont::Timer` class to provide an accurate measurement of operations that happen on devices that VTK-m can use. By default, `Timer` will time operations on all possible devices.

The timer is started by calling the `Timer::Start` method. The timer can subsequently be stopped by calling `Timer::Stop`. The time elapsed between calls to `Start` and `Stop` (or the current time if `Stop` was not called) can be retrieved with a call to the `Timer::GetElapsedTime` method. Subsequently calling `Start` again will restart the timer.

Example 13.1: Using `vtkm::cont::Timer`.

```
vtkm::filter::PointElevation elevationFilter;
elevationFilter.SetUseCoordinateSystemAsField(true);
elevationFilter.SetOutputFieldName("elevation");

vtkm::cont::Timer timer;

timer.Start();

vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);

// This code makes sure data is pulled back to the host in a host/device
// architecture.
vtkm::cont::ArrayHandle<vtkm::Float64> outArray;
result.GetField("elevation").GetData().CopyTo(outArray);
outArray.GetPortalConstControl();

timer.Stop();

vtkm::Float64 elapsedTime = timer.GetElapsedTime();

std::cout << "Time to run: " << elapsedTime << std::endl;
```

Common Errors

Some device require data to be copied between the host CPU and the device. In this case you might want *to measure the time to copy data back to the host. This can be done by "touching" the data on the host by getting a control portal.*

The VTK-m Timer does its best to capture the time it takes for all parallel operations run between calls to Start and Stop to complete. It does so by synchronizing to concurrent execution on devices that might be in use.

Common Errors

Because Timer synchronizes with devices (essentially waiting for the device to finish executing), that can have an effect on how your program runs. Be aware that using a Timer can itself change the performance of your code. In particular, starting and stopping the timer many times to measure the parts of a sequence of operations can potentially make the whole operation run slower.

By default, Timer will synchronize with all active devices. However, if you want to measure the time for a specific device, then you can pass the device adapter tag or id to vtkm::cont::Timer's constructor. You can also change the device being used by passing a device adapter tag or id to the Timer::Reset method. A device can also be specified through an optional argument to the Timer::GetElapsedTime method.

The following methods are provided by vtkm::cont::Timer.

Start Causes the Timer to begin timing. The elapsed time will record an interval beginning when this method is called.

Started Returns true if Start has been called. It is invalid to try to get the elapsed time if Started is not true.

Stop Causes the Timer to finish timing. The elapsed time will record an interval ending when this method is called. It is invalid to stop the timer if Started is not true.

Stopped Returns true if Stop has been called. If Stopped is true, then the elapsed time will no longer increase. If Stopped is false and Started is true, then the timer is still running.

Ready Returns true if the timer has finished the synchronization required to get the timing result from the device.

GetElapsedTime Returns the amount of time that has elapsed between calling Start and Stop. If Stop was not called, then the amount of time between calling Start and GetElapsedTime is returned. GetElapsedTime can optionally take a device adapter tag or id to specify for which device to return the elapsed time.

Reset Restores the initial state of the Timer. All previous recorded time is erased. Reset optionally takes a device adapter tag or id that specifies on which device to time and synchronize.

GetDevice Returns the id of the device adapter for which this timer is synchronized. If the device adapter has the same id as vtkm::cont::DeviceAdapterTagAny (the default), then the timer will synchronize on all devices.

CHAPTER

FOURTEEN

IMPLICIT FUNCTIONS

VTK-m's implicit Functions are objects that are constructed with values representing 3D spatial coordinates that often describe a shape. Each implicit function is typically defined by the surface formed where the value of the function is equal to 0. All vtkm::ImplicitFunction implement Value and Gradient methods that describe the orientation of a provided point with respect to the vtkm::ImplicitFunction's shape.

Value The Value method for a vtkm::ImplicitFunction takes a vtkm::Vec3f and returns a vtkm::FloatDefault representing the orientation of the point with respect to the vtkm::ImplicitFunction's shape. Negative scalar values represent vector points inside of the vtkm::ImplicitFunction's shape. Positive scalar values represent vector points outside the vtkm::ImplicitFunction's shape. Zero values represent vector points that lie on the surface of the vtkm::ImplicitFunction

Gradient The Gradient method for a vtkm::ImplicitFunction takes a vtkm::Vec3f and returns a vtkm::Vec3f representing the pointing direction from the vtkm::ImplicitFunction's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

Implicit functions are useful when trying to clip regions from a dataset. For example, it is possible to use vtkm::filter::ClipWithImplicitFunction to remove a region in a provided dataset according to the shape of an implicit function. See Section9.1.4 for more information on clipping with implicit functions.

14.1 Provided Implicit Functions

VTK-m has implementations of various implicit functions provided by the following sublcasses.

14.1.1 Plane

vtkm::Plane defines an infinite plane. The plane is defined by a pair of vtkm::Vec3f values that represent the origin, which is any point on the plane, and a normal, which is a vector that is tangent to the plane. These are set with the SetOrigin and SetNormal methods, respectively. Planes extend infinitely from the origin point in the direction perpendicular form the Normal. An example vtkm::Plane is shown in Figure 14.1.

14.1.2 Sphere

vtkm::Sphere defines a sphere. The Sphere is defined by a center location and a radius, which are set with the SetCenter and SetRadius methods, respectively. An example vtkm::Sphere is shown in Figure 14.2.

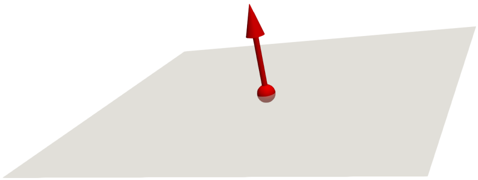


Figure 14.1: Visual Representation of an Implicit Plane. The red dot and arrow represent the origin and normal of the plane, respectively. For demonstrative purposes the plane as shown with limited area, but in actuality the plane extends infinitely.

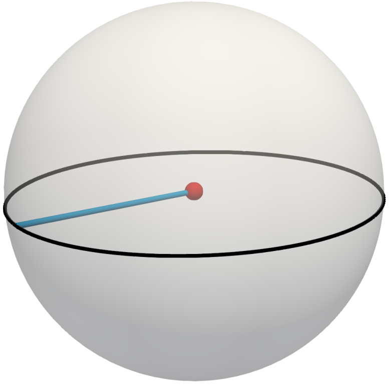


Figure 14.2: Visual Representation of an Implicit Sphere. The red dot represents the center of the sphere. The radius is the length of any line (like the blue one shown here) that extends from the center in any direction to the surface.

14.1.3 Cylinder

vtkm::Cylinder defines a cylinder that extends infinitely along its axis. The cylinder is defined with a center point, a direction of the center axis, and a radius, which are set with `SetCenter`, `SetAxis`, and `SetRadius`, respectively. An example vtkm::Cylinder is shown in Figure 14.3 with set origin, radius, and axis values.

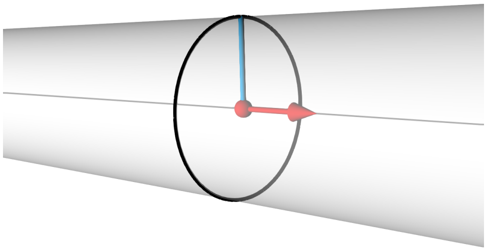


Figure 14.3: Visual Representation of an Implicit Cylinder. The red dot represents the center value, and the red arrow represents the vector that points in the direction of the axis. The radius is the length of any line (like the blue one shown here) that extends perpendicular from the axis to the surface.

14.1.4 Box

vtkm::Box defines an axis-aligned box. The box is defined with a pair of vtkm::Vec3f values that represent the minimum point coordinates and maximum point coordinates, which are set with SetMinPoint and SetMaxPoint, respectively. The Box is the shape enclosed by intersecting axis-parallel lines drawn from each point. Alternately, the Box can be specified with a vtkm::Bounds object using the SetBounds method. An example vtkm::Box is shown in Figure 14.4.

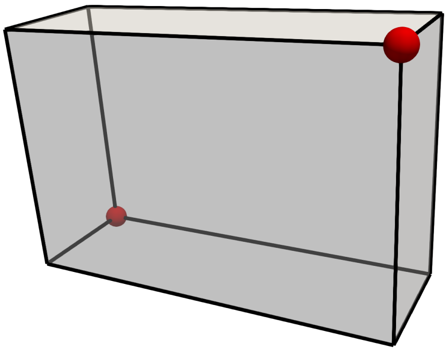


Figure 14.4: Visual Representation of an Implicit Box. The red dots represent the minimum and maximum points.

14.1.5 Frustum

vtkm::Frustum defines a hexahedral region with potentially oblique faces. A Frustum is typically used to define the tapered region of space visible in a perspective camera projection. The frustum is defined by the 6 planes that make up its 6 faces. Each plane is defined by a point and a normal vector, which are set with SetPlane and SetNormal, respectively. Parameters for all 6 planes can be set at once using the SetPlanes and SetNormals methods. Alternately, the Frustum can be defined by the 8 points at the vertices of the enclosing hexahedron using the CreateFromPoints method. The points given to CreateFromPoints must be in hex-cell order where the first four points are assumed to be a plane, and the last four points are assumed to be a plane. An example vtkm::Frustum is shown in Figure 14.5.

14.2 Implicit Function Handles

To support VTK-m's device agnostic execution environment, VTK-m also provides a vtkm::cont::ImplicitFunctionHandle. The ImplicitFunction that your code creates on the host processor of your computer might not work as is on a device that a filter executes on. Thus the ImplicitFunction object needs to be transformed before it is ready for running in a parallel algorithm.

This transformation is managed by the vtkm::cont::ImplicitFunctionHandle class. ImplicitFunctionHandle holds a reference to an ImplicitFunction. The reference of ImplicitFunctionHandle is not dependent on the type of ImplicitFunction being held, so it is a convenient way to accept a user-defined function. ImplicitFunctionHandle also manages the transfer of the ImplicitFunction to the device.

An ImplicitFunctionHandle is easily created with the vtkm::cont::make_ImplicitFunctionHandle function. The following example demonstrates creating an ImplicitFunction and then putting it into an ImplicitFunctionHandle for use with the vtkm::filter::ClipWithImplicitFunction filter.

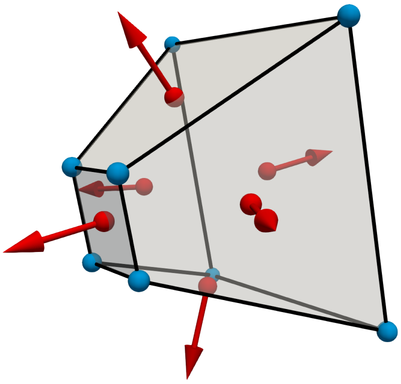


Figure 14.5: Visual Representation of an Implicit Frustum. The red dots and arrows represent the points and normals defining each enclosing plane. The blue dots represent the 8 vertices, which can also be used to define the frustum.

Example 14.1: Using `ImplicitFunctionHandle`.

```
// Parameters needed for implicit function
vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);

// Create an instance of a clip filter with this implicit function.
vtkm::filter::ClipWithImplicitFunction clip;
clip.SetImplicitFunction(
  vtkm::cont::make_ImplicitFunctionHandle(implicitFunction));
```

Part III

Developing Algorithms

CHAPTER

FIFTEEN

GENERAL APPROACH

VTK-m is designed to provide a *pervasive parallelism* throughout all its visualization algorithms, meaning that the algorithm is designed to operate with independent concurrency at the finest possible level throughout. VTK-m provides this pervasive parallelism by providing a programming construct called a *worklet*, which operates on a very fine granularity of data. The worklets are designed as serial components, and VTK-m handles whatever layers of concurrency are necessary, thereby removing the onus from the visualization algorithm developer. Worklet operation is then wrapped into *filters*, which provide a simplified interface to end users.

A worklet is essentially a functor or kernel designed to operate on a small element of data. (The name "worklet" means a small amount of the work. We mean small in this sense to be the amount of data, not necessarily the amount of instructions performed.) The worklet is constrained to contain a serial and stateless function. These constraints form three critical purposes. First, the constraints on the worklets allow VTK-m to schedule worklet invocations on a great many independent concurrent threads and thereby making the algorithm pervasively parallel. Second, the constraints allow VTK-m to provide thread safety. By controlling the memory access the toolkit can insure that no worklet will have any memory collisions, false sharing, or other parallel programming pitfalls. Third, the constraints encourage good programming practices. The worklet model provides a natural approach to visualization algorithm design that also has good general performance characteristics.

VTK-m allows developers to design algorithms that are run on massive amounts of threads. However, VTK-m also allows developers to interface to applications, define data, and invoke algorithms that they have written or are provided otherwise. These two modes represent significantly different operations on the data. The operating code of an algorithm in a worklet is constrained to access only a small portion of data that is provided by the framework. Conversely, code that is building the data structures needs to manage the data in its entirety, but has little reason to perform computations on any particular element.

Consequently, VTK-m is divided into two *environments* that handle each of these use cases. Each environment has its own API, and direct interaction between the environments is disallowed. The environments are as follows.

Execution Environment This is the environment in which the computational portion of algorithms are executed. The API for this environment provides work for one element with convenient access to information such as connectivity and neighborhood as needed by typical visualization algorithms. Code for the execution environment is designed to always execute on a very large number of threads.

Control Environment This is the environment that is used to interface with applications, interface with I/O devices, and schedule parallel execution of the algorithms. The associated API is designed for users that want to use VTK-m to analyze their data using provided or supplied filters. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

These dual programming environments are partially a convenience to isolate the application from the execution of the worklets and are partially a necessity to support GPU languages with host and device environments. The control and execution environments are logically equivalent to the host and device environments, respectively, in CUDA and other associated GPU languages.

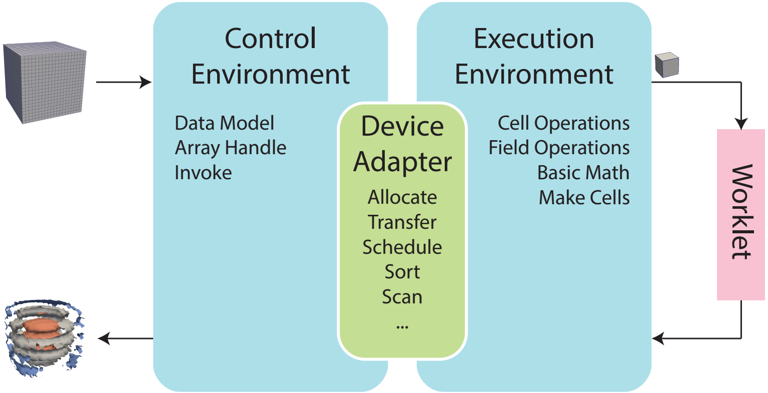


Figure 15.1: Diagram of the VTK-m framework.

Figure 15.1 displays the relationship between the control and execution environment. The typical workflow when using VTK-m is that first the control thread establishes a data set in the control environment and then invokes a parallel operation on the data using a filter. From there the data is logically divided into its constituent elements, which are sent to independent invocations of a worklet. The worklet invocations, being independent, are run on as many concurrent threads as are supported by the device. On completion the results of the worklet invocations are collected to a single data structure and a handle is returned back to the control environment.

Did you know?

Are you only planning to use filters in VTK-m that already exist? If so, then everything you work with will be in the control environment. The execution environment is only used when implementing algorithms for filters.

15.1 Package Structure

VTK-m is organized in a hierarchy of nested packages. VTK-m places definitions in *namespaces* that correspond to the package (with the exception that one package may specialize a template defined in a different namespace).

The base package is named `vtkm` . All classes within VTK-m are placed either directly in the `vtkm` package or in a package beneath it. This helps prevent name collisions between VTK-m and any other library.

As described at the beginning of this chapter, the VTK-m API is divided into two distinct environments: the control environment and the execution environment. The API for these two environments are located in the `vtkm::cont` and `vtkm::exec` packages, respectively. Items located in the base `vtkm` namespace are available in both environments.

Although it is conventional to spell out names in identifiers,[1] there is an exception to abbreviate control and execution to `cont` and `exec`, respectively. This is because it is also part of the coding convention to declare the entire namespace when using an identifier that is part of the corresponding package. The shorter names

[1] VTK-m coding conventions are outlined in the doc/CodingConventions.md file in the VTK-m source code and at https://gitlab.kitware.com/vtk/vtk-m/blob/master/docs/CodingConventions.md

make the identifiers easier to read, faster to type, and more feasible to pack lines in 80 column displays. These abbreviations are also used instead of more common abbreviations (e.g. ctrl for control) because, as part of actual English words, they are easier to type.

Further functionality in VTK-m is built on top of the base vtkm , vtkm::cont , and vtkm::exec packages. Support classes for building worklets, introduced in Chapter 17, are contained in the vtkm::worklet package. Other facilities in VTK-m are provided in their own packages such as vtkm::io , vtkm::filter , and vtkm::rendering . These packages are described in Part II.

VTK-m contains code that uses specialized compiler features, such as those with CUDA, or libraries, such as Intel Threading Building Blocks, that will not be available on all machines. Code for these features are encapsulated in their own packages under the vtkm::cont namespace: vtkm::cont::cuda and vtkm::cont::tbb .

VTK-m contains OpenGL interoperability that allows data generated with VTK-m to be efficiently transferred to OpenGL objects. This feature is encapsulated in the vtkm::opengl package.

Figure 15.2 provides a diagram of the VTK-m package hierarchy.

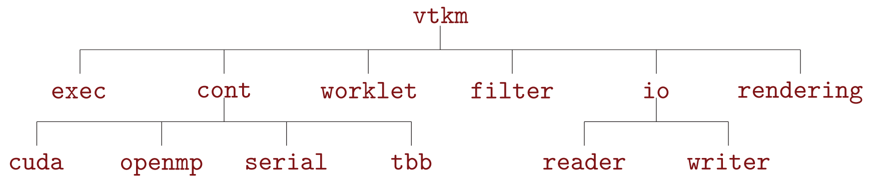


Figure 15.2: VTK-m package hierarchy.

By convention all classes will be defined in a file with the same name as the class name (with a .h extension) located in a directory corresponding to the package name. For example, the vtkm::cont::DataSet class is found in the vtkm/cont/DataSet.h header. There are, however, exceptions to this rule. Some smaller classes and types are grouped together for convenience. These exceptions will be noted as necessary.

Within each namespace there may also be internal and detail sub-namespaces. The internal namespaces contain features that are used internally and may change without notice. The detail namespaces contain features that are used by a particular class but must be declared outside of that class. Users should generally ignore classes in these namespaces.

15.2 Function and Method Environment Modifiers

Any function or method defined by VTK-m must come with a modifier that determines in which environments the function may be run. These modifiers are C macros that VTK-m uses to instruct the compiler for which architectures to compile each method. Most user code outside of VTK-m need not use these macros with the important exception of any classes passed to VTK-m. This occurs when defining new worklets, array storage, and device adapters.

VTK-m provides three modifier macros, VTKM_CONT, VTKM_EXEC, and VTKM_EXEC_CONT, which are used to declare functions and methods that can run in the control environment, execution environment, and both environments, respectively. These macros get defined by including just about any VTK-m header file, but including vtkm/Types.h will ensure they are defined.

The modifier macro is placed after the template declaration, if there is one, and before the return type for the function. Here is a simple example of a function that will square a value. Since most types you would use this function on have operators in both the control and execution environments, the function is declared for both places.

Example 15.1: Usage of an environment modifier macro on a function.

```
template<typename ValueType>
VTKM_EXEC_CONT ValueType Square(const ValueType& inValue)
{
  return inValue * inValue;
}
```

The primary function of the modifier macros is to inject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control modifiers have `__host__` in them and execution modifiers have `__device__` in them.

It is sometimes the case that a function declared as VTKM_EXEC_CONT has to call a method declared as VTKM_EXEC or VTKM_CONT. Generally functions should not call other functions with incompatible control/execution modifiers, but sometimes a generic VTKM_EXEC_CONT function calls another function determined by the template parameters, and the valid environments of this subfunction may be inconsistent. For cases like this, you can use the VTKM_SUPPRESS_EXEC_WARNINGS to tell the compiler to ignore the inconsistency when resolving the template. When applied to a templated function or method, VTKM_SUPPRESS_EXEC_WARNINGS is placed before the `template` keyword. When applied to a non-templated method in a templated class, VTKM_SUPPRESS_EXEC_WARNINGS is placed before the environment modifier macro.

Example 15.2: Suppressing warnings about functions from mixed environments.

```
VTKM_SUPPRESS_EXEC_WARNINGS
template<typename Functor>
VTKM_EXEC_CONT void OverlyComplicatedForLoop(Functor& functor,
                                             vtkm::Id numInterations)
{
  for (vtkm::Id index = 0; index < numInterations; index++)
  {
    functor();
  }
}
```

CHAPTER

SIXTEEN

BASIC ARRAY HANDLES

Chapter 7 describes the basic data sets used by VTK-m. This chapter dives deeper into how VTK-m represents data. Ultimately, data structures like `vtkm::cont::DataSet` can be broken down into arrays of numbers. Arrays in VTK-m are managed by a unit called an *array handle*.

An array handle, which is implemented with the `vtkm::cont::ArrayHandle` class, manages an array of data that can be accessed or manipulated by VTK-m algorithms. It is typical to construct an array handle in the control environment to pass data to an algorithm running in the execution environment. It is also typical for an algorithm running in the execution environment to populate an array handle, which can then be read back in the control environment. It is also possible for an array handle to manage data created by one VTK-m algorithm and passed to another, remaining in the execution environment the whole time and never copied to the control environment.

Did you know?

The array handle may have up to two copies of the array, one for the control environment and one for the execution environment. However, depending on the device and how the array is being used, the array handle will only have one copy when possible. Copies between the environments are implicit and lazy. They are copied only when an operation needs data in an environment where the data is not.

`vtkm::cont::ArrayHandle` behaves like a shared smart pointer in that when the C++ object is copied, each copy holds a reference to the same array. These copies are reference counted so that when all copies of the `vtkm::cont::ArrayHandle` are destroyed, any allocated memory is released.

An `ArrayHandle` defines the following methods. Note, however, that the brief overview of this chapter will not cover the use of most of these methods. Further descriptions are given in later chapters that explore even further the features of `ArrayHandle`s.

`GetNumberOfValues` Returns the number of entries in the array.

`Allocate` Resizes the array to include the number of entries given. Any previously stored data might be discarded.

`Shrink` Resizes the array to the number of entries given. Any data stored in the array is preserved. The number of entries must be less than those given in the last call to `Allocate`.

`ReleaseResourcesExecution` If the `ArrayHandle` is holding any data on a device (such as a GPU), that memory is released to be used elsewhere. No data is lost from this call. Any data on the released resources is copied to the control environment (the local CPU) before the memory is released.

ReleaseResources Releases all memory managed by this ArrayHandle. Any data in this memory is lost.

SyncControlArray Makes sure any data in the execution environment is also available in the control environment. This method is useful when timing parallel algorithms and you want to include the time to transfer data between parallel devices and their hosts.

GetPortalControl Returns an array portal that can be used to access the data in the array handle in the control environment. Array portals are described in Section 27.1.

GetPortalConstControl Like ArrayHandle::GetPortalControl but returns a read-only array portal rather than a read/write array portal.

PrepareForInput Readies the data as input to a parallel algorithm. See Section 27.4 for more details.

PrepareForOutput Readies the data as output to a parallel algorithm. See Section 27.4 for more details.

PrepareForInPlace Readies the data as input and output to a parallel algorithm. See Section 27.4 for more details.

GetDeviceAdapterId Returns a vtkm::cont::DeviceAdapterId describing on which device adapter, if any, the array handle's data is available. Device adapter ids are described in Section 12.2.

GetStorage Returns the vtkm::cont::Storage object that manages the data. The type of the storage object is defined by the storage tag template parameter of the ArrayHandle. Storage objects are described in detail in Chapter 36.

16.1 Creating Array Handles

vtkm::cont::ArrayHandle is templated on the type of values being stored in the array. There are multiple ways to create and populate an array handle. The default vtkm::cont::ArrayHandle constructor will create an empty array with nothing allocated in either the control or execution environment. This is convenient for creating arrays used as the output for algorithms.

Example 16.1: Creating an ArrayHandle for output data.

```
vtkm::cont::ArrayHandle<vtkm::Float32> outputArray;
```

Constructing an ArrayHandle that points to a provided C array or std::vector is straightforward with the vtkm::cont::make_ArrayHandle functions. These functions will make an array handle that points to the array data that you provide.

Example 16.2: Creating an ArrayHandle that points to a provided C array.

```
vtkm::Float32 dataBuffer[50];
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
  vtkm::cont::make_ArrayHandle(dataBuffer, 50);
```

Example 16.3: Creating an ArrayHandle that points to a provided std::vector.

```
std::vector<vtkm::Float32> dataBuffer;
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
  vtkm::cont::make_ArrayHandle(dataBuffer);
```

Be aware that vtkm::cont::make_ArrayHandle by default makes a shallow pointer copy. This means that if you change or delete the data provided, the internal state of ArrayHandle becomes invalid and undefined behavior can ensue. The most common manifestation of this error happens when a std::vector goes out of scope. This subtle interaction will cause the vtkm::cont::ArrayHandle to point to an unallocated portion of the memory heap. For example, if the code in Example 16.3 were to be placed within a callable function or method, it could cause the vtkm::cont::ArrayHandle to become invalid.

Example 16.4: Invalidating an ArrayHandle by letting the source std::vector leave scope.

```
 1 VTKM_CONT
 2 vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
 3 {
 4   std::vector<vtkm::Float32> dataBuffer;
 5   // Populate dataBuffer with meaningful data. Perhaps read data from a file.
 6
 7   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
 8     vtkm::cont::make_ArrayHandle(dataBuffer);
 9
10   return inputArray;
11   // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
12   // memory. However, inputArray has a pointer to that memory, which becomes an
13   // invalid pointer in the returned object. Bad things will happen when the
14   // ArrayHandle is used.
15 }
16
17 VTKM_CONT
18 vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad()
19 {
20   std::vector<vtkm::Float32> dataBuffer;
21   // Populate dataBuffer with meaningful data. Perhaps read data from a file.
22
23   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
24     vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::On);
25
26   return inputArray;
27   // This is safe.
28 }
```

An easy way around the problem of having an ArrayHandle's data going out of scope is to copy the data into the ArrayHandle. To make this easy, the make_ArrayHandle function takes an optional second argument of type vtkm::CopyFlag. If the second argument is CopyFlag::Off (the default), then the data is only shallow copied and errors will happen if the memory gets deallocated. However, if the second argument is CopyFlag::On, then the data are copied into the ArrayHandle. This solution is shown on line 24 of Example 16.4.

16.2 Deep Array Copies

As stated previously, an ArrayHandle object behaves as a smart pointer that copies references to the data without copying the data itself. This is clearly faster and more memory efficient than making copies of the data itself and usually the behavior desired. However, it is sometimes the case that you need to make a separate copy of the data.

To simplify copying the data, VTK-m comes with the vtkm::cont::ArrayCopy convenience function defined in vtkm/cont/ArrayCopy.h. ArrayCopy takes the array to copy from (the source) as its first argument and the array to copy to (the destination) as its second argument. The destination array will be properly reallocated to the correct size.

Example 16.5: Using ArrayCopy.

```
  vtkm::cont::ArrayCopy(srcArray, destArray);
```

Did you know?

ArrayCopy will copy data in parallel. If desired, you can specify the device as the third argument to ArrayCopy using either a device adapter tag or a runtime device tracker. Both the tags and tracker are described in Chapter 12.

16.3 The Hidden Second Template Parameter

We have already seen that vtkm::cont::ArrayHandle is a templated class with the template parameter indicating the type of values stored in the array. However, ArrayHandle has a second hidden parameter that indicates the *storage* of the array. We have so far been able to ignore this second template parameter because VTK-m will assign a default storage for us that will store the data in a basic array.

Example 16.6: Declaration of the vtkm::cont::ArrayHandle templated class.

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
class ArrayHandle;
```

Changing the storage of an ArrayHandle lets us do many weird and wonderful things. We will explore these options in later chapters, but for now we can ignore this second storage template parameter. However, there are a couple of things to note concerning the storage.

First, if the compiler gives an error concerning your use of ArrayHandle, the compiler will report the ArrayHandle type with not one but two template parameters. A second template parameter of vtkm::cont::StorageTagBasic can be ignored.

Second, if you write a function, method, or class that is templated based on an ArrayHandle type, it is good practice to accept ArrayHandles with non-default storage types. There are two ways to do this. The first way is to template both the value type and the storage type.

Example 16.7: Templating a function on an ArrayHandle's parameters

```
template<typename T, typename Storage>
void Foo(const vtkm::cont::ArrayHandle<T, Storage>& array)
{
```

The second way is to template the whole array type rather than the sub types. If you create a template where you expect one of the parameters to be an ArrayHandle, you should use the VTKM_IS_ARRAY_HANDLE macro to verify that the type is indeed an ArrayHandle.

Example 16.8: A template parameter that should be an ArrayHandle.

```
template<typename ArrayType>
void Bar(const ArrayType& array)
{
  VTKM_IS_ARRAY_HANDLE(ArrayType);
```

CHAPTER

SEVENTEEN

SIMPLE WORKLETS

The simplest way to implement an algorithm in VTK-m is to create a *worklet*. A worklet is fundamentally a functor that operates on an element of data. Thus, it is a `class` or `struct` that has an overloaded parenthesis operator (which must be declared `const` for thread safety). However, worklets are also embedded with a significant amount of metadata on how the data should be managed and how the execution should be structured.

Example 17.1: A simple worklet.

```
 1  struct PoundsPerSquareInchToNewtonsPerSquareMeterWorklet
 2    : vtkm::worklet::WorkletMapField
 3  {
 4    using ControlSignature = void(FieldIn psi, FieldOut nsm);
 5    using ExecutionSignature = void(_1, _2);
 6    using InputDomain = _1;
 7
 8    template<typename T>
 9    VTKM_EXEC void operator()(const T& psi, T& nsm) const
10    {
11      // 1 psi = 6894.76 N/m^2
12      nsm = T(6894.76f) * psi;
13    }
14  };
```

As can be seen in Example 17.1, a worklet is created by implementing a `class` or `struct` with the following features.

1. The class must publicly inherit from a base worklet class that specifies the type of operation being performed (line 1).

2. The class must contain a functional type named `ControlSignature` (line 4), which specifies what arguments are expected when invoking the class in the control environment.

3. The class must contain a functional type named `ExecutionSignature` (line 5), which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.

4. The class specifies an `InputDomain` (line 6), which identifies which input parameter defines the input domain of the data.

5. The class must contain an implementation of the parenthesis operator, which is the method that is executed in the execution environment (lines 8–13). The parenthesis operator must be declared `const`.

17.1 Control Signature

The control signature of a worklet is a functional type named ControlSignature. The function prototype matches what data are provided when the worklet is invoked (as described in Section 17.5).

Example 17.2: A ControlSignature.

```
using ControlSignature = void(FieldIn psi, FieldOut nsm);
```

Did you know?

If the code in Example 17.2 looks strange, you may be unfamiliar with function types. In C++, functions have types just as variables and classes do. A function with a prototype like

```
    void functionName(int arg1, float arg2);
```

has the type void(int,float)*. VTK-m uses function types like this as a* signature *that defines the structure of a function call.*

The return type of the function prototype is always void. The parameters of the function prototype are *tags* that identify the type of data that is expected to be passed to invoke. ControlSignature tags are defined by the worklet type and the various tags are documented more fully in Chapter 21. In the case of Example 17.2, the two tags FieldIn and FieldOut represent input and output data, respectively.

By convention, ControlSignature tag names start with the base concept (e.g. Field or Topology) followed by the domain (e.g. Point or Cell) followed by In or Out. For example, FieldPointIn would specify values for a field on the points of a mesh that are used as input (read only). Although they should be there in most cases, some tag names might leave out the domain or in/out parts if they are obvious or ambiguous.

17.2 Execution Signature

Like the control signature, the execution signature of a worklet is a functional type named ExecutionSignature. The function prototype must match the parenthesis operator (described in Section 17.4) in terms of arity and argument semantics.

Example 17.3: An ExecutionSignature.

```
using ExecutionSignature = void(_1, _2);
```

The arguments of the ExecutionSignature's function prototype are tags that define where the data come from. The most common tags are an underscore followed by a number, such as _1, _2, etc. These numbers refer back to the corresponding argument in the ControlSignature. For example, _1 means data from the first control signature argument, _2 means data from the second control signature argument, etc.

Unlike the control signature, the execution signature optionally can declare a return type if the parenthesis operator returns a value. If this is the case, the return value should be one of the numeric tags (i.e. _1, _2, etc.) to refer to one of the data structures of the control signature. If the parenthesis operator does not return a value, then ExecutionSignature should declare the return type as void.

In addition to the numeric tags, there are other execution signature tags to represent other types of data. For example, the WorkIndex tag identifies the instance of the worklet invocation. Each call to the worklet function will have a unique WorkIndex. Other such tags exist and are described in the following section on worklet types where appropriate.

17.3 Input Domain

All worklets represent data parallel operations that are executed over independent elements in some domain. The type of domain is inherent from the worklet type, but the size of the domain is dependent on the data being operated on.

A worklet identifies the argument specifying the domain with a type alias named InputDomain. The InputDomain must be aliased to one of the execution signature numeric tags (i.e. _1, _2, etc.). By default, the InputDomain points to the first argument, but a worklet can override that to point to any argument.

Example 17.4: An InputDomain declaration.

```
using InputDomain = _1;
```

Different types of worklets can have different types of domain. For example a simple field map worklet has a FieldIn argument as its input domain, and the size of the input domain is taken from the size of the associated field array. Likewise, a worklet that maps topology has a CellSetIn argument as its input domain, and the size of the input domain is taken from the cell set.

Specifying the InputDomain is optional. If it is not specified, the first argument is assumed to be the input domain.

17.4 Worklet Operator

A worklet is fundamentally a functor that operates on an element of data. Thus, the algorithm that the worklet represents is contained in or called from the parenthesis operator method.

Example 17.5: An overloaded parenthesis operator of a worklet.

```
template<typename T>
VTKM_EXEC void operator()(const T& psi, T& nsm) const
{
```

There are some constraints on the parenthesis operator. First, it must have the same arity as the ExecutionSignature, and the types of the parameters and return must be compatible. Second, because it runs in the execution environment, it must be declared with the VTKM_EXEC (or VTKM_EXEC_CONT) modifier. Third, the method must be declared const to help preserve thread safety.

17.5 Invoking a Worklet

Previously in this chapter we discussed creating a simple worklet. In this section we describe how to run the worklet in parallel.

A worklet is run using the vtkm::cont::Invoker class.

Example 17.6: Invoking a worklet.

```
vtkm::cont::ArrayHandle<vtkm::FloatDefault> psiArray;
// Fill psiArray with values...

```

```
4   vtkm::cont::Invoker invoke;
5
6   vtkm::cont::ArrayHandle<vtkm::FloatDefault> nsmArray;
7   invoke(PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, psiArray, nsmArray);
```

Using an `Invoker` is simple. First, an `Invoker` can be simply constructed with no arguments (line 4). Next, the `Invoker` is called as if it were a function (line 7).

The first argument to the invoke is always an instance of the worklet. The remaining arguments are data that are passed (indirectly) to the worklet. Each of these arguments (after the worklet) match a corresponding argument listed in the `ControlSignature`. So in the invocation on Example 17.6, line 7, the second and third arguments correspond the the two `ControlSignature` arguments given in Example 17.2. `psiArray` corresponds to the `FieldIn` argument and `nmsArray` corresponds to the `FieldOut` argument.

17.6 Preview of More Complex Worklets

This chapter demonstrates the creation of a worklet that performs a very simple math operation in parallel. However, we have just scratched the surface of the kinds of algorithms that can be expressed with VTK-m worklets. There are many more execution patterns and data handling constructs. The following example gives a preview of some of the more advanced features of worklets.

Example 17.7: A more complex worklet.

```
struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
  using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag,
           typename PointIndexVecType,
           typename EdgeIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::IdComponent edgeIndex,
                            EdgeIndexVecType& edgeIndices) const
  {
```

We will discuss the many features available in the worklet framework throughout Part IV.

CHAPTER

EIGHTEEN

BASIC FILTER IMPLEMENTATION

Chapter 17 introduced the concept of a worklet and demonstrated how to create and one run to execute an algorithm on a device. Although worklets provide a powerful mechanism for designing heavily threaded visualization algorithms, invoking them requires quite a bit of knowledge of the workings of VTK-m. Instead, most users execute algorithms in VTK-m using filters. Thus, to expose algorithms implemented with worklets to general users, we need to implement a filter to encapsulate the worklets. In this chapter we will create a filter that encapsulate the worklet algorithm presented in Chapter 17, which converted the units of a pressure field from pounds per square inch (psi) to Newtons per square meter ($\mathrm{N/m^2}$).

Filters in VTK-m are implemented by deriving one of the filter base classes provided in `vtkm::filter` . There are multiple base filter classes that we can choose from. These different classes are documented later in Chapter 22. For this example we will use the `vtkm::filter::FilterField` base class, which is used to derive one field from another field.

The following example shows the declaration of our pressure unit conversion filter. By convention, this declaration would be placed in a header file with a `.h` extension.

Example 18.1: Header declaration for a simple filter.

```
namespace vtkm
{
namespace filter
{

class PoundsPerSquareInchToNewtonsPerSquareMeterFilter
  : public vtkm::filter::FilterField<
      PoundsPerSquareInchToNewtonsPerSquareMeterFilter>
{
public:
  VTKM_CONT PoundsPerSquareInchToNewtonsPerSquareMeterFilter();

  template<typename ArrayHandleType, typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet,
    const ArrayHandleType& inField,
    const vtkm::filter::FieldMetadata& fieldMetadata,
    vtkm::filter::PolicyBase<Policy>);
};

}
} // namespace vtkm::filter
```

It should be noted that the `FilterField` superclass is actually a templated class that expects to be templated by the concrete subclass (in this case the `PoundsPerSquareInchToNewtonsPerSquareMeterFilter` class we are creating). This allows templated methods in the superclass call templated methods in the concrete subclass.

This technique of templating a superclass based on the subclass is known as the Curiously Recurring Template Pattern (CRTP).

It is typical for a filter to have a constructor to set up its initial state. A filter will also implement a DoExecute method. Different filter types require different arguments for DoExecute, but the FilterField version has 4 arguments: the input DataSet, an ArrayHandle holding the data for the input field, metadata about the input field, and a policy used to pull other information from the input DataSet. (We will not need to use the policy in this example as the superclass will give us all the data we need. Policies are demonstrated in Chapter 22.)

Once the filter class is declared in the .h file, the implementation filter is by convention given in a separate .hxx file. Given the definition of our filter in Example 18.1, we will need to provide the implementation for the constructor and the DoExecute method. The constructor is quite simple. It initializes the name of the output field name, which is managed by the superclass.

Example 18.2: Constructor for a simple filter.

```
VTKM_CONT vtkm::filter::PoundsPerSquareInchToNewtonsPerSquareMeterFilter::
  PoundsPerSquareInchToNewtonsPerSquareMeterFilter()
{
  this->SetOutputFieldName("");
}
```

In this case, we are setting the output field name to the empty string. This is not to mean that the default name of the output field should be the empty string, which is not a good idea. Rather, as we will see later, we will use the empty string to flag an output name that should be derived from the input name.

The meat of the filter implementation is located in the DoExecute method.

Example 18.3: Implementation of DoExecute for a simple filter.

```
template<typename ArrayHandleType, typename Policy>
VTKM_CONT vtkm::cont::DataSet
vtkm::filter::PoundsPerSquareInchToNewtonsPerSquareMeterFilter::DoExecute(
  const vtkm::cont::DataSet& inDataSet,
  const ArrayHandleType& inField,
  const vtkm::filter::FieldMetadata& fieldMetadata,
  vtkm::filter::PolicyBase<Policy>)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);

  using ValueType = typename ArrayHandleType::ValueType;

  vtkm::cont::ArrayHandle<ValueType> outField;

  this->Invoke(
    PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, inField, outField);

  std::string outFieldName = this->GetOutputFieldName();
  if (outFieldName == "")
  {
    outFieldName = fieldMetadata.GetName() + "_N/m^2";
  }

  return vtkm::filter::CreateResult(
    inDataSet, outField, outFieldName, fieldMetadata);
}
```

The second argument to DoExecute is a vtkm::cont::ArrayHandle that contains the data for the input field. This argument is templated because, as described in Chapter 16, ArrayHandles are templated classes that can change based on the types of values and how they are stored. The superclass has determined what these types are and has passed a concrete version of the ArrayHandle to DoExecute. It is good practice to check that the type we get is in fact an `ArrayHandle` using the `VTKM_IS_ARRAY_HANDLE` macro (demonstrated in line 9 of Example 18.3).

The first operation our filter does is to create a new `ArrayHandle` to store the new computed field (line 13). The type of the computed output field is typically derived from the type of the input array.

Once the filter has both the input and output `ArrayHandle`s, it is ready to invoke the worklet to compute the output. For convenience, all filter superclasses provide an `Invoke` member that is a `vtkm::cont::Invoker` class. Thus, worklets can be invoked within a filter by calling `this->Invoke` as if it were a method. This is shown starting on line 15. Recall that the worklet created in Chapter 17 takes two parameters: an input array and an output array, which are shown in this invocation.

With the output data created, the filter has to build the output structure to return. All implementations of `DoExecute` must return a `vtkm::cont::DataSet`, and for a simple field filter like this we want to return the same `DataSet` as the input with the output field added. The output field needs a name, and we get the appropriate name from the superclass (line 18). However, we would like a special case where if the user does not specify an output field name we construct one based on the input field name. Recall from Example 18.2 that by default we set the output field name to the empty string. Thus, our filter checks for this empty string, and if it is encountered, it builds a field name by appending "_N/M$\hat{2}$" to it.

Finally, our filter constructs the output `DataSet` using the `vtkm::filter::CreateResult` helper function. There are multiple versions of `CreateResult`, but the version we are using is adding our new array as a field as the same type as the input.

This chapter has just provided a brief introduction to creating filters. There are several more filter superclasses to help express algorithms of different types. After some more worklet concepts to implement more complex algorithms are introduced in Part IV, we will see a more complete documentation of the types of filters in Chapter 22.

Part IV

Advanced Development

CHAPTER

NINETEEN

ADVANCED TYPES

Chapter 4 introduced some of the base data types defined for use in VTK-m. However, for simplicity Chapter 4 just briefly touched the high-level concepts of these types. In this chapter we dive into much greater depth and introduce several more types.

19.1 Single Number Types

As described in Chapter 4, VTK-m provides aliases for all the base C types to ensure the representation matches the variable use. When a specific type width is not required, then the most common types to use are `vtkm::FloatDefault` for floating-point numbers, `vtkm::Id` for array and similar indices, and `vtkm::IdComponent` for shorter-width vector indices.

If a specific type width is desired, then one of the following is used to clearly declare the type and width.

bytes	floating point	signed integer	unsigned integer
1		`vtkm::Int8`	`vtkm::UInt8`
2		`vtkm::Int16`	`vtkm::UInt16`
4	`vtkm::Float32`	`vtkm::Int32`	`vtkm::UInt32`
8	`vtkm::Float64`	`vtkm::Int64`	`vtkm::UInt64`

These VTK-m–defined types should be preferred over basic C types like `int` or `float`.

19.2 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides the `vtkm::Vec <T,Size>` templated type, which is essentially a fixed length array of a given type.

The default constructor of `vtkm::Vec` objects leaves the values uninitialized. All vectors have a constructor with one argument that is used to initialize all components. All `vtkm::Vec` objects also have a constructor that allows you to set the individual components (one per argument). All `vtkm::Vec` objects with a size that is greater than 4 are constructed at run time and support an arbitrary number of initial values. Likewise, there is a `vtkm::make_Vec` convenience function that builds initialized vector types with an arbitrary number of components. Once created, you can use the bracket operator to get and set component values with the same syntax as an array.

Example 19.1: Creating vector types.

```
vtkm::Vec3f_32 A{ 1 };                         // A is (1, 1, 1)
A[1] = 2;                                      // A is now (1, 2, 1)
vtkm::Vec3f_32 B{ 1, 2, 3 };                   // B is (1, 2, 3)
vtkm::Vec3f_32 C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
// Longer Vecs specified with template.
vtkm::Vec<vtkm::Float32, 5> D{ 1 };                    // D is (1, 1, 1, 1, 1)
vtkm::Vec<vtkm::Float32, 5> E{ 1, 2, 3, 4, 5 };       // E is (1, 2, 3, 4, 5)
vtkm::Vec<vtkm::Float32, 5> F = { 6, 7, 8, 9, 10 }; // F is (6, 7, 8, 9, 10)
auto G = vtkm::make_Vec(1, 3, 5, 7, 9);               // G is (1, 3, 5, 7, 9)
```

The types vtkm::Id2, vtkm::Id3, and vtkm::Id4 are type aliases of vtkm::Vec <vtkm::Id,2>, vtkm::Vec <vtkm::Id,3>, and vtkm::Vec <vtkm::Id,4>. These are used to index arrays of 2, 3, and 4 dimensions, which is common. Likewise, vtkm::IdComponent2, vtkm::IdComponent4, and vtkm::IdComponent4 are type aliases of vtkm::Vec <vtkm::IdComponent,2>, vtkm::Vec <vtkm::IdComponent,3>, and vtkm::Vec <vtkm::IdComponent,4>.

Because declaring vtkm::Vec <T,Size> with all of its template parameters can be cumbersome, VTK-m provides easy to use aliases for small vectors of base types. As introduced in Section 4.3, the following type aliases are available.

bytes	size	floating point	signed integer	unsigned integer
default	2	vtkm::Vec2f	vtkm::Vec2i	vtkm::Vec2ui
	3	vtkm::Vec3f	vtkm::Vec3i	vtkm::Vec3ui
	4	vtkm::Vec4f	vtkm::Vec4i	vtkm::Vec4ui
1	2		vtkm::Vec2i_8	vtkm::Vec2ui_8
	3		vtkm::Vec3i_8	vtkm::Vec3ui_8
	4		vtkm::Vec4i_8	vtkm::Vec4ui_8
2	2		vtkm::Vec2i_16	vtkm::Vec2ui_16
	3		vtkm::Vec3i_16	vtkm::Vec3ui_16
	4		vtkm::Vec4i_16	vtkm::Vec4ui_16
4	2	vtkm::Vec2f_32	vtkm::Vec2i_32	vtkm::Vec2ui_32
	3	vtkm::Vec3f_32	vtkm::Vec3i_32	vtkm::Vec3ui_32
	4	vtkm::Vec4f_32	vtkm::Vec4i_32	vtkm::Vec4ui_32
8	2	vtkm::Vec2f_64	vtkm::Vec2i_64	vtkm::Vec2ui_64
	3	vtkm::Vec3f_64	vtkm::Vec3i_64	vtkm::Vec3ui_64
	4	vtkm::Vec4f_64	vtkm::Vec4i_64	vtkm::Vec4ui_64

vtkm::Vec supports component-wise arithmetic using the operators for plus (+), minus (-), multiply (*), and divide (/). It also supports scalar to vector multiplication with the multiply operator. The comparison operators equal (==) is true if every pair of corresponding components are true and not equal (!=) is true otherwise. A special vtkm::Dot function is overloaded to provide a dot product for every type of vector.

Example 19.2: Vector operations.

```
vtkm::Vec3f_32 A{ 1, 2, 3 };
vtkm::Vec3f_32 B{ 4, 5, 6.5 };
vtkm::Vec3f_32 C = A + B;                   // C is (5, 7, 9.5)
vtkm::Vec3f_32 D = 2.0f * C;                // D is (10, 14, 19)
vtkm::Float32 s = vtkm::Dot(A, B);          // s is 33.5
bool b1 = (A == B);                         // b1 is false
bool b2 = (A == vtkm::make_Vec(1, 2, 3)); // b2 is true

vtkm::Vec<vtkm::Float32, 5> E{ 1, 2.5, 3, 4, 5 };     // E is (1, 2, 3, 4, 5)
vtkm::Vec<vtkm::Float32, 5> F{ 6, 7, 8.5, 9, 10.5 }; // F is (6, 7, 8, 9, 10)
vtkm::Vec<vtkm::Float32, 5> G = E + F; // G is (7, 9.5, 11.5, 13, 15.5)
bool b3 = (E == F);                      // b3 is false
bool b4 = (G == vtkm::make_Vec(7.f, 9.5f, 11.5f, 13.f, 15.5f)); // b4 is true
```

These operators, of course, only work if they are also defined for the component type of the vtkm::Vec. For example, the multiply operator will work fine on objects of type vtkm::Vec <char,3>, but the multiply operator will not work on objects of type vtkm::Vec <std::string,3> because you cannot multiply objects of type std::string.

In addition to generalizing vector operations and making arbitrarily long vectors, vtkm::Vec can be repurposed for creating any sequence of homogeneous objects. Here is a simple example of using vtkm::Vec to hold the state of a polygon.

Example 19.3: Repurposing a vtkm::Vec.

```
  vtkm::Vec<vtkm::Vec2f_32, 3> equilateralTriangle = { { 0.0f, 0.0f },
                                                       { 1.0f, 0.0f },
                                                       { 0.5f, 0.8660254f } };
```

The vtkm::Vec class provides a convenient structure for holding and passing small vectors of data. However, there are times when using Vec is inconvenient or inappropriate. For example, the size of vtkm::Vec must be known at compile time, but there may be need for a vector whose size is unknown until compile time. Also, the data populating a vtkm::Vec might come from a source that makes it inconvenient or less efficient to construct a vtkm::Vec. For this reason, VTK-m also provides several Vec-*like* objects that behave much like vtkm::Vec but are a different class. These Vec-like objects have the same interface as vtkm::Vec except that the NUM_COMPONENTS constant is not available on those that are sized at run time. Vec-like objects also come with a CopyInto method that will take their contents and copy them into a standard Vec class. (The standard Vec class also has a CopyInto method for consistency.)

The first Vec-like object is vtkm::VecC, which exposes a C-type array as a Vec. The constructor for vtkm::VecC takes a C array and a size of that array. There is also a constant version of VecC named vtkm::VecCConst, which takes a constant array and cannot be mutated. The vtkm/Types.h header defines both VecC and VecCConst as well as multiple versions of vtkm::make_VecC to easily convert a C array to either a VecC or VecCConst.

The following example demonstrates converting values from a constant table into a vtkm::VecCConst for further consumption. The table and associated methods define how 8 points come together to form a hexahedron.

Example 19.4: Using vtkm::VecCConst with a constant array.

```
VTKM_EXEC
vtkm::VecCConst<vtkm::IdComponent> HexagonIndexToIJK(vtkm::IdComponent index)
{
  static const vtkm::IdComponent HexagonIndexToIJKTable[8][3] = {
    { 0, 0, 0 }, { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 0 },
    { 0, 0, 1 }, { 1, 0, 1 }, { 1, 1, 1 }, { 0, 1, 1 }
  };

  return vtkm::make_VecC(HexagonIndexToIJKTable[index], 3);
}

VTKM_EXEC
vtkm::IdComponent HexagonIJKToIndex(vtkm::VecCConst<vtkm::IdComponent> ijk)
{
  static const vtkm::IdComponent HexagonIJKToIndexTable[2][2][2] = {
    {
      // i=0
      { 0, 4 }, // j=0
      { 3, 7 }, // j=1
    },
    {
      // i=1
      { 1, 5 }, // j=0
      { 2, 6 }, // j=1
    }
  };

  return HexagonIJKToIndexTable[ijk[0]][ijk[1]][ijk[2]];
}
```

Common Errors

The vtkm::VecC *and* vtkm::VecCConst *classes only hold a pointer to a buffer that contains the data. They do not manage the memory holding the data. Thus, if the pointer given to* vtkm::VecC *or* vtkm::VecCConst *becomes invalid, then using the object becomes invalid. Make sure that the scope of the* vtkm::VecC *or* vtkm::VecCConst *does not outlive the scope of the data it points to.*

The next Vec-like object is vtkm::VecVariable, which provides a Vec-like object that can be resized at run time to a maximum value. Unlike VecC, VecVariable holds its own memory, which makes it a bit safer to use. But also unlike VecC, you must define the maximum size of VecVariable at compile time. Thus, VecVariable is really only appropriate to use when there is a predetermined limit to the vector size that is fairly small.

The following example uses a vtkm::VecVariable to store the trace of edges within a hexahedron. This example uses the methods defined in Example 19.4.

Example 19.5: Using vtkm::VecVariable.

```
vtkm::VecVariable<vtkm::IdComponent, 4> HexagonShortestPath(
  vtkm::IdComponent startPoint,
  vtkm::IdComponent endPoint)
{
  vtkm::VecCConst<vtkm::IdComponent> startIJK = HexagonIndexToIJK(startPoint);
  vtkm::VecCConst<vtkm::IdComponent> endIJK = HexagonIndexToIJK(endPoint);

  vtkm::IdComponent3 currentIJK;
  startIJK.CopyInto(currentIJK);

  vtkm::VecVariable<vtkm::IdComponent, 4> path;
  path.Append(startPoint);
  for (vtkm::IdComponent dimension = 0; dimension < 3; dimension++)
  {
    if (currentIJK[dimension] != endIJK[dimension])
    {
      currentIJK[dimension] = endIJK[dimension];
      path.Append(HexagonIJKToIndex(currentIJK));
    }
  }

  return path;
}
```

VTK-m provides further examples of Vec-like objects as well. For example, the vtkm::VecFromPortal and vtkm::VecFromPortalPermute objects allow you to treat a subsection of an arbitrarily large array as a Vec. These objects work by attaching to array portals, which are described in Section 27.1. Another example of a Vec-like object is vtkm::VecRectilinearPointCoordinates, which efficiently represents the point coordinates in an axis-aligned hexahedron. Such shapes are common in structured grids. These and other data sets are described in Chapter 7.

19.3 Pair

VTK-m defines a vtkm::Pair <T1,T2> templated object that behaves just like std::pair from the standard template library. The difference is that vtkm::Pair will work in both the execution and control environment, whereas the STL std::pair does not always work in the execution environment.

The VTK-m version of vtkm::Pair supports the same types, fields, and operations as the STL version. VTK-m also provides a vtkm::make_Pair function for convenience.

19.4 Range

VTK-m provides a convenience structure named vtkm::Range to help manage a range of values. The Range struct contains two data members, Min and Max, which represent the ends of the range of numbers. Min and Max are both of type vtkm::Float64. Min and Max can be directly accessed, but Range also comes with the following helper functions to make it easier to build and use ranges. Note that all of these functions treat the minimum and maximum value as inclusive to the range.

IsNonEmpty Returns true if the range covers at least one value.

Contains Takes a single number and returns true if that number is contained within the range.

Length Returns the distance between Min and Max. Empty ranges return a length of 0. Note that if the range is non-empty and the length is 0, then Min and Max must be equal, and the range contains exactly one number.

Center Returns the number equidistant to Min and Max. If the range is empty, NaN is returned.

Include Takes either a single number or another range and modifies this range to include the given number or range. If necessary, the range is grown just enough to encompass the given argument. If the argument is already in the range, nothing changes.

Union A nondestructive version of Include, which builds a new Range that is the union of this range and the argument. The + operator is also overloaded to compute the union.

The following example demonstrates the operation of vtkm::Range.

Example 19.6: Using vtkm::Range.

```
vtkm::Range range;           // default constructor is empty range
bool b1 = range.IsNonEmpty(); // b1 is false

range.Include(0.5);            // range now is [0.5 .. 0.5]
bool b2 = range.IsNonEmpty();  // b2 is true
bool b3 = range.Contains(0.5); // b3 is true
bool b4 = range.Contains(0.6); // b4 is false

range.Include(2.0);            // range is now [0.5 .. 2]
bool b5 = range.Contains(0.5); // b3 is true
bool b6 = range.Contains(0.6); // b4 is true

range.Include(vtkm::Range(-1, 1)); // range is now [-1 .. 2]

range.Include(vtkm::Range(3, 4)); // range is now [-1 .. 4]

vtkm::Float64 lower = range.Min;       // lower is -1
vtkm::Float64 upper = range.Max;       // upper is 4
```

```
vtkm::Float64 length = range.Length(); // length is 5
vtkm::Float64 center = range.Center(); // center is 1.5
```

19.5 Bounds

VTK-m provides a convenience structure named vtkm::Bounds to help manage an axis-aligned region in 3D space. Among other things, this structure is often useful for representing a bounding box for geometry. The Bounds struct contains three data members, X, Y, and Z, which represent the range of the bounds along each respective axis. All three of these members are of type vtkm::Range, which is discussed previously in Section 19.4. X, Y, and Z can be directly accessed, but Bounds also comes with the following helper functions to make it easier to build and use ranges.

IsNonEmpty Returns true if the bounds cover at least one value.

Contains Takes a vtkm::Vec of size 3 and returns true if those point coordinates are contained within the range.

Center Returns the point at the center of the range as a vtkm::Vec <vtkm::Float64,3>.

Include Takes either a vtkm::Vec of size 3 or another bounds and modifies this bounds to include the given point or bounds. If necessary, the bounds are grown just enough to encompass the given argument. If the argument is already in the bounds, nothing changes.

Union A nondestructive version of Include, which builds a new Bounds that is the union of this bounds and the argument. The + operator is also overloaded to compute the union.

The following example demonstrates the operation of vtkm::Bounds.

Example 19.7: Using vtkm::Bounds.

```
vtkm::Bounds bounds;           // default constructor makes empty
bool b1 = bounds.IsNonEmpty(); // b1 is false

bounds.Include(vtkm::make_Vec(0.5, 2.0, 0.0));            // bounds contains only
                                                          // the point [0.5, 2, 0]
bool b2 = bounds.IsNonEmpty();                            // b2 is true
bool b3 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b3 is true
bool b4 = bounds.Contains(vtkm::make_Vec(1, 1, 1));       // b4 is false
bool b5 = bounds.Contains(vtkm::make_Vec(0, 0, 0));       // b5 is false

bounds.Include(vtkm::make_Vec(4, -1, 2)); // bounds is region [0.5 .. 4] in X,
                                          //                  [-1 .. 2] in Y,
                                          //              and [0 .. 2] in Z
bool b6 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b6 is true
bool b7 = bounds.Contains(vtkm::make_Vec(1, 1, 1));       // b7 is true
bool b8 = bounds.Contains(vtkm::make_Vec(0, 0, 0));       // b8 is false

vtkm::Bounds otherBounds(vtkm::make_Vec(0, 0, 0), vtkm::make_Vec(3, 3, 3));
// otherBounds is region [0 .. 3] in X, Y, and Z
bounds.Include(otherBounds); // bounds is now region [0 .. 4] in X,
                             //                      [-1 .. 3] in Y,
                             //                  and [0 .. 3] in Z

vtkm::Vec3f_64 lower(bounds.X.Min, bounds.Y.Min, bounds.Z.Min);
// lower is [0, -1, 0]
vtkm::Vec3f_64 upper(bounds.X.Max, bounds.Y.Max, bounds.Z.Max);
// upper is [4, 3, 3]

vtkm::Vec3f_64 center = bounds.Center(); // center is [2, 1, 1.5]
```

19.6 Traits

When using templated types, it is often necessary to get information about the type or specialize code based on general properties of the type. VTK-m uses *traits* classes to publish and retrieve information about types. A traits class is simply a templated structure that provides type aliases for tag structures, empty types used for identification. The traits classes might also contain constant numbers and helpful static functions. See *Effective C++ Third Edition* by Scott Mayers for a description of traits classes and their uses.

19.6.1 Type Traits

The vtkm::TypeTraits <T> templated class provides basic information about a core type. These type traits are available for all the basic C++ types as well as the core VTK-m types described in Chapter 4. vtkm::TypeTraits contains the following elements.

NumericTag This type is set to either vtkm::TypeTraitsRealTag or vtkm::TypeTraitsIntegerTag to signal that the type represents either floating point numbers or integers.

DimensionalityTag This type is set to either vtkm::TypeTraitsScalarTag or vtkm::TypeTraitsVectorTag to signal that the type represents either a single scalar value or a tuple of values.

ZeroInitialization A static member function that takes no arguments and returns 0 (or the closest equivalent to it) cast to the type.

The definition of vtkm::TypeTraits for vtkm::Float32 could like something like this.

Example 19.8: Definition of vtkm::TypeTraits <vtkm::Float32 >.

```cpp
namespace vtkm {

template<>
struct TypeTraits<vtkm::Float32>
{
  using NumericTag = vtkm::TypeTraitsRealTag;
  using DimensionalityTag = vtkm::TypeTraitsScalarTag;

  VTKM_EXEC_CONT
  static vtkm::Float32 ZeroInitialization() { return vtkm::Float32(0); }
};

}
```

Here is a simple example of using vtkm::TypeTraits to implement a generic function that behaves like the remainder operator (%) for all types including floating points and vectors.

Example 19.9: Using TypeTraits for a generic remainder.

```cpp
#include <vtkm/TypeTraits.h>

#include <vtkm/Math.h>

template<typename T>
T AnyRemainder(const T& numerator, const T& denominator);

namespace detail
{

template<typename T>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   vtkm::TypeTraitsIntegerTag,
                   vtkm::TypeTraitsScalarTag)
{
  return numerator % denominator;
}

template<typename T>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   vtkm::TypeTraitsRealTag,
                   vtkm::TypeTraitsScalarTag)
{
  // The VTK-m math library contains a Remainder function that operates on
  // floating point numbers.
  return vtkm::Remainder(numerator, denominator);
}

template<typename T, typename NumericTag>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   NumericTag,
                   vtkm::TypeTraitsVectorTag)
{
  T result;
  for (int componentIndex = 0; componentIndex < T::NUM_COMPONENTS; componentIndex++)
  {
    result[componentIndex] =
      AnyRemainder(numerator[componentIndex], denominator[componentIndex]);
  }
  return result;
}

} // namespace detail

template<typename T>
T AnyRemainder(const T& numerator, const T& denominator)
{
  return detail::AnyRemainderImpl(numerator,
                                  denominator,
                                  typename vtkm::TypeTraits<T>::NumericTag(),
                                  typename vtkm::TypeTraits<T>::DimensionalityTag());
}
```

19.6.2 Vector Traits

The templated `vtkm::Vec` class contains several items for introspection (such as the component type and its size). However, there are other types that behave similarly to `Vec` objects but have different ways to perform this introspection.

For example, VTK-m contains `Vec`-like objects that essentially behave the same but might have different features. Also, there may be reason to interchangeably use basic scalar values, like an integer or floating point number, with vectors. To provide a consistent interface to access these multiple types that represents vectors, the `vtkm::VecTraits <T>` templated class provides information and accessors to vector types.It contains the following elements.

`ComponentType` This type is set to the type for each component in the vector. For example, a `vtkm::Id3` has `ComponentType` defined as `vtkm::Id`.

IsSizeStatic This type is set to either vtkm::VecTraitsTagSizeStatic if the vector has a static number of components that can be determined at compile time or set to vtkm::VecTraitsTagSizeVariable if the size of the vector is determined at run time. If IsSizeStatic is set to VecTraitsTagSizeVariable, then VecTraits will be missing some information that cannot be determined at compile time.

HasMultipleComponents This type is set to either vtkm::VecTraitsTagSingleComponent if the vector length is size 1 or vtkm::VecTraitsTagMultipleComponents otherwise. This tag can be useful for creating specialized functions when a vector is really just a scalar. If the vector type is of variable size (that is, IsSizeStatic is VecTraitsTagSizeVariable), then HasMultipleComponents might be VecTraitsTagMultipleComponents even when at run time there is only one component.

NUM_COMPONENTS An integer specifying how many components are contained in the vector. NUM_COMPONENTS is not available for vector types of variable size (that is, IsSizeStatic is VecTraitsTagSizeVariable).

GetNumberOfComponents A static method that takes an instance of a vector and returns the number of components the vector contains. The result of GetNumberOfComponents is the same value of NUM_COMPONENTS for vector types that have a static size (that is, IsSizeStatic is VecTraitsTagSizeStatic). But unlike NUM_COMPONENTS, GetNumberOfComponents works for vectors of any type.

GetComponent A static method that takes a vector and returns a particular component.

SetComponent A static method that takes a vector and sets a particular component to a given value.

CopyInto A static method that copies the components of a vector to a vtkm::Vec.

The definition of vtkm::VecTraits for vtkm::Id3 could look something like this.

Example 19.10: Definition of vtkm::VecTraits <vtkm::Id3 >.

```
namespace vtkm {

template<>
struct VecTraits<vtkm::Id3>
{
  using ComponentType = vtkm::Id;
  static const int NUM_COMPONENTS = 3;
  using IsSizeStatic = vtkm::VecTraitsTagSizeStatic;
  using HasMultipleComponents = vtkm::VecTraitsTagMultipleComponents;

  VTKM_EXEC_CONT
  static vtkm::IdComponent GetNumberOfComponents(const vtkm::Id3&)
  {
    return NUM_COMPONENTS;
  }

  VTKM_EXEC_CONT
  static const vtkm::Id& GetComponent(const vtkm::Id3& vector, int component)
  {
    return vector[component];
  }
  VTKM_EXEC_CONT
  static vtkm::Id& GetComponent(vtkm::Id3& vector, int component)
  {
    return vector[component];
  }

  VTKM_EXEC_CONT
  static void SetComponent(vtkm::Id3& vector, int component, vtkm::Id value)
  {
    vector[component] = value;
  }

  template<vtkm::IdComponent DestSize>
  VTKM_EXEC_CONT static void CopyInto(const vtkm::Id3& src,
                                      vtkm::Vec<vtkm::Id, DestSize>& dest)
  {
    for (vtkm::IdComponent index = 0; (index < NUM_COMPONENTS) && (index < DestSize);
         index++)
    {
      dest[index] = src[index];
    }
  }
};

} // namespace vtkm
```

The real power of vector traits is that they simplify creating generic operations on any type that can look like a vector. This includes operations on scalar values as if they were vectors of size one. The following code uses vector traits to simplify the implementation of less functors that define an ordering that can be used for sorting and other operations.

Example 19.11: Using VecTraits for less functors.

```
#include <vtkm/VecTraits.h>

// This functor provides a total ordering of vectors. Every compared vector
// will be either less, greater, or equal (assuming all the vector components
// also have a total ordering).
template<typename T>
struct LessTotalOrder
{
  VTKM_EXEC_CONT
  bool operator()(const T& left, const T& right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
      const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType& rightValue =
        vtkm::VecTraits<T>::GetComponent(right, index);
      if (leftValue < rightValue)
      {
        return true;
      }
      if (rightValue < leftValue)
      {
        return false;
      }
    }
    // If we are here, the vectors are equal (or at least equivalent).
    return false;
  }
};

// This functor provides a partial ordering of vectors. It returns true if and
// only if all components satisfy the less operation. It is possible for
// vectors to be neither less, greater, nor equal, but the transitive closure
// is still valid.
template<typename T>
struct LessPartialOrder
{
  VTKM_EXEC_CONT
  bool operator()(const T& left, const T& right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
      const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType& rightValue =
        vtkm::VecTraits<T>::GetComponent(right, index);
      if (!(leftValue < rightValue))
      {
        return false;
      }
    }
    // If we are here, all components satisfy less than relation.
    return true;
  }
};
```

19.7 List Tags

VTK-m internally uses template metaprogramming, which utilizes C++ templates to run source-generating programs, to customize code to various data and compute platforms. One basic structure often uses with template metaprogramming is a list of class names (also sometimes called a tuple or vector, although both of those names have different meanings in VTK-m).

Many VTK-m users only need predefined lists, such as the type lists specified in Section 19.7.2. Those users can skip most of the details of this section. However, it is sometimes useful to modify lists, create new lists, or operate on lists, and these usages are documented here.

VTK-m uses a tag-based mechanism for defining lists, which differs significantly from lists in many other template metaprogramming libraries such as with `boost::mpl::vector` or `boost::vector`. Rather than enumerating all list entries as template arguments, the list is referenced by a single tag class with a descriptive name. The intention is to make fully resolved types shorter and more readable. (Anyone experienced with template programming knows how insanely long and unreadable types can get in compiler errors and warnings.)

19.7.1 Building List Tags

List tags are constructed in VTK-m by defining a `struct` that publicly inherits from another list tags. The base list tags are defined in the vtkm/ListTag.h header.

The most basic list is defined with `vtkm::ListTagEmpty`. This tag represents an empty list.

`vtkm::ListTagBase <T, ...>` represents a list of the types given as template parameters. `vtkm::ListTagBase` supports a variable number of parameters with the maximum specified by `VTKM_MAX_BASE_LIST`.

Finally, lists can be combined together with `vtkm::ListTagJoin <ListTag1,ListTag2>`, which concatinates two lists together.

The following example demonstrates how to build list tags using these base lists classes. Note first that all the list tags are defined as `struct` rather than `class`. Although these are roughly synonymous in C++, `struct` inheritance is by default public, and public inheritance is important for the list tags to work. Note second that these tags are created by inheritance rather than using a type alias. Although a type alias defined with `using` will work, it will lead to much uglier type names defined by the compiler.

Example 19.12: Creating list tags.

```
#include <vtkm/ListTag.h>

// Placeholder classes representing things that might be in a template
// metaprogram list.
class Foo;
class Bar;
class Baz;
class Qux;
class Xyzzy;

// The names of the following tags are indicative of the lists they contain.

struct FooList : vtkm::ListTagBase<Foo>
{
};

struct FooBarList : vtkm::ListTagBase<Foo, Bar>
{
};

struct BazQuxXyzzyList : vtkm::ListTagBase<Baz, Qux, Xyzzy>
{
};

struct QuxBazBarFooList : vtkm::ListTagBase<Qux, Baz, Bar, Foo>
{
};

struct FooBarBazQuxXyzzyList : vtkm::ListTagJoin<FooBarList, BazQuxXyzzyList>
{
};
```

19.7.2 Type Lists

One of the major use cases for template metaprogramming lists in VTK-m is to identify a set of potential data types for arrays. The vtkm/TypeListTag.h header contains predefined lists for known VTK-m types. Although technically all these lists are of C++ types, the types we refer to here are those data types stored in data arrays. The following lists are provided.

`vtkm::TypeListTagId` Contains the single item `vtkm::Id`.

`vtkm::TypeListTagId2` Contains the single item `vtkm::Id2`.

`vtkm::TypeListTagId3` Contains the single item `vtkm::Id3`.

`vtkm::TypeListTagIndex` A list of all types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`.

`vtkm::TypeListTagFieldScalar` A list containing types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`).

`vtkm::TypeListTagFieldVec2` A list containing types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec3` A list containing types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec4` A list containing types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers.

vtkm::TypeListTagField A list containing all the types generally used for fields. It is the combination of vtkm::TypeListTagFieldScalar, vtkm::TypeListTagFieldVec2, vtkm::TypeListTagFieldVec3, and vtkm::TypeListTagFieldVec4.

vtkm::TypeListTagScalarAll A list of all scalar types. It contains signed and unsigned integers of widths from 8 to 64 bits. It also contains floats of 32 and 64 bit widths.

vtkm::TypeListTagVecCommon A list of the most common vector types. It contains all vtkm::Vec class of size 2 through 4 containing components of unsigned bytes, signed 32-bit integers, signed 64-bit integers, 32-bit floats, or 64-bit floats.

vtkm::TypeListTagVecAll A list of all vtkm::Vec classes with standard integers or floating points as components and lengths between 2 and 4.

vtkm::TypeListTagAll A list of all types included in vtkm/Types.h with vtkm::Vec s with up to 4 components.

vtkm::TypeListTagCommon A list containing only the most used types in visualization. This includes signed integers and floats that are 32 or 64 bit. It also includes 3 dimensional vectors of floats. This is the default list used when resolving the type in variant arrays (described in Chapter 33).

If these lists are not sufficient, it is possible to build new type lists using the existing type lists and the list bases from Section 19.7.1 as demonstrated in the following example.

Example 19.13: Defining new type lists.

```
#define VTKM_DEFAULT_TYPE_LIST_TAG MyCommonTypes

#include <vtkm/ListTag.h>
#include <vtkm/TypeListTag.h>

// A list of 2D vector types.
struct Vec2List : vtkm::ListTagBase<vtkm::Id2, vtkm::Vec2f_32, vtkm::Vec2f_64>
{
};

// An application that uses 2D geometry might commonly encounter this list of
// types.
struct MyCommonTypes : vtkm::ListTagJoin<Vec2List, vtkm::TypeListTagCommon>
{
};
```

The vtkm/TypeListTag.h header also defines a macro named VTKM_DEFAULT_TYPE_LIST_TAG that defines a default list of types to use in classes like vtkm::cont::VariantArrayHandle (Chapter 33). This list can be overridden by defining the VTKM_DEFAULT_TYPE_LIST_TAG macro *before* any VTK-m headers are included. If included after a VTK-m header, the list is not likely to take effect. Do not ignore compiler warnings about the macro being redefined, which you will not get if defined correctly. Example 19.13 also contains an example of overriding the VTKM_DEFAULT_TYPE_LIST_TAG macro.

19.7.3 Operating on Lists

VTK-m template metaprogramming lists are typically just passed to VTK-m methods that internally operate on the lists. Although not typically used outside of the VTK-m library, these operations are also available.

The vtkm/ListTag.h header comes with a vtkm::ListForEach function that takes a functor object and a list tag. It then calls the functor object with the default object of each type in the list. This is most typically used with C++ run-time type information to convert a run-time polymorphic object to a statically typed (and possibly inlined) call.

The following example shows a rudimentary version of coverting a dynamically-typed array to a statically-typed array similar to what is done in VTK-m classes like vtkm::cont::VariantArrayHandle (which is documented in Chapter 33).

Example 19.14: Converting dynamic types to static types with ListForEach.

```cpp
struct MyArrayBase
{
  // A virtual destructor makes sure C++ RTTI will be generated. It also helps
  // ensure subclass destructors are called.
  virtual ~MyArrayBase() {}
};

template<typename T>
struct MyArrayImpl : public MyArrayBase
{
  std::vector<T> Array;
};

template<typename T>
void PrefixSum(std::vector<T>& array)
{
  T sum(typename vtkm::VecTraits<T>::ComponentType(0));
  for (typename std::vector<T>::iterator iter = array.begin(); iter != array.end();
       iter++)
  {
    sum = sum + *iter;
    *iter = sum;
  }
}

struct PrefixSumFunctor
{
  MyArrayBase* ArrayPointer;

  PrefixSumFunctor(MyArrayBase* arrayPointer)
    : ArrayPointer(arrayPointer)
  {
  }

  template<typename T>
  void operator()(T)
  {
    using ConcreteArrayType = MyArrayImpl<T>;
    ConcreteArrayType* concreteArray =
      dynamic_cast<ConcreteArrayType*>(this->ArrayPointer);
    if (concreteArray != NULL)
    {
      PrefixSum(concreteArray->Array);
    }
  }
};

void DoPrefixSum(MyArrayBase* array)
{
  PrefixSumFunctor functor = PrefixSumFunctor(array);
  vtkm::ListForEach(functor, vtkm::TypeListTagCommon());
}
```

CHAPTER

TWENTY

LOGGING

VTK-m features a logging system that allows status updates and timing. VTK-m uses the loguru project to provide runtime logging facilities.[1] Logging is enabled by setting the CMake variable VTKm_ENABLE_LOGGING. When this flag is enabled, any messages logged to the Info, Warn, Error, and Fatal levels are printed to stderr by default.

20.1 Initializing Logging

Additional logging features are enabled by calling `vtkm::cont::Initialize` as described in Chapter 6. Although calling `Initialize` is not strictly necessary for output messages, initialization adds the following features.

- Set human-readable names for the log levels in the output.
- Allow the stderr logging level to be set at runtime by passing a `-v [level]` argument to the executable (if provided).
- Name the main thread.
- Print a preamble with details of the program's startup (arguments, etc).
- Install signal handlers to automatically print stack traces and error contexts (Linux only) on crashes.

Example 20.1 in the following section provides an example of initializing with additional logging setup.

The logging implementation is thread-safe. When working in a multithreaded environment, each thread may be assigned a human-readable name using `vtkm::cont::SetThreadName` (which can later be retrieved with `vtkm::cont::GetThreadName`. This name will appear in the log output so that per-thread messages can be easily tracked.

20.2 Logging Levels

The logging in VTK-m provides several "levels" of logging. Logging levels are ordered by precedence. When selecting which log message to output, a single logging level is provided. Any logging message with that or a higher precedence is output. For example, if warning messages are on, then error messages are also outputted

[1] A sample of the log output can be found at https://gitlab.kitware.com/snippets/427.

because errors are a higher precedence than warnings. Likewise, if information messages are on, then error and warning messages are also outputted.

Common Errors

All logging levels are assigned a number, and logging levels with a higher precedence actually have a smaller number.

All logging levels are listed in the `vtkm::cont::LogLevel` enum. The available logging levels, in order of precedence, are as follows.

`LogLevel::Off` A placeholder used to silence all logging.

`LogLevel::Fatal` Fatal errors that should abort execution.

`LogLevel::Error` Important but non-fatal errors, such as device fail-over.

`LogLevel::Warn` Less important user errors, such as out-of-bounds parameters.

`LogLevel::Info` Information messages (detected hardware, etc) and temporary debugging output.

`LogLevel::UserFirst` The first in a range of logging levels reserved for code that uses VTK-m. Internal VTK-m code will not log on these levels but will report these logs.

`LogLevel::UserLast` The last in a range of logging levels reserved for code that uses VTK-m.

`LogLevel::Perf` General timing data and algorithm flow information, such as filter execution, worklet dispatches, and device algorithm calls.

`LogLevel::MemCont` Host-side resource memory allocations and frees such as `ArrayHandle` control buffers.

`LogLevel::MemExec` Device-side resource memory allocations and frees such as `ArrayHandle` device buffers)

`LogLevel::MemTransfer` Transferring of data between a host and device.

`LogLevel::Cast` Report when a dynamic object is (or is not) resolved via a `CastAndCall` or other casting method.

`LogLevel::UserVerboseFirst` The first in a range of logging levels reserved for code that uses VTK-m. Internal VTK-m code will not log on these levels but will report these logs. These are used similarly to those in the `UserFirst` range but are at a lower precedence that also includes more verbose reporting from VTK-m.

`LogLevel::UserVerboseLast` The last in a range of logging levels reserved for code that uses VTK-m.

When VTK-m outputs an entry in its log, it annotates the message with the logging level. VTK-m will automatically provide descriptions for all log levels described in `vtkm::cont::LogLevel`. A custom log level can be described by calling the `vtkm::cont::SetLogLevelName` function. (The log name can likewise be retrieved with `vtkm::cont::GetLogLevelName`.)

Common Errors

The SetLogLevelName *function must be called before* vtkm::cont::Initialize *to have an effect.*

Common Errors

The descriptions for each log level are only set up if vtkm::cont::Initialize *is called. If it is not, then all log levels will be represented with a numerical value.*

If vtkm::cont::Initialize is called with argc/argv, then the user can control the logging level with the "-v" command line argument. Alternatively, you can control which logging levels are reported with the vtkm::cont::SetStderrLogLevel.

Example 20.1: Initializing logging.

```
static const vtkm::cont::LogLevel CustomLogLevel = vtkm::cont::LogLevel::UserFirst;

int main(int argc, char** argv)
{
  vtkm::cont::SetLogLevelName(CustomLogLevel, "custom");

  // For this example we will set the log level manually.
  // The user can override this with the -v command line flag.
  vtkm::cont::SetStderrLogLevel(CustomLogLevel);

  vtkm::cont::Initialize(argc, argv);

  // Do interesting stuff...
```

20.3 Log Entries

Log entries are created with a collection of macros provided in vtkm/cont/Logging.h. In addition to basic log entries, VTK-m logging can also provide conditional logging, scope levels of logs, and generate special logs on crashes.

20.3.1 Basic Log Entries

The main logging entry points are the macros VTKM_LOG_S and VTKM_LOG_F, which use C++ stream and printf syntax, respectively. Both macros take a logging level as the first argument. The remaining arguments specify the message printed to the log. VTKM_LOG_S takes a single argument with a C++ stream expression (so << operators can exist in the expression). VTKM_LOG_F takes a C string as its second argument that has printf-style formatting codes. The remaining arguments fulfill those codes.

Example 20.2: Basic logging.

```
  VTKM_LOG_F(vtkm::cont::LogLevel::Info,
             "Base VTK-m version: %d.%d",
             VTKM_VERSION_MAJOR,
             VTKM_VERSION_MINOR);
```

```
VTKM_LOG_S(vtkm::cont::LogLevel::Info,
           "Full VTK-m version: " << VTKM_VERSION_FULL);
```

20.3.2 Conditional Log Entries

The macros VTKM_LOG_IF_S VTKM_LOG_IF_F behave similarly to VTKM_LOG_S and VTKM_LOG_F, respectively, except they have an extra argument that contains the condition. If the condition is true, then the log entry is created. If the condition is false, then the statement is ignored and nothing is recorded in the log.

Example 20.3: Conditional logging.

```
for (size_t i = 0; i < 5; i++)
{
  VTKM_LOG_IF_S(
    vtkm::cont::LogLevel::Info, i % 2 == 0, "Found an even number: " << i);
}
```

20.3.3 Scoped Log Entries

The logging back end supports the concept of scopes. Scopes allow the nesting of log messages, which allows a complex operation to report when it starts, when it ends, and what log messages happen in the middle. Scoped log entries are also timed so you can get an idea of how long operations take. Scoping can happen to arbitrary depths.

Common Errors

Although the timing reported in scoped log entries can give an idea of the time each operation takes, the reported time should not be considered accurate in regards to timing parallel operations. If a parallel algorithm is invoked inside a log scope, the program may return from that scope before the parallel algorithm is complete. See Chapter 13 for information on more accurate timers.

Scoped log entries follow the same scoping of your C++ code. A scoped log can be created with the VTKM_LOG_SCOPE macro. This macro behaves similarly to VTKM_LOG_F except that it creates a scoped log that starts when VTKM_LOG_SCOPE and ends when the program leaves the given scope.

Example 20.4: Scoped logging.

```
for (vtkm::IdComponent trial = 0; trial < numTrials; ++trial)
{
  VTKM_LOG_SCOPE(CustomLogLevel, "Trial %d", trial);

  VTKM_LOG_F(CustomLogLevel, "Do thing 1");

  VTKM_LOG_F(CustomLogLevel, "Do thing 2");

  //...
}
```

It is also common, and typically good code structure, to structure scoped concepts around functions or methods. Thus, VTK-m provides VTKM_LOG_SCOPE_FUNCTION. When placed at the beginning of a function or macro, VTKM_LOG_SCOPE_FUNCTION will automatically create a scoped log around it.

Example 20.5: Scoped logging in a function.

```
void TestFunc()
{
  VTKM_LOG_SCOPE_FUNCTION(vtkm::cont::LogLevel::Info);
  VTKM_LOG_S(vtkm::cont::LogLevel::Info, "Showcasing function logging");
}
```

20.3.4 Error Context

The VTK-m logging is capable of capturing some crashes and writing information to the log before the program terminates. The VTKM_LOG_ERROR_CONTEXT can be used to record some information that should be reported if an error occurs. If the program terminates successfully, then information is never recorded to the log.

Example 20.6: Providing an error context for logging.

```
  // This message is only logged if a crash occurs
  VTKM_LOG_ERROR_CONTEXT("Some variable value", 42);
```

20.4 Helper Functions

The vtkm/cont/Logging.h header file also contains several helper functions that provide useful functions when reporting information about the system.

Did you know?

Although provided with the logging utilities, these functions can be useful in contexts outside of the logging as well. These functions are available even if VTK-m is compiled with logging off.

The vtkm::cont::TypeToString function provides run-time type information (RTTI) based type-name information. TypeToString is a templated function for which you have to explicitly declare the type. TypeToString returns a std::string containing a representation of the type provided. When logging is enabled, TypeToString uses the logging back end to demangle symbol names on supported platforms.

The vtkm::cont::GetHumanReadableSize takes a size of memory in bytes and returns a human readable string (for example ”64 bytes”, ”1.44 MiB”, ”128 GiB”, etc). vtkm::cont::GetSizeString is a similar function that returns the same thing as GetHumanReadableSize followed by “(# bytes)” (with # replaced with the number passed to the function). Both GetHumanReadableSize and GetSizeString take an optional second argument that is the number of digits of precision to display. By default, they display 2 digits of precision.

The vtkm::cont::GetStackTrace function returns a string containing a trace of the stack, which can be helpful for debugging. GetStackTrace takes an optional argument for the number of stack frames to skip. Reporting the stack trace is not available on all platforms. On platforms that are not supported, a simple string reporting that the stack trace is unavailable is returned.

Example 20.7: Helper log functions.

```
template<typename T>
void DoSomething(T&& x)
{
  VTKM_LOG_S(CustomLogLevel,
             "Doing something with type " << vtkm::cont::TypeToString<T>());

  vtkm::Id arraySize = 100000 * sizeof(T);
  VTKM_LOG_S(CustomLogLevel,
             "Size of array is " << vtkm::cont::GetHumanReadableSize(arraySize));
  VTKM_LOG_S(CustomLogLevel,
             "More precisely it is " << vtkm::cont::GetSizeString(arraySize, 4));

  VTKM_LOG_S(CustomLogLevel, "Stack location: " << vtkm::cont::GetStackTrace());
```

CHAPTER

TWENTYONE

WORKLET TYPE REFERENCE

Chapter 17 introduces worklets and provides a simple example of creating a worklet to run an algorithm on a many core device. Different operations in visualization can have different data access patterns, perform different execution flow, and require different provisions. VTK-m manages these different accesses, execution, and provisions by grouping visualization algorithms into common classes of operation and supporting each class with its own worklet type.

Each worklet type has a generic superclass that worklets of that particular type must inherit. This makes the type of the worklet easy to identify. The following list describes each worklet type provided by VTK-m and the superclass that supports it.

Field Map A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over a mesh, a `WorkletMapField` may actually be applied to any array. Thus, a field map can be used as a basic map operation.

Topology Map A worklet deriving `vtkm::worklet::WorkletMapTopology` or one of its child classes performs a mapping operation that applies a function (the operator in the worklet) on all elements of a particular type (such as points or cells) and creates a new field for those elements. The basic operation is similar to a field map except that in addition to access fields being mapped on, the worklet operation also has access to incident fields.

There are multiple convenience classes available for the most common types of topology mapping. `vtkm::worklet::WorkletVisitCellsWithPoints` calls the worklet operation for each cell and makes every incident point available. This type of map also has access to cell structures and can interpolate point fields. Likewise, `vtkm::worklet::WorkletVisitPointsWithCells` calls the worklet operation for each point and makes every incident cell available.

Point Neighborhood A worklet deriving from `vtkm::worklet::WorkletPointNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all points of a structured mesh. The basic operation is similar to a field map except that in addition to having access to the point being operated on, you can get the field values of nearby points within a neighborhood of a given size. Point neighborhood worklets can only applied to structured cell sets.

Reduce by Key A worklet deriving `vtkm::worklet::WorkletReduceByKey` operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a `Vec`-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

The remainder of this chapter provides details on how to create worklets of each type. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 41.

21.1 Field Map

A worklet deriving vtkm::worklet::WorkletMapField performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over the mesh, a WorkletMapField can actually be applied to any array.

A field map worklet supports the following tags in the parameters of its ControlSignature.

FieldIn This tag represents an input field. A FieldIn argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array.

The worklet's InputDomain can be set to a FieldIn argument. In this case, the input domain will be the size of the array.

FieldOut This tag represents an output field. A FieldOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldInOut This tag represents field that is both an input and an output. A FieldInOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

The worklet's InputDomain can be set to a FieldInOut argument. In this case, the input domain will be the size of the array.

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 227.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 233.

A field map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 31.1).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 21.1: Implementation and use of a field map worklet.

```
namespace vtkm
{
namespace worklet
{

class ComputeMagnitude : public vtkm::worklet::WorkletMapField
{
public:
  using ControlSignature = void(FieldIn inputVectors, FieldOut outputMagnitudes);
  using ExecutionSignature = _2(_1);

  using InputDomain = _1;

  template<typename T, vtkm::IdComponent Size>
  VTKM_EXEC T operator()(const vtkm::Vec<T, Size>& inVector) const
  {
    return vtkm::Magnitude(inVector);
  }
};

} // namespace worklet
} // namespace vtkm
```

Although simple, the WorkletMapField worklet type can be used (and abused) as a general parallel-for/scheduling mechanism. In particular, the WorkIndex execution signature tag can be used to get a unique index, the WholeArray* tags can be used to get random access to arrays, and the ExecObject control signature tag can be used to pass execution objects directly to the worklet. Whole arrays and execution objects are talked about in more detail in Chapters 28 and 29, respectively, in more detail, but here is a simple example that uses the random access of WholeArrayOut to make a worklet that copies an array in reverse order.

Example 21.2: Leveraging field maps and field maps for general processing.

```
namespace vtkm
{
namespace worklet
{

struct ReverseArrayCopyWorklet : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn inputArray, WholeArrayOut outputArray);
  using ExecutionSignature = void(_1, _2, WorkIndex);
  using InputDomain = _1;

  template<typename InputType, typename OutputArrayPortalType>
  VTKM_EXEC void operator()(const InputType& inputValue,
                            const OutputArrayPortalType& outputArrayPortal,
                            vtkm::Id workIndex) const
  {
    vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
    if (outIndex >= 0)
    {
      outputArrayPortal.Set(outIndex, inputValue);
    }
    else
    {
      this->RaiseError("Output array not sized correctly.");
    }
  }
};

} // namespace worklet
} // namespace vtkm
```

21.2 Topology Map

A topology map performs a mapping that it applies a function (the operator in the worklet) on all the elements of a DataSet of a particular type (i.e. point, edge, face, or cell). While operating on the element, the worklet has access to data from all incident elements of another type.

There are several versions of topology maps that differ in what type of element being mapped from and what type of element being mapped to. The subsequent sections describe these different variations of the topology maps.

21.2.1 Visit Cells with Points

A worklet deriving vtkm::worklet::WorkletVisitCellsWithPoints performs a mapping operation that applies a function (the operator in the worklet) on all the cells of a DataSet. While operating on the cell, the worklet has access to fields associated both with the cell and with all incident points. Additionally, the worklet can get information about the structure of the cell and can perform operations like interpolation on it.

A visit cells with points worklet supports the following tags in the parameters of its ControlSignature.

CellSetIn This tag represents the cell set that defines the collection of cells the map will operate on. A CellSetIn argument expects a CellSet subclass or a DynamicCellSet in the associated parameter of the Invoker. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.)

There must be exactly one CellSetIn argument, and the worklet's InputDomain must be set to this argument.

FieldInPoint This tag represents an input field that is associated with the points. A FieldInPoint argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of points.

Each invocation of the worklet gets a Vec-like object containing the field values for all the points incident with the cell being visited. The order of the entries is consistent with the defined order of the vertices for the visited cell's shape. If the field is a vector field, then the provided object is a Vec of Vecs.

FieldInCell This tag represents an input field that is associated with the cells. A FieldInCell argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

FieldOutCell This tag represents an output field, which is necessarily associated with cells. A FieldOutCell argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOut is an alias for FieldOutCell (since output arrays can only be defined on cells).

FieldInOutCell This tag represents field that is both an input and an output, which is necessarily associated with cells. A FieldInOutCell argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOut is an alias for FieldInOutCell (since output arrays can only be defined on cells).

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker.

A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 227.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 233.

A visit cells with points worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,… These reference the corresponding parameter in the ControlSignature.

CellShape This tag produces a shape tag corresponding to the shape of the visited cell. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.) This is the same value that gets provided if you reference the CellSetIn parameter.

PointCount This tag produces a vtkm::IdComponent equal to the number of points incident on the cell being visited. The Vecs provided from a FieldInPoint parameter will be the same size as PointCount.

PointIndices This tag produces a Vec-like object of vtkm::Id s giving the indices for all incident points. Like values from a FieldInPoint parameter, the order of the entries is consistent with the defined order of the vertices for the visited cell's shape.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 31.1).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

Point to cell field maps are a powerful construct that allow you to interpolate point fields throughout the space of the data set. See Chapter 25 for a description on how to work with the cell information provided to the worklet. The following example provides a simple demonstration that finds the geometric center of each cell by interpolating the point coordinates to the cell centers.

Example 21.3: Implementation and use of a visit cells with points worklet.

```
namespace vtkm
{
namespace worklet
{

struct CellCenter : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet,
                                FieldInPoint inputPointField,
                                FieldOut outputCellField);
  using ExecutionSignature = _3(_1, PointCount, _2);

  using InputDomain = _1;

  template<typename CellShape, typename InputPointFieldType>
  VTKM_EXEC typename InputPointFieldType::ComponentType operator()(
    CellShape shape,
    vtkm::IdComponent numPoints,
    const InputPointFieldType& inputPointField) const
  {
    vtkm::Vec3f parametricCenter =
      vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, *this);
    return vtkm::exec::CellInterpolate(
      inputPointField, parametricCenter, shape, *this);
  }
};

} // namespace worklet
} // namespace vtkm
```

21.2.2 Visit Points with Cells

A worklet deriving `vtkm::worklet::WorkletVisitPointsWithCells` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `DataSet`. While operating on the point, the worklet has access to fields associated both with the point and with all incident cells.

A visit points with cells worklet supports the following tags in the parameters of its `ControlSignature`.

`CellSetIn` This tag represents the cell set that defines the collection of points the map will operate on. A `CellSetIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the `Invoker`.

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or a `VariantArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of cells.

Each invocation of the worklet gets a `Vec`-like object containing the field values for all the cells incident with the point being visited. The order of the entries is arbitrary but will be consistent with the values of all other `FieldInCell` arguments for the same worklet invocation. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or a `VariantArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of points. Each invocation of the worklet gets a single value out of this array.

FieldOutPoint This tag represents an output field, which is necessarily associated with points. A FieldOutPoint argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOut is an alias for FieldOutPoint (since output arrays can only be defined on points).

FieldInOutPoint This tag represents field that is both an input and an output, which is necessarily associated with points. A FieldInOutPoint argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOut is an alias for FieldInOutPoint (since output arrays can only be defined on points).

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 227.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 233.

A visit points with cells worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

`CellCount` This tag produces a `vtkm::IdComponent` equal to the number of cells incident on the point being visited. The Vecs provided from a `FieldInCell` parameter will be the same size as `CellCount`.

`CellIndices` This tag produces a `Vec`-like object of `vtkm::Id` s giving the indices for all incident cells. The order of the entries is arbitrary but will be consistent with the values of all other `FieldInCell` arguments for the same worklet invocation.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

Cell to point field maps are typically used for converting fields associated with cells to points so that they can be interpolated. The following example does a simple averaging, but you can also implement other strategies such as a volume weighted average.

Example 21.4: Implementation and use of a visit points with cells worklet.

```
namespace vtkm
{
namespace worklet
{

class AverageCellField : public vtkm::worklet::WorkletVisitPointsWithCells
{
public:
  using ControlSignature = void(CellSetIn cellSet,
                                FieldInCell inputCellField,
                                FieldOut outputPointField);
  using ExecutionSignature = void(CellCount, _2, _3);

  using InputDomain = _1;

  template<typename InputCellFieldType, typename OutputFieldType>
  VTKM_EXEC void operator()(vtkm::IdComponent numCells,
                            const InputCellFieldType& inputCellField,
                            OutputFieldType& fieldAverage) const
  {
    fieldAverage = OutputFieldType(0);

    for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
    {
      fieldAverage = fieldAverage + inputCellField[cellIndex];
    }

    fieldAverage = fieldAverage / OutputFieldType(numCells);
  }
};

} // namespace worklet
} // namespace vtkm

//
// Later in the associated Filter class...
//

        vtkm::cont::ArrayHandle<T> outFieldData;
        this->Invoke(
          vtkm::worklet::AverageCellField{}, inCellSet, inFieldData, outFieldData);
```

21.2.3 General Topology Maps

A worklet deriving vtkm::worklet::WorkletMapTopology performs a mapping operation that applies a function (the operator in the worklet) on all the elements of a specified type from a DataSet. While operating on each element, the worklet has access to fields associated both with that element and with all incident elements of a different specified type.

The WorkletMapTopology class is a template with two template parameters. The first template parameter specifies the "visit" topology element, and the second parameter specifies the "incident" topology element. The worklet is scheduled such that each instance is associated with a particular "visit" topology element and has access to "incident" topology elements.

These visit and incident topology elements are specified with topology element tags, which are defined in the vtkm/TopologyElementTag.h header file. The available topology element tags are vtkm::TopologyElementTagCell, vtkm::TopologyElementTagPoint, vtkm::TopologyElementTagEdge, and vtkm::TopologyElementTagFace, which represent the cell, point, edge, and face elements, respectively.

WorkletMapTopology is a generic form of a topology map, and it can perform identically to the aforementioned forms of topology map with the correct template parameters. For example,

```
vtkm::worklet::WorkletMapTopology <vtkm::TopologyElementTagCell, vtkm::TopologyElementTagPoint >
```

is equivalent to the vtkm::worklet::WorkletVisitCellsWithPoints class except the signature tags have different names. The names used in the specific topology map superclasses (such as WorkletVisitCellsWithPoints) tend to be easier to read and are thus preferable. However, the generic WorkletMapTopology is available for topology combinations without a specific superclass or to support more general mappings in a worklet.

The general topology map worklet supports the following tags in the parameters of its ControlSignature, which are equivalent to tags in the other topology maps but with different (more general) names.

CellSetIn This tag represents the cell set that defines the collection of elements the map will operate on. A CellSetIn argument expects a CellSet subclass or a DynamicCellSet in the associated parameter of the Invoker. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.)

There must be exactly one CellSetIn argument, and the worklet's InputDomain must be set to this argument.

FieldInVisit This tag represents an input field that is associated with the "visit" element. A FieldInVisit argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

FieldInIncident This tag represents an input field that is associated with the "incident" elements. A FieldInIncident argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of "incident" elements.

Each invocation of the worklet gets a Vec-like object containing the field values for all the "incident" elements incident with the "visit" element being visited. If the field is a vector field, then the provided object is a Vec of Vecs.

FieldOut This tag represents an output field, which is necessarily associated with "visit" elements. A FieldOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldInOut This tag represents field that is both an input and an output, which is necessarily associated with "visit" elements. A FieldInOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 227.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 233.

A general topology map worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

CellShape This tag produces a shape tag corresponding to the shape of the visited element. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.) This is the same value that gets provided if you reference the CellSetIn parameter.

If the "visit" element is cells, the CellShape clearly will match the shape of each cell. Other elements will have shapes to match their structures. Points have vertex shapes, edges have line shapes, and faces have some type of polygonal shape.

IncidentElementCount This tag produces a vtkm::IdComponent equal to the number of elements incident on the element being visited. The Vecs provided from a FieldInIncident parameter will be the same size as IncidentElementCount.

IncidentElementIndices This tag produces a Vec-like object of vtkm::Id s giving the indices for all incident elements. The order of the entries is consistent with the values of all other FieldInIncident arguments for the same worklet invocation.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 31.1).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

21.3 Point Neighborhood

A worklet deriving vtkm::worklet::WorkletPointNeighborhood performs a mapping operation that applies a function (the operator in the worklet) on all the points of a DataSet. While operating on the point, the worklet has access to field values on nearby points within a neighborhood.

A point neighborhood worklet supports the following tags in the parameters of its ControlSignature.

CellSetIn This tag represents the cell set that defines the collection of points the map will operate on. A CellSetIn argument expects a vtkm::cont::CellSetStructured object in the associated parameter of the Invoker. The object could also be stored in a DynamicCellSet, but it is an error to use any object other than CellSetStructured.

There must be exactly one CellSetIn argument, and the worklet's InputDomain must be set to this argument.

FieldIn This tag represents an input field that is associated with the points. A FieldIn argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of points. Each invocation of the worklet gets a single value out of this array.

FieldInNeighborhood This tag represents an input field that is associated with the points. A FieldInNeighborhood argument expects an ArrayHandle or a VariantArrayHandle in the Invoker. The size of the array must be exactly the number of points.

What differentiates FieldInNeighborhood from FieldIn is that FieldInNeighborhood allows the worklet function to access the field value at the point it is visiting and the field values in the neighborhood around it. Thus, instead of getting a single value out of the array, each invocation of the worklet gets a vtkm::exec::FieldNeighborhood object. FieldNeighborhood objects are described in the Neighborhood Information section starting on page 160.

FieldOut This tag represents an output field, which is necessarily associated with points. A FieldOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldInOut This tag represents field that is both an input and an output, which is necessarily associated with points. A FieldInOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 225.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 227.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 233.

A point neighborhood worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

Boundary This tag produces a vtkm::exec::arg::BoundaryState object, which provides information about where the local neighborhood is in relationship to the full mesh. BoundaryState objects are described in the Neighborhood Information section starting on page 160.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 31.1).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

21.3.1 Neighborhood Information

As stated earlier in this section, what makes a WorkletPointNeighborhood worklet special is its ability to get field information in a neighborhood surrounding a point rather than just the point itself. This is done using the special FieldInNeighborhood ControlSignature tag. When you use this tag, rather than getting the single field value for the point, you get a vtkm::exec::FieldNeighborhood object.

The FieldNeighborhood class (which has a single template argument of the array portal type values are stored in) contains a Get method that retrieves a field value relative to the local neighborhood. FieldNeighborhood::Get takes the *i*, *j*, *k* index of the point with respect to the local point. So, calling FieldNeighborhood::Get(0,0,0) retrieves at the point being visited. Likewise, Get(-1,0,0) gets the value to the "left" of the point visited and Get(1,0,0) gets the value to the "right." FieldNeighborhood::Get is overloaded to accept the index as either three separate vtkm::IdComponent values or a single vtkm::Vec <vtkm::IdComponent,3>.

Example 21.5: Retrieve neighborhood field value.

```
            sum = sum + inputField.Get(i, j, k);
```

When performing operations on a neighborhood within the mesh, it is often important to know whether the expected neighborhood is contained completely within the mesh or whether the neighborhood extends beyond the borders of the mesh. This can be queried using a vtkm::exec::BoundaryState object, which is provided when a Boundary tag is listed in the ExecutionSignature.

Generally, BoundaryState allows you to specify the size of the neighborhood at runtime. The neighborhood size is specified by a *radius*. The radius specifies the number of items in each direction the neighborhood extends. So, for example, a point neighborhood with radius 1 would contain a $3 \times 3 \times 3$ neighborhood centered around the point. Likewise, a point neighborhood with radius 2 would contain a $5 \times 5 \times 5$ neighborhood centered around the point. BoundaryState provides several methods to determine if the neighborhood is contained in the mesh.

MinNeighborIndices Given a radius for the neighborhood, returns a vtkm::Vec <vtkm::IdComponent,3> for the "lower left" (minimum) index. If the visited point is in the middle of the mesh, the returned triplet is the negative radius for all components. But if the visited point is near the mesh boundary, then the minimum index will be clipped.

For example, if the visited point is at $[5,5,5]$ and `MinNeighborIndices(2)` is called, then $[-2,-2,-2]$ is returned. However, if the visited point is at $[0,1,2]$ and `MinNeighborIndices(2)` is called, then $[0,-1,-2]$ is returned.

`MaxNeighborIndices` Given a radius for the neighborhood, returns a `vtkm::Vec <vtkm::IdComponent,3>` for the "upper right" (maximum) index. If the visited point is in the middle of the mesh, the returned triplet is the negative radius for all components. But if the visited point is near the mesh boundary, then the maximum index will be clipped.

For example, if the visited point is at $[5,5,5]$ in a 10^3 mesh and `MaxNeighborIndices(2)` is called, then $[2,2,2]$ is returned. However, if the visited point is at $[7,8,9]$ in the same mesh and `MaxNeighborIndices(2)` is called, then $[2,1,0]$ is returned.

`InBoundary` Given a radius for the neighborhood, returns true if the neighborhood is contained completely within the boundary of the mesh, false otherwise.

`InXBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative x (I) direction, true otherwise.

`InYBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative y (J) direction, true otherwise.

`InZBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative z (K) direction, true otherwise.

The `BoundaryState::MinNeighborIndices` and `BoundaryState::MaxNeighborIndices` are particularly useful for iterating over the valid portion of the neighborhood.

Example 21.6: Iterating over the valid portion of a neighborhood.

```
    auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
    auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);

    T sum = 0;
    vtkm::IdComponent size = 0;
    for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
    {
      for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)
      {
        for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
        {
          sum = sum + inputField.Get(i, j, k);
          ++size;
        }
      }
    }
```

21.3.2 Convolving Small Kernels

A common use case for point neighborhood worklets is to convolve a small kernel with a structured mesh. A very simple example of this is averaging out the values the values within some distance to the central point. This has the effect of smoothing out the field (although smoothing filters with better properties exist). The following example shows a worklet that applies this simple "box" averaging.

Example 21.7: Implementation and use of a point neighborhood worklet.

```
class ApplyBoxKernel : public vtkm::worklet::WorkletPointNeighborhood
{
```

```
private:
  vtkm::IdComponent NumberOfLayers;

public:
  using ControlSignature = void(CellSetIn cellSet,
                                FieldInNeighborhood inputField,
                                FieldOut outputField);
  using ExecutionSignature = _3(_2, Boundary);

  using InputDomain = _1;

  ApplyBoxKernel(vtkm::IdComponent kernelSize)
  {
    VTKM_ASSERT(kernelSize >= 3);
    VTKM_ASSERT((kernelSize % 2) == 1);

    this->NumberOfLayers = (kernelSize - 1) / 2;
  }

  template<typename InputFieldPortalType>
  VTKM_EXEC typename InputFieldPortalType::ValueType operator()(
    const vtkm::exec::FieldNeighborhood<InputFieldPortalType>& inputField,
    const vtkm::exec::BoundaryState& boundary) const
  {
    using T = typename InputFieldPortalType::ValueType;

    auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
    auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);

    T sum = 0;
    vtkm::IdComponent size = 0;
    for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
    {
      for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)
      {
        for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
        {
          sum = sum + inputField.Get(i, j, k);
          ++size;
        }
      }
    }

    return static_cast<T>(sum / size);
  }
};
```

21.4 Reduce by Key

A worklet deriving vtkm::worklet::WorkletReduceByKey operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a Vec-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

Figure 21.1 show a pictorial representation of how VTK-m collects data for a reduce by key worklet. All calls to a reduce by key worklet has exactly one array of keys. The key array in this example has 4 unique keys: 0, 1, 2, 4. These 4 unique keys will result in 4 calls to the worklet function. This example also has 2 arrays of values associated with the keys. (A reduce by keys worklet can have any number of values arrays.)

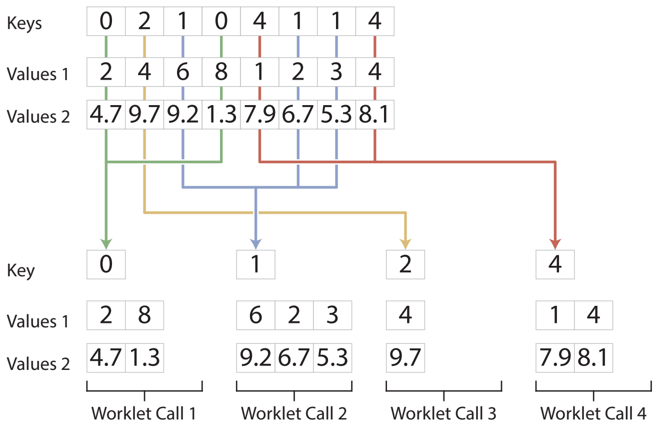


Figure 21.1: The collection of values for a reduce by key worklet.

When the worklet is invoked, all these common keys will be collected with their associated values. The parenthesis operator of the worklet will be called once per each unique key. The worklet call will be given a Vec-like containing all values that have the key.

A reduce by key worklet supports the following tags in the parameters of its ControlSignature.

KeysIn This tag represents the input keys. A KeysIn argument expects a vtkm::worklet::Keys object in the associated parameter of the Invoker. The Keys object, which wraps around an ArrayHandle containing the keys and manages the auxiliary structures for collecting like keys, is described later in this section.

Each invocation of the worklet gets a single unique key.

A WorkletReduceByKey object must have exactly one KeysIn parameter in its ControlSignature, and the InputDomain must point to the KeysIn parameter.

ValuesIn This tag represents a set of input values that are associated with the keys. A ValuesIn argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The number of values in this array must be equal to the size of the array used with the KeysIn argument. Each invocation of the worklet gets a Vec-like object containing all the values associated with the unique key.

ValuesInOut This tag behaves the same as ValuesIn except that the worklet may write values back into the Vec-like object, and these values will be placed back in their original locations in the array. Use of ValuesInOut is rare.

ValuesOut This tag behaves the same as ValuesInOut except that the array is resized appropriately and no input values are passed to the worklet. As with ValuesInOut, values the worklet writes to its Vec-like object get placed in the location of the original arrays. Use of ValuesOut is rare.

ReducedValuesOut This tag represents the resulting reduced values. A ReducedValuesOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

ReducedValuesIn This tag represents input values that come from (typically) from a previous invocation of a reduce by key. A ReducedValuesOut argument expects an ArrayHandle or a VariantArrayHandle in the associated parameter of the Invoker. The number of values in the array must equal the number of *unique* keys.

A ReducedValuesIn argument is usually used to pass reduced values from one invocation of a reduce by key worklet to another invocation of a reduced by key worklet such as in an algorithm that requires iterative steps.

ReducedValuesInOut This tag behaves the same as ReducedValuesIn except that the worklet may write values back into the array. Make sure that the associated parameter to the worklet operator is a reference so that the changed value gets written back to the array.

A reduce by key worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,… These reference the corresponding parameter in the ControlSignature.

ValueCount This tag produces a vtkm::IdComponent that is equal to the number of times the key associated with this call to the worklet occurs in the input. This is the same size as the Vec-like objects provided by ValuesIn arguments.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

InputIndex This tag produces a vtkm::Id that identifies the index of the input element, which can differ from the WorkIndex in a worklet with a scatter (as described in Section 31.1).

OutputIndex This tag produces a vtkm::Id that identifies the index of the output element. (This is generally the same as WorkIndex.)

ThreadIndices This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 41.2, but most users can get the information they need through other signature tags.

As stated earlier, the reduce by key worklet is useful for collected like values. To demonstrate the reduce by key worklet, we will create a simple mechanism to generate a histogram in parallel. (VTK-m comes with its own histogram implementation, but we create our own version here for a simple example.) The way we can use the reduce by key worklet to compute a histogram is to first identify which bin of the histogram each value is in, and then use the bin identifiers as the keys to collect the information. To help with this example, we will first create a helper class named BinScalars that helps us manage the bins.

Example 21.8: A helper class to manage histogram bins.

```
class BinScalars
{
public:
  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Id numBins)
    : Range(range)
    , NumBins(numBins)
  {
  }

  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Float64 tolerance)
    : Range(range)
  {
    this->NumBins = vtkm::Id(this->Range.Length() / tolerance) + 1;
  }

  VTKM_EXEC_CONT
  vtkm::Id GetBin(vtkm::Float64 value) const
  {
    vtkm::Float64 ratio = (value - this->Range.Min) / this->Range.Length();
    vtkm::Id bin = vtkm::Id(ratio * this->NumBins);
    bin = vtkm::Max(bin, vtkm::Id(0));
    bin = vtkm::Min(bin, this->NumBins - 1);
    return bin;
  }

private:
  vtkm::Range Range;
  vtkm::Id NumBins;
};
```

Using this helper class, we can easily create a simple map worklet that takes values, identifies a bin, and writes that result out to an array that can be used as keys.

Example 21.9: A simple map worklet to identify histogram bins, which will be used as keys.

```
  struct IdentifyBins : vtkm::worklet::WorkletMapField
  {
    using ControlSignature = void(FieldIn data, FieldOut bins);
    using ExecutionSignature = _2(_1);
    using InputDomain = _1;

    BinScalars Bins;

    VTKM_CONT
    IdentifyBins(const BinScalars& bins)
      : Bins(bins)
    {
    }

    VTKM_EXEC
    vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
  };
```

Once you generate an array to be used as keys, you need to make a vtkm::worklet::Keys object. The Keys object is what will be passed to the Invoker for the argument associated with the KeysIn ControlSignature tag. This of course happens in the control environment after calling the Invoker for our worklet for generating the keys.

Example 21.10: Creating a vtkm::worklet::Keys object.

```
    vtkm::cont::ArrayHandle<vtkm::Id> binIds;
    this->Invoke(IdentifyBins(bins), valuesArray, binIds);

    vtkm::worklet::Keys<vtkm::Id> keys(binIds);
```

Now that we have our keys, we are finally ready for our reduce by key worklet. A histogram is simply a count of the number of elements in a bin. In this case, we do not really need any values for the keys. We just need the size of the bin, which can be identified with the internally calculated ValueCount.

A complication we run into with this histogram filter is that it is possible for a bin to be empty. If a bin is empty, there will be no key associated with that bin, and the Invoker will not call the worklet for that bin/key. To manage this case, we have to initialize an array with 0's and then fill in the non-zero entities with our reduce by key worklet. We can find the appropriate entry into the array by using the key, which is actually the bin identifier, which doubles as an index into the histogram. The following example gives the implementation for the reduce by key worklet that fills in positive values of the histogram.

Example 21.11: A reduce by key worklet to write histogram bin counts.

```
struct CountBins : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys, WholeArrayOut binCounts);
  using ExecutionSignature = void(_1, ValueCount, _2);
  using InputDomain = _1;

  template<typename BinCountsPortalType>
  VTKM_EXEC void operator()(vtkm::Id binId,
                            vtkm::IdComponent numValuesInBin,
                            BinCountsPortalType& binCounts) const
  {
    binCounts.Set(binId, numValuesInBin);
  }
};
```

The previous example demonstrates the basic usage of the reduce by key worklet to count common keys. A more common use case is to collect values associated with those keys, do an operation on those values, and provide a "reduced" value for each unique key. The following example demonstrates such an operation by providing a worklet that finds the average of all values in a particular bin rather than counting them.

Example 21.12: A worklet that averages all values with a common key.

```
struct BinAverage : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys,
                                ValuesIn originalValues,
                                ReducedValuesOut averages);
  using ExecutionSignature = _3(_2);
  using InputDomain = _1;

  template<typename OriginalValuesVecType>
  VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
    const OriginalValuesVecType& originalValues) const
  {
    typename OriginalValuesVecType::ComponentType sum = 0;
    for (vtkm::IdComponent index = 0;
         index < originalValues.GetNumberOfComponents();
         index++)
    {
      sum = sum + originalValues[index];
    }
    return sum / originalValues.GetNumberOfComponents();
  }
};
```

To complete the code required to average all values that fall into the same bin, the following example shows the full code required to invoke such a worklet. Note that this example repeats much of the previous examples, but shows it in a more complete context.

Example 21.13: Using a reduce by key worklet to average values falling into the same bin.

```
struct IdentifyBins : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn data, FieldOut bins);
  using ExecutionSignature = _2(_1);
  using InputDomain = _1;

  BinScalars Bins;

  VTKM_CONT
  IdentifyBins(const BinScalars& bins)
    : Bins(bins)
  {
  }

  VTKM_EXEC
  vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
};

struct BinAverage : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys,
                                ValuesIn originalValues,
                                ReducedValuesOut averages);
  using ExecutionSignature = _3(_2);
  using InputDomain = _1;

  template<typename OriginalValuesVecType>
  VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
    const OriginalValuesVecType& originalValues) const
  {
    typename OriginalValuesVecType::ComponentType sum = 0;
    for (vtkm::IdComponent index = 0;
         index < originalValues.GetNumberOfComponents();
         index++)
    {
      sum = sum + originalValues[index];
    }
    return sum / originalValues.GetNumberOfComponents();
  }
};

//
// Later in the associated Filter class...
//

  vtkm::Range range =
    vtkm::cont::ArrayRangeCompute(inField).GetPortalConstControl().Get(0);
  BinScalars bins(range, numBins);

  vtkm::cont::ArrayHandle<vtkm::Id> binIds;
  this->Invoke(IdentifyBins(bins), inField, binIds);

  vtkm::worklet::Keys<vtkm::Id> keys(binIds);

  vtkm::cont::ArrayHandle<T> combinedValues;

  this->Invoke(BinAverage{}, keys, inField, combinedValues);
```

CHAPTER

TWENTYTWO

FILTER TYPE REFERENCE

In Chapters 17 and 21 we discuss how to implement an algorithm in the VTK-m framework by creating a worklet. Worklets might be straightforward to implement and invoke for those well familiar with the appropriate VTK-m API. However, novice users have difficulty using worklets directly. For simplicity, worklet algorithms are generally wrapped in what are called filter objects for general usage. Chapter 9 introduces the concept of filters and documents those that come with the VTK-m library. Chapter 18 gave a brief introduction on implementing filters. This chapter elaborates on building new filter objects by introducing new filter types. These will be used to wrap filters around the extended worklet examples in Chapter 21.

Unsurprisingly, the base filter objects are contained in the `vtkm::filter` package. The basic implementation of a filter involves subclassing one of the base filter objects and implementing the `DoExecute` method. The `DoExecute` method performs the operation of the filter and returns a new data set containing the result.

As with worklets, there are several flavors of filter types to address different operating behaviors although their is not a one-to-one relationship between worklet and filter types. This chapter is sectioned by the different filter types with an example of implementations for each.

22.1 Field Filters

Field filters are a category of filters that generate a new field. These new fields are typically derived from one or more existing fields or point coordinates on the data set. For example, mass, volume, and density are interrelated, and any one can be derived from the other two.

Field filters are implemented in classes that derive the `vtkm::filter::FilterField` base class. `FilterField` is a templated class that has a single template argument, which is the type of the concrete subclass. The propagation of additional fields from the input to the output is handled automatically by `vtkm::filter::FilterField`. If the default of automatic propagation isn't desired the caller can select which fields to propagate via `SetFieldsToPass`.

Did you know?

The convention of having a subclass be templated on the derived class' type is known as the Curiously Recurring Template Pattern (CRTP). In the case of `FilterField` and other filter base classes, VTK-m uses this CRTP behavior to allow the general implementation of these algorithms to run `DoExecute` in the subclass, which as we see in a moment is itself templated.

All `FilterField` subclasses must implement a `DoExecute` method. The `FilterField` base class implements an

Execute method (actually several overloaded versions of Execute), processes the arguments, and then calls the DoExecute method of its subclass. The DoExecute method has the following 4 arguments.

- An input data set contained in a vtkm::cont::DataSet object. (See Chapter 7 for details on DataSet objects.)
- The field from the DataSet specified in the Execute method to operate on. The field is always passed as an instance of vtkm::cont::ArrayHandle. (See Chapter 16 for details on ArrayHandle objects.) The type of the ArrayHandle is generally not known until the class is used and requires a template type.
- A vtkm::filter::FieldMetadata object that contains the associated metadata of the field not contained in the ArrayHandle of the second argument. The FieldMetadata contains information like the name of the field and what topological element the field is associated with (such as points or cells).
- A policy class. See Section 22.4 for information on using policies. The type of the policy is generally unknown until the class is used and requires a template type.

In this section we provide an example implementation of a field filter that wraps the "magnitude" worklet provided in Example 21.1 (listed on page 149). By convention, filter implementations are split into two files. The first file is a standard header file with a .h extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named FieldMagnitude.h.

Example 22.1: Header declaration for a field filter.

```
 1 namespace vtkm
 2 {
 3 namespace filter
 4 {
 5
 6 class FieldMagnitude : public vtkm::filter::FilterField<FieldMagnitude>
 7 {
 8 public:
 9   VTKM_CONT FieldMagnitude();
10
11   using SupportedTypes = vtkm::ListTagBase<vtkm::Vec2f_32,
12                                            vtkm::Vec2f_64,
13                                            vtkm::Vec3f_32,
14                                            vtkm::Vec3f_64,
15                                            vtkm::Vec4f_32,
16                                            vtkm::Vec4f_64>;
17
18   template<typename ArrayHandleType, typename Policy>
19   VTKM_CONT vtkm::cont::DataSet DoExecute(
20     const vtkm::cont::DataSet& inDataSet,
21     const ArrayHandleType& inField,
22     const vtkm::filter::FieldMetadata& fieldMetadata,
23     vtkm::filter::PolicyBase<Policy>);
24 };
25
26 } // namespace filter
27 } // namespace vtkm
```

Notice that within the declaration of FieldMagnitude in Example 22.1, there is also the declaration of a type named SupportedTypes (starting on line 11). This type provides a type list containing all the types that are valid for the input field. (Type lists are discussed in detail in Section 19.7.2.) In the particular case of our FieldMagnitude filter, the filter expects to operate on some type of vector field. Thus, SupportedTypes is set to a list of all standard floating point Vecs.

Once the filter class is declared in the .h file, the implementation filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named FieldMagnitude.hxx. The

.h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of DoExecute is straightforward. A worklet is invoked to compute a new field array. DoExecute then returns a newly constructed vtkm::cont::DataSet object containing the result. There are numerous ways to create a DataSet, but to simplify making filters, VTK-m provides the vtkm::filter::CreateResult function to simplify the process. There are several overloaded versions of vtkm::filter::CreateResult (defined in header file vtkm/filter/CreateResult.h), but the simplest one to use for a filter that adds a field takes the following 5 arguments.

- The input data set. This is the same data set passed to the first argument of DoExecute.
- The array containing the data for the new field, which was presumably computed by the filter.
- The name of the new field.
- A vtkm::filter::FieldMetadata object containing information about the association of the array to which topological elements of the input (for example cells or points). Typically, this FieldMetadata object comes from the input parameters of DoExecute.

Note that this form of CreateResult is generally applicable when the field created is of the same type as the input field. For example, when computing the magnitude of a vector field, if the input vectors are on the points, the output scalars will also be on the points and likewise if the input is on the cells. In the case where the field association changes, it is better to use a different form of CreateResult. If the output field is associated with points, such as when averaging cell fields to each point, then it is best to use vtkm::filter::CreateResultFieldPoint. Likewise, if the output field is associated with cells, such as when computing the volume of each cell, then it is best to use vtkm::filter::CreateResultFieldCell. Both functions only need the three arguments of the input data set, the array for the field's data, and the name of the new field.

Note that all fields need a unique name, which is the reason for the third argument to CreateResult. The vtkm::filter::FilterField base class contains a pair of methods named SetOutputFieldName and GetOutputFieldName to allow users to specify the name of output fields. The DoExecute method should respect the given output field name. However, it is also good practice for the filter to have a default name if none is given. This might be simply specifying a name in the constructor, but it is worthwhile for many filters to derive a name based on the name of the input field.

Example 22.2: Implementation of a field filter.

```
namespace vtkm
{
namespace filter
{

VTKM_CONT
FieldMagnitude::FieldMagnitude()
{
  this->SetOutputFieldName("");
}

template<typename ArrayHandleType, typename Policy>
VTKM_CONT vtkm::cont::DataSet FieldMagnitude::DoExecute(
  const vtkm::cont::DataSet& inDataSet,
  const ArrayHandleType& inField,
  const vtkm::filter::FieldMetadata& fieldMetadata,
  vtkm::filter::PolicyBase<Policy>)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);

  using ComponentType =
    typename vtkm::VecTraits<typename ArrayHandleType::ValueType>::ComponentType;

  vtkm::cont::ArrayHandle<ComponentType> outField;

  this->Invoke(vtkm::worklet::ComputeMagnitude{}, inField, outField);

  std::string outFieldName = this->GetOutputFieldName();
  if (outFieldName == "")
  {
    outFieldName = fieldMetadata.GetName() + "_magnitude";
  }

  return vtkm::filter::CreateResult(
    inDataSet, outField, outFieldName, fieldMetadata);
}

} // namespace filter
} // namespace vtkm
```

22.1.1 Using Cell Connectivity

The previous example performed a simple operation on each element of a field independently. However, it is also common for a "field" filter to take into account the topology of a data set. The interface for the filter is similar, but the implementation involves pulling a vtkm::cont::CellSet from the input vtkm::cont::DataSet and performing operations on fields associated with different topological elements. The steps involve calling DataSet::GetCellSet to retrieve the CellSet object, applying the policy on that object, and then using topology-based worklets, described in Section 21.2, to operate on them.

In this section we provide an example implementation of a field filter on cells that wraps the "cell center" worklet provided in Example 21.3 (listed on page 153). By convention, filter implementations are split into two files. The first file is a standard header file with a .h extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named CellCenter.h.

Example 22.3: Header declaration for a field filter using cell topology.

```
namespace vtkm
{
namespace filter
{

class CellCenters : public vtkm::filter::FilterField<CellCenters>
{
public:
  VTKM_CONT
  CellCenters();

  template<typename ArrayHandleType, typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet,
    const ArrayHandleType& inField,
    const vtkm::filter::FieldMetadata& FieldMetadata,
    vtkm::filter::PolicyBase<Policy>);
};

} // namespace filter
} // namespace vtkm
```

Did you know?

You may have noticed that Example 22.1 provided a specification for SupportedTypes *but Example 22.3 provides no such specification. This demonstrates that declaring* SupportedTypes *is optional. If a filter only works on some limited number of types, then it can use* SupportedTypes *to specify the specific types it supports. But if a filter is generally applicable to many field types, it can simply use the default filter types.*

Once the filter class is declared in the .h file, the implementation filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named CellCenter.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

As with any subclass of FilterField, the filter implements DoExecute, which in this case invokes a worklet to compute a new field array and then return a newly constructed vtkm::cont::DataSet object.

Example 22.4: Implementation of a field filter using cell topology.

```
namespace vtkm
{
namespace filter
{

VTKM_CONT
CellCenters::CellCenters()
{
  this->SetOutputFieldName("");
}

template<typename ArrayHandleType, typename Policy>
VTKM_CONT cont::DataSet CellCenters::DoExecute(
  const vtkm::cont::DataSet& inDataSet,
  const ArrayHandleType& inField,
  const vtkm::filter::FieldMetadata& fieldMetadata,
  vtkm::filter::PolicyBase<Policy>)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);
  using ValueType = typename ArrayHandleType::ValueType;

  if (!fieldMetadata.IsPointField())
  {
    throw vtkm::cont::ErrorBadType("Cell Centers filter operates on point data.");
  }

  vtkm::cont::DynamicCellSet cellSet = inDataSet.GetCellSet();

  vtkm::cont::ArrayHandle<ValueType> outField;

  this->Invoke(vtkm::worklet::CellCenter{},
               vtkm::filter::ApplyPolicyCellSet(cellSet, Policy()),
               inField,
               outField);

  std::string outFieldName = this->GetOutputFieldName();
  if (outFieldName == "")
  {
    outFieldName = fieldMetadata.GetName() + "_center";
  }

  return vtkm::filter::CreateResultFieldCell(inDataSet, outField, outFieldName);
}

} // namespace filter
} // namespace vtkm
```

Common Errors

The policy passed to the DoExecute *method contains information on what types of cell sets should be supported by the execution. This list of cell set types could be different than the default types specified the* DynamicCellSet *returned from* GetCellSet*. Thus, it is important to apply the policy to the cell set before passing it to the* Invoke *method. The policy is applied by calling the* vtkm::filter::ApplyPolicy *function on the* DynamicCellSet*. The use of* ApplyPolicy *is demonstrated in Example 22.4.*

22.2 Data Set Filters

Data set filters are a category of filters that generate a data set with a new cell set based off the cells of an input data set. For example, a data set can be significantly altered by adding, removing, or replacing cells.

Data set filters are implemented in classes that derive the vtkm::filter::FilterDataSet base class. FilterDataSet is a templated class that has a single template argument, which is the type of the concrete subclass.

All FilterDataSet subclasses must implement two methods: DoExecute and DoMapField. The FilterDataSet base class implements Execute and MapFieldOntoOutput methods that process the arguments and then call the DoExecute and DoMapField methods, respectively, of its subclass.

Like vtkm::filter::FilterField all fields are considered to be eligible for propagation. If this behavior is not desired by the caller they can select which fields to propagate via SetFieldsToPass. The actual transformation of the fields so they are valid is handled by DoMapField.

The DoExecute method has the following 2 arguments.

- An input data set contained in a vtkm::cont::DataSet object. (See Chapter 7 for details on DataSet objects.)
- A policy class. See Section 22.4 for information on using policies. The type of the policy is generally unknown until the class is used and requires a template type.

The DoMapField method has the following 4 arguments.

- A vtkm::cont::DataSet object that was returned from the last call to DoExecute. The method both needs information in the DataSet object and writes its results back to it, so the parameter should be declared as a non-const reference. (See Chapter 7 for details on DataSet objects.)
- A field from the DataSet specified in the Execute method to convert so it is applicable to the output DataSet. The field is always passed as an instance of vtkm::cont::ArrayHandle. (See Chapter 16 for details on ArrayHandle objects.) The type of the ArrayHandle is generally not known until the class is used and requires a template type.
- A vtkm::filter::FieldMetadata object that contains the associated metadata of the field not contained in the ArrayHandle of the second argument. The FieldMetadata contains information like the name of the field and what topological element the field is associated with (such as points or cells).

- A policy class. See Section 22.4 for information on using policies. The type of the policy is generally unknown until the class is used and requires a template type.

Did you know?

Some FilterDataSet *subclasses have no need to process each individual field as they can safely propagate or drop all fields. In these cases it is better to override* MapFieldOntoOutput *allowing you to perform this logic directly with* Field *objects and not incur the compile and runtime cost of deducing the underlying* ArrayHandle*.*

In this section we provide an example implementation of a data set filter that wraps the functionality of extracting the edges from a data set as line elements. Many variations of implementing this functionality are given in Chapter 32. For the sake of argument, we are assuming that all the worklets required to implement edge extraction are wrapped up in structure named vtkm::worklet ::ExtractEdges. Furthermore, we assume that ExtractEdges has a pair of methods, Run and ProcessCellField, that create a cell set of lines from the edges in another cell set and average a cell field from input to output, respectively. The ExtractEdges may hold state. All of these assumptions are consistent with the examples in Chapter 32.

By convention, filter implementations are split into two files. The first file is a standard header file with a .h extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named ExtractEdges.h.

Example 22.5: Header declaration for a data set filter.

```
namespace vtkm
{
namespace filter
{

class ExtractEdges : public vtkm::filter::FilterDataSet<ExtractEdges>
{
public:
  template<typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inData,
                                          vtkm::filter::PolicyBase<Policy> policy);

  template<typename T, typename StorageType, typename Policy>
  VTKM_CONT bool DoMapField(vtkm::cont::DataSet& result,
                            const vtkm::cont::ArrayHandle<T, StorageType>& input,
                            const vtkm::filter::FieldMetadata& fieldMeta,
                            const vtkm::filter::PolicyBase<Policy>& policy);

private:
  vtkm::worklet::ScatterCounting::OutputToInputMapType OutputToInputCellMap;
  vtkm::worklet::Keys<vtkm::HashType> CellToEdgeKeys;
  vtkm::cont::ArrayHandle<vtkm::IdComponent> LocalEdgeIndices;
  std::shared_ptr<vtkm::worklet::ScatterCounting> HashCollisionScatter;
};

} // namespace filter
} // namespace vtkm
```

Once the filter class is declared in the .h file, the implementation of the filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named ExtractEdges.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of DoExecute first calls the worklet methods to generate a new CellSet class. It then constructs a DataSet containing this CellSet. It also has to pass all the coordinate systems to the new DataSet. Finally, it returns the DataSet.

Example 22.6: Implementation of the DoExecute method of a data set filter.

```
template<typename Policy>
inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
  const vtkm::cont::DataSet& inData,
  vtkm::filter::PolicyBase<Policy> policy)
{

  const vtkm::cont::DynamicCellSet& inCellSet =
    vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), policy);

  // Count number of edges in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);

  // Build the scatter object (for non 1-to-1 mapping of input to output)
  vtkm::worklet::ScatterCounting scatter(edgeCounts);
  this->OutputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());

  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
               scatter,
               inCellSet,
               vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  vtkm::cont::CellSetSingleType<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  vtkm::cont::DataSet outData;

  outData.SetCellSet(outCellSet);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

The implementation of DoMapField checks to see what elements the given field is associated with (e.g. points or cells), processes the field data as necessary, and adds the field to the DataSet. The vtkm::filter::FieldMetadata passed to DoMapField provides some features to make this process easier. FieldMetadata contains methods to query what the field is associated with such as IsPointField and FieldMetadata::IsCellField. FieldMetadata also has a method named FieldMetadata::AsField that creates a new vtkm::cont::Field object with a new data array and metadata that matches the input.

Example 22.7: Implementation of the DoMapField method of a data set filter.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool ExtractEdges::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& inputArray,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
```

```
{
  vtkm::cont::Field outputField;

  if (fieldMeta.IsPointField())
  {
    outputField = fieldMeta.AsField(inputArray); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    vtkm::cont::ArrayHandle<T> outputCellArray;
    vtkm::cont::ArrayCopy(vtkm::cont::make_ArrayHandlePermutation(
                            this->OutputToInputCellMap, inputArray),
                          outputCellArray);
    outputField = fieldMeta.AsField(outputCellArray);
  }
  else
  {
    return false;
  }

  result.AddField(outputField);

  return true;
}
```

22.3 Data Set with Field Filters

Data set with field filters are a category of filters that generate a data set with a new cell set based off the cells of an input data set along with the data in at least one field. For example, a field might determine how each cell is culled, clipped, or sliced.

Data set with field filters are implemented in classes that derive the `vtkm::filter::FilterDataSetWithField` base class. `FilterDataSetWithField` is a templated class that has a single template argument, which is the type of the concrete subclass.

All `FilterDataSetWithField` subclasses must implement two methods: `DoExecute` and `DoMapField`. The `FilterDataSetWithField` base class implements `Execute` and `MapFieldOntoOutput` methods that process the arguments and then call the `DoExecute` and `DoMapField` methods, respectively, of its subclass.

Like `vtkm::filter::FilterField` all fields are considered to be eligible for propagation. If this behavior is not desired by the caller they can select which fields to propagate via `SetFieldsToPass`. The actual transformation of the fields so they are valid is handled by `DoMapField`.

The `DoExecute` method has the following 4 arguments. (They are the same arguments for `DoExecute` of field filters as described in Section 22.1.)

- An input data set contained in a `vtkm::cont::DataSet` object. (See Chapter 7 for details on `DataSet` objects.)
- The field from the `DataSet` specified in the `Execute` method to operate on. The field is always passed as an instance of `vtkm::cont::ArrayHandle`. (See Chapter 16 for details on `ArrayHandle` objects.) The type of the `ArrayHandle` is generally not known until the class is used and requires a template type.
- A `vtkm::filter::FieldMetadata` object that contains the associated metadata of the field not contained in the `ArrayHandle` of the second argument. The `FieldMetadata` contains information like the name of the field and what topological element the field is associated with (such as points or cells).

- A policy class. See Section 22.4 for information on using policies. The type of the policy is generally unknown until the class is used and requires a template type.

The DoMapField method has the following 4 arguments. (They are the same arguments for DoExecute of field filters as described in Section 22.2.)

- A vtkm::cont::DataSet object that was returned from the last call to DoExecute. The method both needs information in the DataSet object and writes its results back to it, so the parameter should be declared as a non-const reference. (See Chapter 7 for details on DataSet objects.)
- A field from the DataSet specified in the Execute method to convert so it is applicable to the output DataSet. The field is always passed as an instance of vtkm::cont::ArrayHandle. (See Chapter 16 for details on ArrayHandle objects.) The type of the ArrayHandle is generally not known until the class is used and requires a template type.
- A vtkm::filter::FieldMetadata object that contains the associated metadata of the field not contained in the ArrayHandle of the second argument. The FieldMetadata contains information like the name of the field and what topological element the field is associated with (such as points or cells).
- A policy class. See Section 22.4 for information on using policies. The type of the policy is generally unknown until the class is used and requires a template type.

Did you know?

Some FilterDataSetWithField *subclasses have no need to process each individual field as they can safely propagate or drop all fields. In these cases it is better to override* MapFieldOntoOutput *allowing you to perform this logic directly with* Field *objects and not incur the compile and runtime cost of deducing the underlying* ArrayHandle*.*

In this section we provide an example implementation of a data set with field filter that blanks the cells in a data set based on a field that acts as a mask (or stencil). Any cell associated with a mask value of zero will be removed.

We leverage the worklets in the VTK-m source code that implement the Threshold functionality. The threshold worklets are templated on a predicate class that, given a field value, returns a flag on whether to pass or cull the given cell. The vtkm::filter::Threshold class uses a predicate that is true for field values within a range. Our blank cells filter will instead use the predicate class vtkm::NotZeroInitialized that will cull all values where the mask is zero.

By convention, filter implementations are split into two files. The first file is a standard header file with a .h extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named BlankCells.h.

Example 22.8: Header declaration for a data set with field filter.

```
namespace vtkm
{
namespace filter
{

class BlankCells : public vtkm::filter::FilterDataSetWithField<BlankCells>
{
public:
  using SupportedTypes = vtkm::TypeListTagScalarAll;

  template<typename T, typename StorageType, typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet,
    const vtkm::cont::ArrayHandle<T, StorageType>& inField,
    const vtkm::filter::FieldMetadata& fieldMetadata,
    vtkm::filter::PolicyBase<Policy>);

  template<typename T, typename StorageType, typename Policy>
  VTKM_CONT bool DoMapField(vtkm::cont::DataSet& result,
                            const vtkm::cont::ArrayHandle<T, StorageType>& input,
                            const vtkm::filter::FieldMetadata& fieldMeta,
                            const vtkm::filter::PolicyBase<Policy>& policy);

private:
  vtkm::worklet::Threshold Worklet;
};


} // namespace filter
} // namespace vtkm
```

Note that the filter class declaration contains a specification for SupportedTypes to specify that the field is meant to operate specifically on scalar types. Once the filter class is declared in the .h file, the implementation of the filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named BlankCells.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of DoExecute first calls the worklet methods to generate a new CellSet class. It then constructs a DataSet containing this CellSet. It also has to pass all the coordinate systems to the new DataSet. Finally, it returns the DataSet.

Example 22.9: Implementation of the DoExecute method of a data set with field filter.

```
template<typename T, typename StorageType, typename Policy>
VTKM_CONT vtkm::cont::DataSet BlankCells::DoExecute(
  const vtkm::cont::DataSet& inData,
  const vtkm::cont::ArrayHandle<T, StorageType>& inField,
  const vtkm::filter::FieldMetadata& fieldMetadata,
  vtkm::filter::PolicyBase<Policy>)
{
  if (!fieldMetadata.IsCellField())
  {
    throw vtkm::cont::ErrorBadValue("Blanking field must be a cell field.");
  }

  auto inCells = vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), Policy{});

  vtkm::cont::DynamicCellSet outCells =
    this->Worklet.Run(vtkm::filter::ApplyPolicyCellSet(inCells, Policy()),
                      inField,
                      fieldMetadata.GetAssociation(),
                      vtkm::NotZeroInitialized());
  vtkm::cont::DataSet outData;

  outData.SetCellSet(outCells);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

The implementation of DoMapField checks to see what elements the given field is associated with (e.g. points or cells), processes the field data as necessary, and adds the field to the DataSet. The vtkm::filter::FieldMetadata passed to DoMapField provides some features to make this process easier. FieldMetadata contains methods to query what the field is associated with such as IsPointField and IsCellField. FieldMetadata also has a method named AsField that creates a new vtkm::cont::Field object with a new data array and metadata that matches the input.

Example 22.10: Implementation of the DoMapField method of a data set with field filter.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool BlankCells::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& input,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field output;

  if (fieldMeta.IsPointField())
  {
    output = fieldMeta.AsField(input); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    output = fieldMeta.AsField(this->Worklet.ProcessCellField(input));
  }
  else
  {
    return false;
  }

  result.AddField(output);

  return true;
}
```

22.4 Applying Policies

As seen throughout this chapter, filters define their operation by implementing a DoExecute method. The prototype and information passed to the method varies across the different filter types discussed later in this chapter, but one thing they all in common is that they all receive a vtkm::cont::DataSet object that was passed to the Execute method. Although some data extracted from the DataSet may also be passed to DoExecute, it is often necessary to gather further information from the DataSet.

The problem will retrieving data from a DataSet is that the specific types of the data are not known. For example, you can get a vtkm::cont::Field object from the DataSet, but to do anything of interest with the Field, you have to then pull the array out of the field. The problem is that you do not know what type of values are stored in the array. Nor do you know the storage mechanism for the array.

This is where *policies* come into play. All DoExecute methods also receive a policy object. The policy object is really just an empty structure, but it contains important compile-time type information that can be used to deduce which types to expect when pulling fields and cell sets from a DataSet. The policy can be applied by using one of the following functions, all of which are defined in vtkm/filter/PolicyBase.h.

> Common Errors
>
> *Policies define a finite list of types to try when retrieving objects from a* DataSet*. If the* DataSet *holds an object of an uncommon type not checked, then a runtime error will occur.*

22.4.1 Getting Field Data

Filter types like FilterField and FilterDataSetWithField will automatically pull an ArrayHandle from a field in the input DataSet and pass it to the subclass' DoExecute. However, sometimes the filter will need more than one field. Any secondary fields that the filter needs will have to be pulled from the DataSet within the implementation of the DoExecute method.

The best way to do this is to use the vtkm::filter::ApplyPolicyFieldOfType method. This method takes 3 arguments: a vtkm::cont::Field object (presumably pulled from the DataSet), the policy, and the filter (i.e. *this). ApplyPolicyFieldOfType also requires an explicit template argument that specifies the value type desired. Typically this type will be the same type as the type of the primary field's ArrayHandle provided to DoExecute.

Example 22.11: Using a policy to get the ArrayHandle out of a Field.

```
1  vtkm::cont::Field secondaryField = inData.GetField(secondaryFieldName);
2  auto secondaryFieldArrayHandle =
3    vtkm::filter::ApplyPolicyFieldOfType<T>(secondaryField, Policy{}, *this);
```

The object returned from ApplyPolicyFieldOfType is an ArrayHandle. The value type of the ArrayHandle will be the same as the type specified in the template to ApplyPolicyFieldOfType. So the type of secondaryFieldArrayHandle, defined on line 2 of example 22.11 is vtkm::cont::ArrayHandle<T,...>. The storage type of the returned ArrayHandle is rather complex and not documented here.

> Did you know?
>
> *When using* ApplyPolicyFieldOfType*, the value type requested does not have to match exactly with the type in the underlying array. For example, if the* Field *contains an* ArrayHandle *with a value type of* vtkm::Int32 *and you use* ApplyPolicyFieldOfType *to get an array with a value type of* vtkm::Float32*, then you will get an* ArrayHandle *that converts the type from* vtkm::Int32 *to* vtkm::Float32*. This is not done with a deep copy. Rather, this is done any time a value is retrieved from the array (usually during worklet invocation).*

22.4.2 Getting Cell Sets

Any filter that does operations on the topology of a DataSet needs to retrieve a vtkm::cont::CellSet it. Of course, CellSets come in many varieties, and the correct one must be deduced to invoke a worklet on it. This can be done with the vtkm::filter::ApplyPolicyCellSet function.

ApplyPolicyCellSet takes 2 arguments: a vtkm::cont::DynamicCellSet (presumably pulled with DataSet::GetCellSet), and the policy.

Example 22.12: Using a policy to get the CellSet from a DataSet.

```
  auto inCells = vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), Policy{});
```

There are also instances where the algorithm on the CellSet is different depending on whether the CellSet is structured or unstructured (or the algorithm may only work on one or the other). In this case you can use either vtkm::filter::ApplyPolicyCellSetStructured or vtkm::filter::ApplyPolicyCellSetUnstructured to only get CellSets that match that type.

Example 22.13: Using a policy to get on structured cell sets.

```
  auto cellSet =
    vtkm::filter::ApplyPolicyCellSetStructured(inDataSet.GetCellSet(), Policy{});
```

CHAPTER

TWENTYTHREE

WORKLET ERROR HANDLING

It is sometimes the case during the execution of an algorithm that an error condition can occur that causes the computation to become invalid. At such a time, it is important to raise an error to alert the calling code of the problem. Since VTK-m uses an exception mechanism to raise errors, we want an error in the execution environment to throw an exception.

However, throwing exceptions in a parallel algorithm is problematic. Some accelerator architectures, like CUDA, do not even support throwing exceptions. Even on architectures that do support exceptions, throwing them in a thread block can cause problems. An exception raised in one thread may or may not be thrown in another, which increases the potential for deadlocks, and it is unclear how uncaught exceptions progress through thread blocks.

VTK-m handles this problem by using a flag and check mechanism. When a worklet (or other subclass of `vtkm::exec::FunctorBase`) encounters an error, it can call its `RaiseError` method to flag the problem and record a message for the error. Once all the threads terminate, the scheduler checks for the error, and if one exists it throws a `vtkm::cont::ErrorExecution` exception in the control environment. Thus, calling `RaiseError` looks like an exception was thrown from the perspective of the control environment code that invoked it.

Example 23.1: Raising an error in the execution environment.

```
struct SquareRoot : vtkm::worklet::WorkletMapField
{
public:
  using ControlSignature = void(FieldIn, FieldOut);
  using ExecutionSignature = _2(_1);

  template<typename T>
  VTKM_EXEC T operator()(T x) const
  {
    if (x < 0)
    {
      this->RaiseError("Cannot take the square root of a negative number.");
    }
    return vtkm::Sqrt(x);
  }
};
```

It is also worth noting that the `VTKM_ASSERT` macro described in Section 11.2 also works within worklets and other code running in the execution environment. Of course, a failed assert will terminate execution rather than just raise an error so is best for checking invalid conditions for debugging purposes.

CHAPTER

TWENTYFOUR

MATH

VTK-m comes with several math functions that tend to be useful for visualization algorithms. The implementation of basic math operations can vary subtly on different accelerators, and these functions provide cross platform support.

All math functions are located in the `vtkm` package. The functions are most useful in the execution environment, but they can also be used in the control environment when needed.

24.1 Basic Math

The vtkm/Math.h header file contains several math functions that replicate the behavior of the basic POSIX math functions as well as related functionality.

> Did you know?
>
> *When writing worklets, you should favor using these math functions provided by VTK-m over the standard math functions in math.h. VTK-m's implementation manages several compiling and efficiency issues when porting.*

`vtkm::Abs` Returns the absolute value of the single argument. If given a vector, performs a component-wise operation.

`vtkm::ACos` Returns the arccosine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ACosH` Returns the hyperbolic arccossine. If given a vector, performs a component-wise operation.

`vtkm::ASin` Returns the arcsine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ASinH` Returns the hyperbolic arcsine. If given a vector, performs a component-wise operation.

`vtkm::ATan` Returns the arctangent of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ATan2` Computes the arctangent of y/x where y is the first argument and x is the second argument. `ATan2` uses the signs of both arguments to determine the quadrant of the return value. `ATan2` is only defined for floating point types (no vectors).

`vtkm::ATanH` Returns the hyperbolic arctangent. If given a vector, performs a component-wise operation.

`vtkm::Cbrt` Takes one argument and returns the cube root of that argument. If called with a vector type, returns a component-wise cube root.

`vtkm::Ceil` Rounds and returns the smallest integer not less than the single argument. If given a vector, performs a component-wise operation.

`vtkm::CopySign` Copies the sign of the second argument onto the first argument and returns that. If the second argument is positive, returns the absolute value of the first argument. If the second argument is negative, returns the negative absolute value of the first argument.

`vtkm::Cos` Returns the cosine of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::CosH` Returns the hyperbolic cosine. If given a vector, performs a component-wise operation.

`vtkm::Epsilon` Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Epsilon32` and `Epsilon64` are non-templated versions that return the precision for a particular precision.

`vtkm::Exp` Computes e^x where x is the argument to the function and e is Euler's number (approximately 2.71828). If called with a vector type, returns a component-wise exponent.

`vtkm::Exp10` Computes 10^x where x is the argument. If called with a vector type, returns a component-wise exponent.

`vtkm::Exp2` Computes 2^x where x is the argument. If called with a vector type, returns a component-wise exponent.

`vtkm::ExpM1` Computes $e^x - 1$ where x is the argument to the function and e is Euler's number (approximately 2.71828). The accuracy of this function is good even for very small values of x. If called with a vector type, returns a component-wise exponent.

`vtkm::Floor` Rounds and returns the largest integer not greater than the single argument. If given a vector, performs a component-wise operation.

`vtkm::FMod` Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards zero to an integer. For example, `FMod(6.5,2.3)` returns 1.9, which is $6.5 - 2 \cdot 4.6$. If given vectors, `FMod` performs a component-wise operation. `FMod` is similar to `Remainder` except that the quotient is rounded toward 0 instead of the nearest integer.

`vtkm::Infinity` Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Infinity32` and `Infinity64` are non-templated versions that return the precision for a particular precision.

`vtkm::IsFinite` Returns true if the argument is a normal number (neither a NaN nor an infinite).

`vtkm::IsInf` Returns true if the argument is either positive infinity or negative infinity.

`vtkm::IsNan` Returns true if the argument is not a number (NaN).

vtkm::IsNegative Returns true if the single argument is less than zero, false otherwise.

vtkm::Log Computes the natural logarithm (i.e. logarithm to the base e) of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log10 Computes the logarithm to the base 10 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log1P Computes $\ln(1+x)$ where x is the single argument and ln is the natural logarithm (i.e. logarithm to the base e). The accuracy of this function is good for very small values. If called with a vector type, returns a component-wise logarithm.

vtkm::Log2 Computes the logarithm to the base 2 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Max Takes two arguments and returns the argument that is greater. If called with a vector type, returns a component-wise maximum.

vtkm::Min Takes two arguments and returns the argument that is lesser. If called with a vector type, returns a component-wise minimum.

vtkm::ModF Returns the integral and fractional parts of the first argument. The second argument is a reference in which the integral part is stored. The return value is the fractional part. If given vectors, ModF performs a component-wise operation.

vtkm::Nan Returns the representation for not-a-number (NaN). A NaN represents an invalid value or the result of an invalid operation such as 0/0. A NaN is neither greater than nor less than nor equal to any other number including other NaNs. The NaN method is templated to specify either a 32 or 64 bit floating point number. The convenience methods Nan32 and NaN64 are non-templated versions that return the precision for a particular precision.

vtkm::NegativeInfinity Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The NegativeInfinity method is templated to specify either a 32 or 64 bit floating point number. The convenience methods NagativeInfinity32 and NegativeInfinity64 are non-templated versions that return the precision for a particular precision.

vtkm::Pi Returns the constant π (about 3.14159).

vtkm::Pi_2 Returns the constant $\pi/2$ (about 1.570796).

vtkm::Pi_3 Returns the constant $\pi/3$ (about 1.047197).

vtkm::Pi_4 Returns the constant $\pi/4$ (about 0.785398).

vtkm::Pow Takes two arguments and returns the first argument raised to the power of the second argument. This function is only defined for vtkm::Float32 and vtkm::Float64.

vtkm::RCbrt Takes one argument and returns the cube root of that argument. The result of this function is equivalent to `1/Cbrt(x)`. However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

vtkm::Remainder Computes the remainder on the division of 2 floating point numbers. The return value is $\mathit{numerator} - n \cdot \mathit{denominator}$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer. For example, `FMod(6.5,2.3)` returns -0.4, which is $6.5 - 3 \cdot 2.3$. If given vectors, Remainder performs a component-wise operation. Remainder is similar to FMod except that the quotient is rounded toward the nearest integer instead of toward 0.

vtkm::RemainderQuotient Performs an operation identical to Reminder. In addition, this function takes a third argument that is a reference in which the quotient is given.

vtkm::Round Rounds and returns the integer nearest the single argument. If given a vector, performs a component-wise operation.

vtkm::RSqrt Takes one argument and returns the square root of that argument. The result of this function is equivalent to `1/Sqrt(x)`. However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

vtkm::SignBit Returns a nonzero value if the single argument is negative.

vtkm::Sin Returns the sine of an angle given in radians. If given a vector, performs a component-wise operation.

vtkm::SinH Returns the hyperbolic sine. If given a vector, performs a component-wise operation.

vtkm::Sqrt Takes one argument and returns the square root of that argument. If called with a vector type, returns a component-wise square root. On some hardware it is faster to find the reciprocal square root, so RSqrt should be used if you actually plan to divide byt the square root.

vtkm::Tan Returns the tangent of an angle given in radians. If given a vector, performs a component-wise operation.

vtkm::TanH Returns the hyperbolic tangent. If given a vector, performs a component-wise operation.

vtkm::TwoPi Returns the constant 2π (about 6.283185).

24.2 Vector Analysis

Visualization and computational geometry algorithms often perform vector analysis operations. The vtkm/VectorAnalysis.h header file provides functions that perform the basic common vector analysis operations.

vtkm::Cross Returns the cross product of two vtkm::Vec of size 3.

vtkm::Lerp Given two values x and y in the first two parameters and a weight w as the third parameter, interpolates between x and y. Specifically, the linear interpolation is $(y-x)w+x$ although Lerp might compute the interpolation faster than using the independent arithmetic operations. The two values may be scalars or equal sized vectors. If the two values are vectors and the weight is a scalar, all components of the vector are interpolated with the same weight. If the weight is also a vector, then each component of the value vectors are interpolated with the respective weight component.

vtkm::Magnitude Returns the magnitude of a vector. This function works on scalars as well as vectors, in which case it just returns the scalar. It is usually much faster to compute MagnitudeSquared, so that should be substituted when possible (unless you are just going to take the square root, which would be besides the point). On some hardware it is also faster to find the reciprocal magnitude, so RMagnitude should be used if you actually plan to divide by the magnitude.

vtkm::MagnitudeSquared Returns the square of the magnitude of a vector. It is usually much faster to compute the square of the magnitude than the length, so you should use this function in place of Magnitude or RMagnitude when needing the square of the magnitude or any monotonically increasing function of a magnitude or distance. This function works on scalars as well as vectors, in which case it just returns the square of the scalar.

vtkm::Normal Returns a normalized version of the given vector. The resulting vector points in the same direction as the argument but has unit length.

vtkm::Normalize Takes a reference to a vector and modifies it to be of unit length. Normalize(v) is functionally equivalent to v *= RMagnitude(v).

vtkm::RMagnitude Returns the reciprocal magnitude of a vector. On some hardware RMagnitude is faster than Magnitude, but neither is as fast as MagnitudeSquared. This function works on scalars as well as vectors, in which case it just returns the reciprocal of the scalar.

vtkm::TriangleNormal Given three points in space (contained in vtkm::Vec s of size 3) that compose a triangle return a vector that is perpendicular to the triangle. The magnitude of the result is equal to twice the area of the triangle. The result points away from the "front" of the triangle as defined by the standard counter-clockwise ordering of the points.

24.3 Matrices

Linear algebra operations on small matrices that are done on a single thread are located in vtkm/Matrix.h.

This header defines the vtkm::Matrix templated class. The template parameters are first the type of component, then the number of rows, then the number of columns. The overloaded parentheses operator can be used to retrieve values based on row and column indices. Likewise, the bracket operators can be used to reference the Matrix as a 2D array (indexed by row first). The following example builds a Matrix that contains the values

$$\begin{vmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \end{vmatrix}$$

Example 24.1: Creating a Matrix.

```
vtkm::Matrix<vtkm::Float32, 2, 3> matrix;

// Using parenthesis notation.
matrix(0, 0) = 0.0f;
matrix(0, 1) = 1.0f;
matrix(0, 2) = 2.0f;

// Using bracket notation.
matrix[1][0] = 10.0f;
matrix[1][1] = 11.0f;
matrix[1][2] = 12.0f;
```

The vtkm/Matrix.h header also defines the following functions that operate on matrices.

vtkm::MatrixDeterminant Takes a square Matrix as its single argument and returns the determinant of that matrix.

vtkm::MatrixGetColumn Given a Matrix and a column index, returns a vtkm::Vec of that column. This function might not be as efficient as vtkm::MatrixRow. (It performs a copy of the column).

vtkm::MatrixGetRow Given a Matrix and a row index, returns a vtkm::Vec of that row.

vtkm::MatrixIdentity Returns the identity matrix. If given no arguments, it creates an identity matrix and returns it. (In this form, the component type and size must be explicitly set.) If given a single square matrix argument, fills that matrix with the identity.

vtkm::MatrixInverse Finds and returns the inverse of a given matrix. The function takes two arguments. The first argument is the matrix to invert. The second argument is a reference to a Boolean that is set to true if the inverse is found or false if the matrix is singular and the returned matrix is incorrect.

vtkm::MatrixMultiply Performs a matrix-multiply on its two arguments. Overloaded to work for matrix-matrix, vector-matrix, or matrix-vector multiply.

vtkm::MatrixSetColumn Given a Matrix, a column index, and a vtkm::Vec, sets the column of that index to the values of the Tuple.

vtkm::MatrixSetRow Given a Matrix, a row index, and a vtkm::Vec, sets the row of that index to the values of the Tuple.

vtkm::MatrixTranspose Takes a Matrix and returns its transpose.

vtkm::SolveLinearSystem Solves the linear system $Ax = b$ and returns x. The function takes three arguments. The first two arguments are the matrix A and the vector b, respectively. The third argument is a reference to a Boolean that is set to true if a single solution is found, false otherwise.

24.4 Newton's Method

VTK-m's matrix methods (documented in Section 24.3) provide a method to solve a small linear system of equations. However, sometimes it is necessary to solve a small nonlinear system of equations. This can be done with the vtkm::NewtonsMethod function defined in the vtkm/NewtonsMethod.h header.

The NewtonsMethod function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the NewtonsMethod. The function takes the following 6 parameters (three of which are optional).

1. A functor whose operation takes a vtkm::Vec and returns a vtkm::Matrix containing the math function's Jacobian vector at that point.
2. A functor whose operation takes a vtkm::Vec and returns the evaluation of the math function at that point as another vtkm::Vec.
3. The vtkm::Vec that represents the desired output of the function.
4. A vtkm::Vec to use as the initial guess. If not specified, the origin is used.
5. The convergence distance. If the iterative method changes all values less than this amount, then it considers the solution found. If not specified, set to 10^{-3}.
6. The maximum amount of iterations to run before giving up and returning the best solution. If not specified, set to 10.

The NewtonsMethod function returns a vtkm::NewtonsMethodResult object. NewtonsMethodResult is a struct templated on the type and number of input values of the nonlinear system. NewtonsMethodResult contains the following items.

Valid A bool that is set to false if the solution runs into a singularity so that no possible solution is found.

Converged A bool that is set to true if a solution is found that is within the convergence distance specified. It is set to false if the method did not convert in the specified number of iterations.

`Solution` A `vtkm::Vec` containing the solution to the nonlinear system. If `Converged` is false, then this value is likely inaccurate. If `Valid` is false, then this value is undefined.

Example 24.2: Using `NewtonsMethod` to solve a small system of nonlinear equations.

```cpp
// A functor for the mathematical function f(x) = [dot(x,x),x[0]*x[1]]
struct FunctionFunctor
{
  template<typename T>
  VTKM_EXEC_CONT vtkm::Vec<T, 2> operator()(const vtkm::Vec<T, 2>& x) const
  {
    return vtkm::make_Vec(vtkm::Dot(x, x), x[0] * x[1]);
  }
};

// A functor for the Jacobian of the mathematical function
// f(x) = [dot(x,x),x[0]*x[1]], which is
//   | 2*x[0] 2*x[1] |
//   |   x[1]   x[0] |
struct JacobianFunctor
{
  template<typename T>
  VTKM_EXEC_CONT vtkm::Matrix<T, 2, 2> operator()(const vtkm::Vec<T, 2>& x) const
  {
    vtkm::Matrix<T, 2, 2> jacobian;
    jacobian(0, 0) = 2 * x[0];
    jacobian(0, 1) = 2 * x[1];
    jacobian(1, 0) = x[1];
    jacobian(1, 1) = x[0];

    return jacobian;
  }
};

VTKM_EXEC
void SolveNonlinear()
{
  // Use Newton's method to solve the nonlinear system of equations:
  //
  //    x^2 + y^2 = 2
  //    x*y = 1
  //
  // There are two possible solutions, which are (x=1,y=1) and (x=-1,y=-1).
  // The one found depends on the starting value.
  vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer1 =
    vtkm::NewtonsMethod(JacobianFunctor(),
                        FunctionFunctor(),
                        vtkm::make_Vec(2.0f, 1.0f),
                        vtkm::make_Vec(1.0f, 0.0f));
  if (!answer1.Valid || !answer1.Converged)
  {
    // Failed to find solution
  }
  // answer1.Solution is [1,1]

  vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer2 =
    vtkm::NewtonsMethod(JacobianFunctor(),
                        FunctionFunctor(),
                        vtkm::make_Vec(2.0f, 1.0f),
                        vtkm::make_Vec(0.0f, -2.0f));
  if (!answer2.Valid || !answer2.Converged)
  {
    // Failed to find solution
  }
  // answer2 is [-1,-1]
}
```

CHAPTER

TWENTYFIVE

WORKING WITH CELLS

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See Section 7.2 starting on page 36 for information on defining cell sets in the control environment.) When worklets that operate on cells are scheduled, these grid structures are broken into their independent cells, and that data is handed to the worklet. Thus, cell-based operations in the execution environment exclusively operate on independent cells.

Unlike some other libraries such as VTK, VTK-m does not have a cell class that holds all the information pertaining to a cell of a particular type. Instead, VTK-m provides tags or identifiers defining the cell shape, and companion data like coordinate and field information are held in separate structures. This organization is designed so a worklet may specify exactly what information it needs, and only that information will be loaded.

25.1 Cell Shape Tags and Ids

Cell shapes can be specified with either a tag (defined with a struct with a name like `CellShapeTag*`) or an enumerated identifier (defined with a constant number with a name like `CELL_SHAPE_*`). These shape tags and identifiers are defined in vtkm/CellShape.h and declared in the `vtkm` namespace (because they can be used in either the control or the execution environment). Figure 25.1 gives both the identifier and the tag names.

In addition to the basic cell shapes, there is a special "empty" cell with the identifier `vtkm::CELL_SHAPE_EMPTY` and tag `vtkm::CellShapeTagEmpty`. This type of cell has no points, edges, or faces and can be thought of as a placeholder for a null or void cell.

There is also a special cell shape "tag" named `vtkm::CellShapeTagGeneric` that is used when the actual cell shape is not known at compile time. `CellShapeTagGeneric` actually has a member variable named `Id` that stores the identifier for the cell shape. There is no equivalent identifier for a generic cell; cell shape identifiers can be placed in a `vtkm::IdComponent` at runtime.

When using cell shapes in templated classes and functions, you can use the `VTKM_IS_CELL_SHAPE_TAG` to ensure a type is a valid cell shape tag. This macro takes one argument and will produce a compile error if the argument is not a cell shape tag type.

25.1.1 Converting Between Tags and Identifiers

Every cell shape tag has a member variable named `Id` that contains the identifier for the cell shape. This provides a convenient mechanism for converting a cell shape tag to an identifier. Most cell shape tags have their `Id` member as a compile-time constant, but `CellShapeTagGeneric` is set at run time.

vtkm/CellShape.h also declares a templated class named `vtkm::CellShapeIdToTag` that converts a cell shape

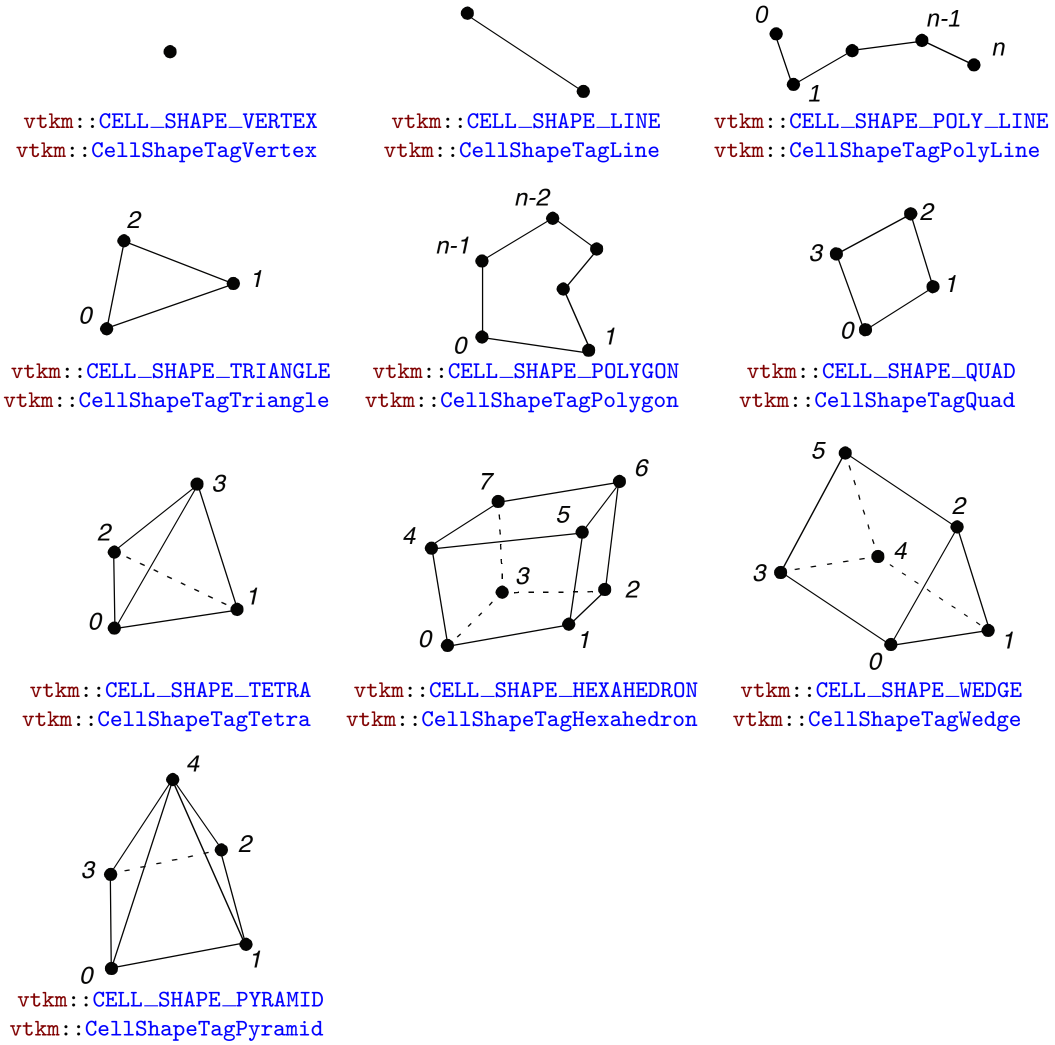


Figure 25.1: Basic Cell Shapes

identifier to a cell shape tag. CellShapeIdToTag has a single template argument that is the identifier. Inside the class is a type named Tag that is the type of the correct tag.

Example 25.1: Using CellShapeIdToTag.

```
void CellFunction(vtkm::CellShapeTagTriangle)
{
  std::cout << "In CellFunction for triangles." << std::endl;
}

void DoSomethingWithACell()
{
  // Calls CellFunction overloaded with a vtkm::CellShapeTagTriangle.
  CellFunction(vtkm::CellShapeIdToTag<vtkm::CELL_SHAPE_TRIANGLE>::Tag());
}
```

However, CellShapeIdToTag is only viable if the identifier can be resolved at compile time. In the case where a cell identifier is stored in a variable or an array or the code is using a CellShapeTagGeneric, the correct cell shape is not known at run time. In this case, vtkmGenericCellShapeMacro can be used to check all possible conditions. This macro is embedded in a switch statement where the condition is the cell shape identifier. vtkmGenericCellShapeMacro has a single argument, which is an expression to be executed. Before the expression is executed, a type named CellShapeTag is defined as the type of the appropriate cell shape tag. Often this method is used to implement the condition for a CellShapeTagGeneric in a function overloaded for cell types. A demonstration of vtkmGenericCellShapeMacro is given in Example 25.2.

25.1.2 Cell Traits

The vtkm/CellTraits.h header file contains a traits class named vtkm::CellTraits that provides information about a cell. Each specialization of CellTraits contains the following members.

TOPOLOGICAL_DIMENSIONS Defines the topological dimensions of the cell type. This is 3 for polyhedra, 2 for polygons, 1 for lines, and 0 for points.

TopologicalDimensionsTag A type set to either vtkm::CellTopologicalDimensionsTag <3>, CellTopologicalDimensionsTag<2>, CellTopologicalDimensionsTag<1>, or CellTopologicalDimensionsTag<0>. The number is consistent with TOPOLOGICAL_DIMENSIONS. This tag is provided for convenience when specializing functions.

IsSizeFixed Set to either vtkm::CellTraitsTagSizeFixed for cell types with a fixed number of points (for example, triangle) or vtkm::CellTraitsTagSizeVariable for cell types with a variable number of points (for example, polygon).

NUM_POINTS A vtkm::IdComponent set to the number of points in the cell. This member is only defined when there is a constant number of points (i.e. IsSizeFixed is set to vtkm::CellTraitsTagSizeFixed).

Example 25.2: Using CellTraits to implement a polygon normal estimator.

```
namespace detail
{

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector, typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
  const PointCoordinatesVector& pointCoordinates,
  vtkm::CellTopologicalDimensionsTag<2>,
  const WorkletType& worklet)
{
  if (pointCoordinates.GetNumberOfComponents() >= 3)
  {
    return vtkm::TriangleNormal(
      pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
  }
  else
  {
    worklet.RaiseError("Degenerate polygon.");
    return typename PointCoordinatesVector::ComponentType();
  }
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector,
         vtkm::IdComponent Dimensions,
         typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
  const PointCoordinatesVector&,
  vtkm::CellTopologicalDimensionsTag<Dimensions>,
  const WorkletType& worklet)
{
  worklet.RaiseError("Only polygons supported for cell normals.");
  return typename PointCoordinatesVector::ComponentType();
}

} // namespace detail

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename CellShape, typename PointCoordinatesVector, typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
  CellShape,
  const PointCoordinatesVector& pointCoordinates,
  const WorkletType& worklet)
{
  return detail::CellNormalImpl(
    pointCoordinates,
    typename vtkm::CellTraits<CellShape>::TopologicalDimensionsTag(),
    worklet);
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector, typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
  vtkm::CellShapeTagGeneric shape,
  const PointCoordinatesVector& pointCoordinates,
  const WorkletType& worklet)
{
  switch (shape.Id)
  {
    vtkmGenericCellShapeMacro(
      return CellNormal(CellShapeTag(), pointCoordinates, worklet));
    default:
      worklet.RaiseError("Unknown cell type.");
      return typename PointCoordinatesVector::ComponentType();
  }
}
```

25.2 Parametric and World Coordinates

Each cell type supports a one-to-one mapping between a set of parametric coordinates in the unit cube (or some subset of it) and the points in 3D space that are the locus contained in the cell. Parametric coordinates are useful because certain features of the cell, such as vertex location and center, are at a consistent location in parametric space irrespective of the location and distortion of the cell in world space. Also, many field operations are much easier with parametric coordinates.

The vtkm/exec/ParametricCoordinates.h header file contains the following functions for working with parametric coordinates.

vtkm::exec::ParametricCoordinatesCenter Returns the parametric coordinates for the center of a given shape. It takes 4 arguments: the number of points in the cell, a vtkm::Vec of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a vtkm::Vec <vtkm::FloatDefault,3> instead of passing it as a parameter.

vtkm::exec::ParametricCoordinatesPoint Returns the parametric coordinates for a given point of a given shape. It takes 5 arguments: the number of points in the cell, the index of the point to query, a vtkm::Vec of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a vtkm::Vec <<vtkm::FloatDefault,3> instead of passing it as a parameter.

vtkm::exec::ParametricCoordinatesToWorldCoordinates Given a vector of point coordinates (usually given by a FieldPointIn worklet argument), a vtkm::Vec of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors), returns the world coordinates.

vtkm::exec::WorldCoordinatesToParametricCoordinates Given a vector of point coordinates (usually given by a FieldPointIn worklet argument), a vtkm::Vec of size 3 containing world coordinates, a shape tag, and a worklet object (for raising errors), returns the parametric coordinates. This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

25.3 Interpolation

The shape of every cell is defined by the connections of some finite set of points. Field values defined on those points can be interpolated to any point within the cell to estimate a continuous field.

The vtkm/exec/CellInterpolate.h header contains the function vtkm::exec::CellInterpolate that takes a vector of point field values (usually given by a FieldPointIn worklet argument), a vtkm::Vec of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors). It returns the field interpolated to the location represented by the given parametric coordinates.

Example 25.3: Interpolating field values to a cell's center.

```
struct CellCenters : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn,
                                FieldInPoint inputField,
                                FieldOutCell outputField);
  using ExecutionSignature = void(CellShape, PointCount, _2, _3);
  using InputDomain = _1;

  template<typename CellShapeTag, typename FieldInVecType, typename FieldOutType>
  VTKM_EXEC void operator()(CellShapeTag shape,
                            vtkm::IdComponent pointCount,
                            const FieldInVecType& inputField,
                            FieldOutType& outputField) const
  {
    vtkm::Vec3f center =
      vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField = vtkm::exec::CellInterpolate(inputField, center, shape, *this);
  }
};
```

25.4 Derivatives

Since interpolations provide a continuous field function over a cell, it is reasonable to consider the derivative of this function. The vtkm/exec/CellDerivative.h header contains the function vtkm::exec::CellDerivative that takes a vector of scalar point field values (usually given by a FieldPointIn worklet argument), a vtkm::Vec of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors). It returns the field derivative at the location represented by the given parametric coordinates. The derivative is return in a vtkm::Vec of size 3 corresponding to the partial derivatives in the x, y, and z directions. This derivative is equivalent to the gradient of the field.

Example 25.4: Computing the derivative of the field at cell centers.

```
struct CellDerivatives : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn,
                                FieldInPoint inputField,
                                FieldInPoint pointCoordinates,
                                FieldOutCell outputField);
  using ExecutionSignature = void(CellShape, PointCount, _2, _3, _4);
  using InputDomain = _1;

  template<typename CellShapeTag,
           typename FieldInVecType,
           typename PointCoordVecType,
           typename FieldOutType>
  VTKM_EXEC void operator()(CellShapeTag shape,
                            vtkm::IdComponent pointCount,
                            const FieldInVecType& inputField,
                            const PointCoordVecType& pointCoordinates,
                            FieldOutType& outputField) const
  {
    vtkm::Vec3f center =
      vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField =
      vtkm::exec::CellDerivative(inputField, pointCoordinates, center, shape, *this);
  }
};
```

25.5 Edges and Faces

As explained earlier in this chapter, a cell is defined by a collection of points and a shape identifier that describes how the points come together to form the structure of the cell. The cell shapes supported by VTK-m are documented in Section 25.1. It contains Figure 25.1 on page 194, which shows how the points for each shape form the structure of the cell.

Most cell shapes can be broken into subelements. 2D and 3D cells have pairs of points that form *edges* at the boundaries of the cell. Likewise, 3D cells have loops of edges that form *faces* that encase the cell. Figure 25.2 demonstrates the relationship of these constituent elements for some example cell shapes.

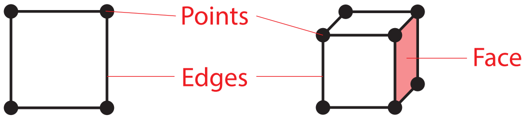


Figure 25.2: The constituent elements (points, edges, and faces) of cells.

The header file vtkm/exec/CellEdge.h contains a collection of functions to help identify the edges of a cell. The first such function is `vtkm::exec::CellEdgeNumberOfEdges`. This function takes the number of points in the cell, the shape of the cell, and an instance of the calling worklet (for error reporting). It returns the number of edges the cell has (as a `vtkm::IdComponent`).

The second function is `vtkm::exec::CellEdgeLocalIndex`. This function takes, respectively, the number of points, the local index of the point in the edge (0 or 1), the local index of the edge (0 to the number of edges in the cell), the shape of the cell, and an instance of the calling worklet. It returns a `vtkm::IdComponent` containing the local index (between 0 and the number of points in the cell) of the requested point in the edge. This local point index is consistent with the point labels in Figure 25.2. To get the point indices relative to the data set, the edge indices should be used to reference a `PointIndices` list.

The third function is vtkm::exec::CellEdgeCanonicalId. This function takes the number of points, the local index of the edge, the shape of the cell, a Vec-like containing the global id of each cell point, and an instance of the calling worklet. It returns a vtkm::Id2 that is globally unique to that edge. If CellEdgeCanonicalId is called on an edge for a different cell, the two will be the same if and only if the two cells share that edge. CellEdgeCanonicalId is useful for finding coincident components of topology.

The following example demonstrates a pair of worklets that use the cell edge functions. As is typical for operations of this nature, one worklet counts the number of edges in each cell and another uses this count to generate the data.

Did you know?

Example 25.5 demonstrates one of many techniques for creating cell sets in a worklet. Chapter 32 describes this and many more such techniques.

Example 25.5: Using cell edge functions.

```
struct EdgesCount : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn, FieldOutCell numEdgesInCell);
  using ExecutionSignature = _2(CellShape, PointCount);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC vtkm::IdComponent operator()(CellShapeTag cellShape,
                                         vtkm::IdComponent numPointsInCell) const
  {
    return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
  }
};

struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
  using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag,
           typename PointIndexVecType,
           typename EdgeIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::IdComponent edgeIndex,
                            EdgeIndexVecType& edgeIndices) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();

    vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 0, edgeIndex, cellShape, *this);
    vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 1, edgeIndex, cellShape, *this);

    edgeIndices[0] = globalPointIndicesForCell[pointInCellIndex0];
    edgeIndices[1] = globalPointIndicesForCell[pointInCellIndex1];
  }
};
```

The header file vtkm/exec/CellFace.h contains a collection of functions to help identify the faces of a cell. The first such function is vtkm::exec::CellFaceNumberOfFaces. This function takes the shape of the cell and an instance of the calling worklet (for error reporting). It returns the number of faces the cell has (as a vtkm::IdComponent).

The second function is vtkm::exec::CellFaceNumberOfPoints. This function takes the local index of the face (0 to the number of faces in the cell), the shape of the cell, and an instance of the calling worklet. It returns the number of points the specified face has (as a vtkm::IdComponent).

The third function is vtkm::exec::CellFaceLocalIndex. This function takes, respectively, the local index of the point in the face (0 to the number of points in the face), the local index of the face (0 to the number of faces in the cell), the shape of the cell, and an instance of the calling worklet. It returns a vtkm::IdComponent containing the local index (between 0 and the number of points in the cell) of the requested point in the face. The points are indexed in counterclockwise order when viewing the face from the outside of the cell. This local point index is consistent with the point labels in Figure 25.2. To get the point indices relative to the data set, the face indices should be used to reference a PointIndices list.

The fourth function is vtkm::exec::CellFaceCanonicalId. This function takes the local index of the face, the shape of the cell, a Vec-like containing the global id of each cell point, and an instance of the calling worklet. It returns a vtkm::Id3 that is globally unique to that face. If CellFaceCanonicalId is called on a face for a different cell, the two will be the same if and only if the two cells share that face. CellFaceCanonicalId is useful for finding coincident components of topology.

The following example demonstrates a triple of worklets that use the cell face functions. As is typical for operations of this nature, the worklets are used in steps to first count entities and then generate new entities. In this case, the first worklet counts the number of faces and the second worklet counts the points in each face. The third worklet generates cells for each face.

Example 25.6: Using cell face functions.

```
struct FacesCount : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn, FieldOutCell numFacesInCell);
  using ExecutionSignature = _2(CellShape);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC vtkm::IdComponent operator()(CellShapeTag cellShape) const
  {
    return vtkm::exec::CellFaceNumberOfFaces(cellShape, *this);
  }
};

struct FacesCountPoints : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn,
                               FieldOutCell numPointsInFace,
                               FieldOutCell faceShape);
  using ExecutionSignature = void(CellShape, VisitIndex, _2, _3);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            vtkm::IdComponent faceIndex,
                            vtkm::IdComponent& numPointsInFace,
                            vtkm::UInt8& faceShape) const
  {
    numPointsInFace =
      vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, *this);
    switch (numPointsInFace)
    {
      case 3:
        faceShape = vtkm::CELL_SHAPE_TRIANGLE;
        break;
      case 4:
        faceShape = vtkm::CELL_SHAPE_QUAD;
        break;
      default:
        faceShape = vtkm::CELL_SHAPE_POLYGON;
        break;
    }
  }
};

struct FacesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn, FieldOutCell faceIndices);
  using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag,
           typename PointIndexVecType,
           typename FaceIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::IdComponent faceIndex,
                            FaceIndexVecType& faceIndices) const
  {
    vtkm::IdComponent numPointsInFace = faceIndices.GetNumberOfComponents();
    VTKM_ASSERT(numPointsInFace ==
                vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, *this));
    for (vtkm::IdComponent pointInFaceIndex = 0;
         pointInFaceIndex < numPointsInFace;
         pointInFaceIndex++)
    {
      vtkm::IdComponent pointInCellIndex = vtkm::exec::CellFaceLocalIndex(
        pointInFaceIndex, faceIndex, cellShape, *this);
      faceIndices[pointInFaceIndex] = globalPointIndicesForCell[pointInCellIndex];
    }
  }
};
```

CHAPTER

TWENTYSIX

FANCY ARRAY HANDLES

One of the features of using ArrayHandles is that they hide the implementation and layout of the array behind a generic interface. This gives us the opportunity to replace a simple C array with some custom definition of the data and the code using the ArrayHandle is none the wiser.

This gives us the opportunity to implement *fancy* arrays that do more than simply look up a value in an array. For example, arrays can be augmented on the fly by mutating their indices or values. Or values could be computed directly from the index so that no storage is required for the array at all.

VTK-m provides many of the fancy arrays, which we explore in this section. Later in Chapter 36 we explore how to create custom arrays that adapt new memory layouts or augment other types of arrays.

Did you know?

One of the advantages of VTK-m's implementation of fancy arrays is that they can define whole arrays without actually storing and values. For example, ArrayHandleConstant*,* ArrayHandleCounting*, and* ArrayHandleIndex *do not store data in any array in memory. Rather, they construct the value for an index at runtime. Likewise, arrays like* ArrayHandlePermute *construct new arrays from the values of other arrays without having to create a copy of the data.*

26.1 Constant Arrays

A constant array is a fancy array handle that has the same value in all of its entries. The constant array provides this array without actually using any memory.

Specifying a constant array in VTK-m is straightforward. VTK-m has a class named vtkm::cont::ArrayHandleConstant. ArrayHandleConstant is a templated class with a single template argument that is the type of value for each element in the array. The constructor for ArrayHandleConstant takes the value to provide by the array and the number of values the array should present. The following example is a simple demonstration of the constant array handle.

Example 26.1: Using ArrayHandleConstant.

```
// Create an array of 50 entries, all containing the number 3. This could be
// used, for example, to represent the sizes of all the polygons in a set
// where we know all the polygons are triangles.
vtkm::cont::ArrayHandleConstant<vtkm::Id> constantArray(3, 50);
```

The vtkm/cont/ArrayHandleConstant.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleConstant that takes a value and a size for the array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.2: Using make_ArrayHandleConstant.

```
  // Create an array of 50 entries, all containing the number 3.
  vtkm::cont::make_ArrayHandleConstant(3, 50)
```

26.2 ArrayHandleView

An array handle view is a fancy array handle that returns a subset of an already existing array handle. The array handle view uses the same memory as the existing array handle the view was created from. This means that changes to the data in the array handle view will also change the data in the original array handle.

To use the ArrayHandleView you must supply an ArrayHandle to the vtkm::cont::ArrayHandleView class constructor. ArrayHandleView is a templated class with a single template argument that is the ArrayHandle type of the array that the view is being created from. The constructor for ArrayHandleView takes a target array, starting index, and length. The follwing example shows a simple usage of the array handle view.

Example 26.3: Using ArrayHandleView.

```
vtkm::cont::ArrayHandle<vtkm::Id> sourceArray;
vtkm::cont::ArrayCopy(vtkm::cont::ArrayHandleIndex(10), sourceArray);
// sourceArray has [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

vtkm::cont::ArrayHandleView<vtkm::cont::ArrayHandle<vtkm::Id>> viewArray(
  sourceArray, 3, 5);
// viewArray has [3, 4, 5, 6, 7]
```

The vtkm/cont/ArrayHandleView.h header contains a templated convenience function vtkm::cont::make_ArrayHandleView that takes a target array, index, and length.

Example 26.4: Using make_ArrayHandleView.

```
  vtkm::cont::make_ArrayHandleView(sourceArray, 3, 5)
```

26.3 Counting Arrays

A counting array is a fancy array handle that provides a sequence of numbers. These fancy arrays can represent the data without actually using any memory.

VTK-m provides two versions of a counting array. The first version is an index array that provides a specialized but common form of a counting array called an index array. An index array has values of type vtkm::Id that start at 0 and count up by 1 (i.e. $0, 1, 2, 3, \ldots$). The index array mirrors the array's index.

Specifying an index array in VTK-m is done with a class named vtkm::cont::ArrayHandleIndex. The constructor for ArrayHandleIndex takes the size of the array to create. The following example is a simple demonstration of the index array handle.

Example 26.5: Using ArrayHandleIndex.

```
// Create an array containing [0, 1, 2, 3, ..., 49].
vtkm::cont::ArrayHandleIndex indexArray(50);
```

The vtkm::cont::ArrayHandleCounting class provides a more general form of counting. ArrayHandleCounting is a templated class with a single template argument that is the type of value for each element in the array. The constructor for ArrayHandleCounting takes three arguments: the start value (used at index 0), the step from one value to the next, and the length of the array. The following example is a simple demonstration of the counting array handle.

Example 26.6: Using ArrayHandleCounting.

```
// Create an array containing [-1.0, -0.9, -0.8, ..., 0.9, 1.0]
vtkm::cont::ArrayHandleCounting<vtkm::Float32> sampleArray(-1.0f, 0.1f, 21);
```

Did you know?

In addition to being simpler to declare, ArrayHandleIndex *is slightly faster than* ArrayHandleCounting*. Thus, when applicable, you should prefer using* ArrayHandleIndex*.*

The vtkm/cont/ArrayHandleCounting.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleCounting that also takes the start value, step, and length as arguments. This function can sometimes be used to avoid having to declare the full array type.

Example 26.7: Using make_ArrayHandleCounting.

```
  // Create an array of 50 entries, all containing the number 3.
  vtkm::cont::make_ArrayHandleCounting(-1.0f, 0.1f, 21)
```

There are no fundamental limits on how ArrayHandleCounting counts. For example, it is possible to count backwards.

Example 26.8: Counting backwards with ArrayHandleCounting.

```
// Create an array containing [49, 48, 47, 46, ..., 0].
vtkm::cont::ArrayHandleCounting<vtkm::Id> backwardIndexArray(49, -1, 50);
```

It is also possible to use ArrayHandleCounting to make sequences of vtkm::Vec values with piece-wise counting in each of the components.

Example 26.9: Using ArrayHandleCounting with vtkm::Vec objects.

```
  // Create an array containg [(0,-3,75), (1,2,25), (3,7,-25)]
  vtkm::cont::make_ArrayHandleCounting(
    vtkm::make_Vec(0, -3, 75), vtkm::make_Vec(1, 5, -50), 3)
```

26.4 Cast Arrays

A cast array is a fancy array that changes the type of the elements in an array. The cast array provides this re-typed array without actually copying or generating any data. Instead, casts are performed as the array is accessed.

VTK-m has a class named vtkm::cont::ArrayHandleCast to perform this implicit casting. ArrayHandleCast is a templated class with two template arguments. The first argument is the type to cast values to. The second argument is the type of the original ArrayHandle. The constructor to ArrayHandleCast takes the ArrayHandle to modify by casting.

Example 26.10: Using ArrayHandleCast.

```
template<typename T>
VTKM_CONT void Foo(const std::vector<T>& inputData)
{
  vtkm::cont::ArrayHandle<T> originalArray = vtkm::cont::make_ArrayHandle(inputData);

  vtkm::cont::ArrayHandleCast<vtkm::Float64, vtkm::cont::ArrayHandle<T>> castArray(
    originalArray);
```

The vtkm/cont/ArrayHandleCast.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleCast that constructs the cast array. The first argument is the original ArrayHandle original array to cast. The optional second argument is of the type to cast to (or you can optionally specify the cast-to type as a template argument.

Example 26.11: Using make_ArrayHandleCast.

```
    vtkm::cont::make_ArrayHandleCast<vtkm::Float64>(originalArray)
```

26.5 Discard Arrays

It is sometimes the case where you will want to run an operation in VTK-m that fills values in two (or more) arrays, but you only want the values that are stored in one of the arrays. It is possible to allocate space for both arrays and then throw away the values that you do not want, but that is a waste of memory. It is also possible to rewrite the functionality to output only what you want, but that is a poor use of developer time.

To solve this problem easily, VTK-m provides vtkm::cont::ArrayHandleDiscard. This array behaves similar to a regular ArrayHandle in that it can be "allocated" and has size, but any values that are written to it are immediately discarded. ArrayHandleDiscard takes up no memory.

Example 26.12: Using ArrayHandleDiscard.

```
template<typename InputArrayType,
         typename OutputArrayType1,
         typename OutputArrayType2>
VTKM_CONT void DoFoo(InputArrayType input,
                     OutputArrayType1 output1,
                     OutputArrayType2 output2);

template<typename InputArrayType>
VTKM_CONT inline vtkm::cont::ArrayHandle<vtkm::FloatDefault> DoBar(
  InputArrayType input)
{
  VTKM_IS_ARRAY_HANDLE(InputArrayType);

  vtkm::cont::ArrayHandle<vtkm::FloatDefault> keepOutput;

  vtkm::cont::ArrayHandleDiscard<vtkm::FloatDefault> discardOutput;

  DoFoo(input, keepOutput, discardOutput);

  return keepOutput;
}
```

26.6 Permuted Arrays

A permutation array is a fancy array handle that reorders the elements in an array. Elements in the array can be skipped over or replicated. The permutation array provides this reordered array without actually coping any data. Instead, indices are adjusted as the array is accessed.

Specifying a permutation array in VTK-m is straightforward. VTK-m has a class named vtkm::cont::ArrayHandlePermutation that takes two arrays: an array of values and an array of indices that maps an index in the permutation to an index of the original values. The index array is specified first. The following example is a simple demonstration of the permutation array handle.

Example 26.13: Using ArrayHandlePermutation.

```
using IdArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
using IdPortalType = IdArrayType::PortalControl;

using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Float64>;
using ValuePortalType = ValueArrayType::PortalControl;

// Create array with values [0.0, 0.1, 0.2, 0.3]
ValueArrayType valueArray;
valueArray.Allocate(4);
ValuePortalType valuePortal = valueArray.GetPortalControl();
valuePortal.Set(0, 0.0);
valuePortal.Set(1, 0.1);
valuePortal.Set(2, 0.2);
valuePortal.Set(3, 0.3);

// Use ArrayHandlePermutation to make an array = [0.3, 0.0, 0.1].
IdArrayType idArray1;
idArray1.Allocate(3);
IdPortalType idPortal1 = idArray1.GetPortalControl();
idPortal1.Set(0, 3);
idPortal1.Set(1, 0);
idPortal1.Set(2, 1);
vtkm::cont::ArrayHandlePermutation<IdArrayType, ValueArrayType> permutedArray1(
  idArray1, valueArray);

// Use ArrayHandlePermutation to make an array = [0.1, 0.2, 0.2, 0.3, 0.0]
IdArrayType idArray2;
idArray2.Allocate(5);
IdPortalType idPortal2 = idArray2.GetPortalControl();
idPortal2.Set(0, 1);
idPortal2.Set(1, 2);
idPortal2.Set(2, 2);
idPortal2.Set(3, 3);
idPortal2.Set(4, 0);
vtkm::cont::ArrayHandlePermutation<IdArrayType, ValueArrayType> permutedArray2(
  idArray2, valueArray);
```

The vtkm/cont/ArrayHandlePermutation.h header also contains the templated convenience function vtkm::cont::make_ArrayHandlePermutation that takes instances of the index and value array handles and returns a permutation array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.14: Using make_ArrayHandlePermutation.

```
    vtkm::cont::make_ArrayHandlePermutation(idArray, valueArray)
```

Common Errors

When using an ArrayHandlePermutation, take care that all the provided indices in the index array point to valid locations in the values array. Bad indices can cause reading from or writing to invalid memory locations, which can be difficult to debug.

Did you know?

You can write to a ArrayHandlePermutation *by, for example, using it as an output array. Writes to the* ArrayHandlePermutation *will go to the respective location in the source array. However,* ArrayHandlePermutation *cannot be resized.*

26.7 Zipped Arrays

A zip array is a fancy array handle that combines two arrays of the same size to pair up the corresponding values. Each element in the zipped array is a `vtkm::Pair` containing the values of the two respective arrays. These pairs are not stored in their own memory space. Rather, the pairs are generated as the array is used. Writing a pair to the zipped array writes the values in the two source arrays.

Specifying a zipped array in VTK-m is straightforward. VTK-m has a class named `vtkm::cont::ArrayHandleZip` that takes the two arrays providing values for the first and second entries in the pairs. The following example is a simple demonstration of creating a zip array handle.

Example 26.15: Using `ArrayHandleZip`.

```
using ArrayType1 = vtkm::cont::ArrayHandle<vtkm::Id>;
using PortalType1 = ArrayType1::PortalControl;

using ArrayType2 = vtkm::cont::ArrayHandle<vtkm::Float64>;
using PortalType2 = ArrayType2::PortalControl;

// Create an array of vtkm::Id with values [3, 0, 1]
ArrayType1 array1;
array1.Allocate(3);
PortalType1 portal1 = array1.GetPortalControl();
portal1.Set(0, 3);
portal1.Set(1, 0);
portal1.Set(2, 1);

// Create a second array of vtkm::Float32 with values [0.0, 0.1, 0.2]
ArrayType2 array2;
array2.Allocate(3);
PortalType2 portal2 = array2.GetPortalControl();
portal2.Set(0, 0.0);
portal2.Set(1, 0.1);
portal2.Set(2, 0.2);

// Zip the two arrays together to create an array of
// vtkm::Pair<vtkm::Id, vtkm::Float64> with values [(3,0.0), (0,0.1), (1,0.2)]
vtkm::cont::ArrayHandleZip<ArrayType1, ArrayType2> zipArray(array1, array2);
```

The vtkm/cont/ArrayHandleZip.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandleZip` that takes instances of the two array handles and returns a zip array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.16: Using `make_ArrayHandleZip`.

```
    vtkm::cont::make_ArrayHandleZip(array1, array2)
```

26.8 Coordinate System Arrays

Many of the data structures we use in VTK-m are described in a 3D coordinate system. Although, as we will see in Chapter 7, we can use any ArrayHandle to store point coordinates, including a raw array of 3D vectors, there are some common patterns for point coordinates that we can use specialized arrays to better represent the data.

There are two fancy array handles that each handle a special form of coordinate system. The first such array handle is vtkm::cont::ArrayHandleUniformPointCoordinates, which represents a uniform sampling of space. The constructor for ArrayHandleUniformPointCoordinates takes three arguments. The first argument is a vtkm::Id3 that specifies the number of samples in the x, y, and z directions. The second argument, which is optional, specifies the origin (the location of the first point at the lower left corner). If not specified, the origin is set to $[0,0,0]$. The third argument, which is also optional, specifies the distance between samples in the x, y, and z directions. If not specified, the spacing is set to 1 in each direction.

Example 26.17: Using ArrayHandleUniformPointCoordinates.

```
// Create a set of point coordinates for a uniform grid in the space between
// -5 and 5 in the x direction and -3 and 3 in the y and z directions. The
// uniform sampling is spaced in 0.08 unit increments in the x direction (for
// 126 samples), 0.08 unit increments in the y direction (for 76 samples) and
// 0.24 unit increments in the z direction (for 26 samples). That makes
// 248,976 values in the array total.
vtkm::cont::ArrayHandleUniformPointCoordinates uniformCoordinates(
  vtkm::Id3(126, 76, 26),
  vtkm::Vec3f{ -5.0f, -3.0f, -3.0f },
  vtkm::Vec3f{ 0.08f, 0.08f, 0.24f });
```

The second fancy array handle for special coordinate systems is vtkm::cont::ArrayHandleCartesianProduct, which represents a rectilinear sampling of space where the samples are axis aligned but have variable spacing. Sets of coordinates of this type are most efficiently represented by having a separate array for each component of the axis, and then for each $[i,j,k]$ index of the array take the value for each component from each array using the respective index. This is equivalent to performing a Cartesian product on the arrays.

ArrayHandleCartesianProduct is a templated class. It has three template parameters, which are the types of the arrays used for the x, y, and z axes. The constructor for ArrayHandleCartesianProduct takes the three arrays.

Example 26.18: Using a ArrayHandleCartesianProduct.

```
using AxisArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
using AxisPortalType = AxisArrayType::PortalControl;

// Create array for x axis coordinates with values [0.0, 1.1, 5.0]
AxisArrayType xAxisArray;
xAxisArray.Allocate(3);
AxisPortalType xAxisPortal = xAxisArray.GetPortalControl();
xAxisPortal.Set(0, 0.0f);
xAxisPortal.Set(1, 1.1f);
xAxisPortal.Set(2, 5.0f);

// Create array for y axis coordinates with values [0.0, 2.0]
AxisArrayType yAxisArray;
yAxisArray.Allocate(2);
AxisPortalType yAxisPortal = yAxisArray.GetPortalControl();
yAxisPortal.Set(0, 0.0f);
yAxisPortal.Set(1, 2.0f);

// Create array for z axis coordinates with values [0.0, 0.5]
AxisArrayType zAxisArray;
zAxisArray.Allocate(2);
```

```
AxisPortalType zAxisPortal = zAxisArray.GetPortalControl();
zAxisPortal.Set(0, 0.0f);
zAxisPortal.Set(1, 0.5f);

// Create point coordinates for a "rectilinear grid" with axis-aligned points
// with variable spacing by taking the Cartesian product of the three
// previously defined arrays. This generates the following 3x2x2 = 12 values:
//
// [0.0, 0.0, 0.0], [1.1, 0.0, 0.0], [5.0, 0.0, 0.0],
// [0.0, 2.0, 0.0], [1.1, 2.0, 0.0], [5.0, 2.0, 0.0],
// [0.0, 0.0, 0.5], [1.1, 0.0, 0.5], [5.0, 0.0, 0.5],
// [0.0, 2.0, 0.5], [1.1, 2.0, 0.5], [5.0, 2.0, 0.5]
vtkm::cont::
  ArrayHandleCartesianProduct<AxisArrayType, AxisArrayType, AxisArrayType>
    rectilinearCoordinates(xAxisArray, yAxisArray, zAxisArray);
```

The vtkm/cont/ArrayHandleCartesianProduct.h/header also contains the templated convenience function `vtkm::cont::make_ArrayHandleCartesianProduct` that takes the three axis arrays and returns an array of the Cartesian product. This function can sometimes be used to avoid having to declare the full array type.

Example 26.19: Using `make_ArrayHandleCartesianProduct`.

```
  vtkm::cont::make_ArrayHandleCartesianProduct(xAxisArray, yAxisArray, zAxisArray)
```

Did you know?

These specialized arrays for coordinate systems greatly reduce the code duplication in VTK-m. Most scientific visualization systems need separate implementations of algorithms for uniform, rectilinear, and unstructured grids. But in VTK-m an algorithm can be written once and then applied to all these different grid structures by using these specialized array handles and letting the compiler's templates optimize the code.

26.9 Composite Vector Arrays

A composite vector array is a fancy array handle that combines two to four arrays of the same size and value type and combines their corresponding values to form a `vtkm::Vec`. A composite vector array is similar in nature to a zipped array (described in Section 26.7) except that values are combined into `vtkm::Vec` s instead of `vtkm::Pair` s. The created `vtkm::Vec` s are not stored in their own memory space. Rather, the `Vecs` are generated as the array is used. Writing `Vecs` to the composite vector array writes values into the components of the source arrays.

A composite vector array can be created using the `vtkm::cont::ArrayHandleCompositeVector` class. This class has a variadic template argument that is a "signature" for the arrays to be combined. The constructor for `ArrayHandleCompositeVector` takes instances of the array handles to combine.

Example 26.20: Using `ArrayHandleCompositeVector`.

```
// Create an array with [0, 1, 2, 3, 4]
using ArrayType1 = vtkm::cont::ArrayHandleIndex;
ArrayType1 array1(5);

// Create an array with [3, 1, 4, 1, 5]
using ArrayType2 = vtkm::cont::ArrayHandle<vtkm::Id>;
ArrayType2 array2;
```

```
array2.Allocate(5);
ArrayType2::PortalControl arrayPortal2 = array2.GetPortalControl();
arrayPortal2.Set(0, 3);
arrayPortal2.Set(1, 1);
arrayPortal2.Set(2, 4);
arrayPortal2.Set(3, 1);
arrayPortal2.Set(4, 5);

// Create an array with [2, 7, 1, 8, 2]
using ArrayType3 = vtkm::cont::ArrayHandle<vtkm::Id>;
ArrayType3 array3;
array3.Allocate(5);
ArrayType2::PortalControl arrayPortal3 = array3.GetPortalControl();
arrayPortal3.Set(0, 2);
arrayPortal3.Set(1, 7);
arrayPortal3.Set(2, 1);
arrayPortal3.Set(3, 8);
arrayPortal3.Set(4, 2);

// Create an array with [0, 0, 0, 0]
using ArrayType4 = vtkm::cont::ArrayHandleConstant<vtkm::Id>;
ArrayType4 array4(0, 5);

// Use ArrayhandleCompositeVector to create the array
// [(0,3,2,0), (1,1,7,0), (2,4,1,0), (3,1,8,0), (4,5,2,0)].
using CompositeArrayType = vtkm::cont::
  ArrayHandleCompositeVector<ArrayType1, ArrayType2, ArrayType3, ArrayType4>;
CompositeArrayType compositeArray(array1, array2, array3, array4);
```

The vtkm/cont/ArrayHandleCompositeVector.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandleCompositeVector` which takes a variable number of array handles and returns an `ArrayHandleCompositeVector`. This function can sometimes be used to avoid having to declare the full array type. `ArrayHandleCompositeVector` is also often used to combine scalar arrays into vector arrays.

Example 26.21: Using `make_ArrayHandleCompositeVector`.

```
  vtkm::cont::make_ArrayHandleCompositeVector(array1, array2, array3, array4)
```

26.10 Extract Component Arrays

Component extraction allows access to a single component of an `ArrayHandle` with a `vtkm::Vec` ValueType. `vtkm::cont::ArrayHandleExtractComponent` allows one component of a vector array to be extracted without creating a copy of the data. `ArrayHandleExtractComponent` can also be combined with `ArrayHandleCompositeVector` (described in Section 26.9) to arbitrarily stitch several components from multiple arrays together.

As a simple example, consider an `ArrayHandle` containing 3D coordinates for a collection of points and a filter that only operates on the points' elevations (Z, in this example). We can easily create the elevation array on-the-fly without allocating a new array as in the following example.

Example 26.22: Extracting components of `Vec`s in an array with `ArrayHandleExtractComponent`.

```
using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec3f_64>;

// Create array with values [ (0.0, 0.1, 0.2), (1.0, 1.1, 1.2), (2.0, 2.1, 2.2) ]
ValueArrayType valueArray;
valueArray.Allocate(3);
auto valuePortal = valueArray.GetPortalControl();
valuePortal.Set(0, vtkm::make_Vec(0.0, 0.1, 0.2));
valuePortal.Set(1, vtkm::make_Vec(1.0, 1.1, 1.2));
valuePortal.Set(2, vtkm::make_Vec(2.0, 2.1, 2.2));
```

```

  // Use ArrayHandleExtractComponent to make an array = [1.3, 2.3, 3.3].
  vtkm::cont::ArrayHandleExtractComponent<ValueArrayType> extractedComponentArray(
    valueArray, 2);
```

The vtkm/cont/ArrayHandleExtractComponent.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleExtractComponent that takes an ArrayHandle of Vecs and vtkm::IdComponent which returns an appropriately typed ArrayHandleExtractComponent containing the values for a specified component. The index of the component to extract is provided as an argument to make_ArrayHandleExtractComponent, which is required. The use of make_ArrayHandleExtractComponent can be used to avoid having to declare the full array type.

Example 26.23: Using make_ArrayHandleExtractComponent.

```
    vtkm::cont::make_ArrayHandleExtractComponent(valueArray, 2)
```

26.11 Swizzle Arrays

It is often useful to reorder or remove specific components from an ArrayHandle with a vtkm::Vec ValueType. vtkm::cont::ArrayHandleSwizzle provides an easy way to accomplish this.

The template parameters of ArrayHandleSwizzle specify a "component map," which defines the swizzle operation. This map consists of the components from the input ArrayHandle, which will be exposed in the ArrayHandleSwizzle. For instance, vtkm::cont::ArrayHandleSwizzle <Some3DArrayType, 3> with Some3DArray and vtkm::Vec <vtkm::IdComponent, 3>(0, 2, 1) as constructor arguments will allow access to a 3D array, but with the Y and Z components exchanged. This rearrangement does not create a copy, and occurs on-the-fly as data are accessed through the ArrayHandleSwizzle's portal. This fancy array handle can also be used to eliminate unnecessary components from an ArrayHandle's data, as shown below.

Example 26.24: Swizzling components of Vecs in an array with ArrayHandleSwizzle.

```
  using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec4f_64>;

  // Create array with values
  // [ (0.0, 0.1, 0.2, 0.3), (1.0, 1.1, 1.2, 1.3), (2.0, 2.1, 2.2, 2.3) ]
  ValueArrayType valueArray;
  valueArray.Allocate(3);
  auto valuePortal = valueArray.GetPortalControl();
  valuePortal.Set(0, vtkm::make_Vec(0.0, 0.1, 0.2, 0.3));
  valuePortal.Set(1, vtkm::make_Vec(1.0, 1.1, 1.2, 1.3));
  valuePortal.Set(2, vtkm::make_Vec(2.0, 2.1, 2.2, 2.3));

  // Use ArrayHandleSwizzle to make an array of Vec-3 with x,y,z,w swizzled to z,x,w
  // [ (0.2, 0.0, 0.3), (1.2, 1.0, 1.3), (2.2, 2.0, 2.3) ]
  vtkm::cont::ArrayHandleSwizzle<ValueArrayType, 3> swizzledArray(
    valueArray, vtkm::IdComponent3(2, 0, 3));
```

The vtkm/cont/ArrayHandleSwizzle.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleSwizzle that takes an ArrayHandle of Vecs and returns an appropriately typed ArrayHandleSwizzle containing swizzled vectors. The indices of the swizzled components are provided as arguments to make_ArrayHandleSwizzle after the ArrayHandle. The use of make_ArrayHandleSwizzle can be used to avoid having to declare the full array type.

Example 26.25: Using make_ArrayHandleSwizzle.

```
    vtkm::cont::make_ArrayHandleSwizzle(valueArray, 2, 0, 3)
```

26.12 Grouped Vector Arrays

A grouped vector array is a fancy array handle that groups consecutive values of an array together to form a vtkm::Vec. The source array must be of a length that is divisible by the requested Vec size. The created vtkm::Vec s are not stored in their own memory space. Rather, the Vecs are generated as the array is used. Writing Vecs to the grouped vector array writes values into the the source array.

A grouped vector array is created using the vtkm::cont::ArrayHandleGroupVec class. This templated class has two template arguments. The first argument is the type of array being grouped and the second argument is an integer specifying the size of the Vecs to create (the number of values to group together).

Example 26.26: Using ArrayHandleGroupVec.

```
// Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
using ArrayType = vtkm::cont::ArrayHandleIndex;
ArrayType sourceArray(12);

// Create an array containing [(0,1), (2,3), (4,5), (6,7), (8,9), (10,11)]
vtkm::cont::ArrayHandleGroupVec<ArrayType, 2> vec2Array(sourceArray);

// Create an array containing [(0,1,2), (3,4,5), (6,7,8), (9,10,11)]
vtkm::cont::ArrayHandleGroupVec<ArrayType, 3> vec3Array(sourceArray);
```

The vtkm/cont/ArrayHandleGroupVec.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleGroupVec that takes an instance of the array to group into Vecs. You must specify the size of the Vecs as a template parameter when using vtkm::cont::make_ArrayHandleGroupVec.

Example 26.27: Using make_ArrayHandleGroupVec.

```
  // Create an array containing [(0,1,2,3), (4,5,6,7), (8,9,10,11)]
  vtkm::cont::make_ArrayHandleGroupVec<4>(sourceArray)
```

ArrayHandleGroupVec is handy when you need to build an array of vectors that are all of the same length, but what about when you need an array of vectors of different lengths? One common use case for this is if you are defining a collection of polygons of different sizes (triangles, quadrilaterals, pentagons, and so on). We would like to define an array such that the data for each polygon were stored in its own Vec (or, rather, Vec-like) object. vtkm::cont::ArrayHandleGroupVecVariable does just that.

ArrayHandleGroupVecVariable takes two arrays. The first array, identified as the “source” array, is a flat representation of the values (much like the array used with ArrayHandleGroupVec). The second array, identified as the “offsets” array, provides for each vector the index into the source array where the start of the vector is. The offsets array must be monotonically increasing. The first and second template parameters to ArrayHandleGroupVecVariable are the types for the source and offset arrays, respectively.

It is often the case that you will start with a group of vector lengths rather than offsets into the source array. If this is the case, then the vtkm::cont::ConvertNumComponentsToOffsets helper function can convert an array of vector lengths to an array of offsets. The first argument to this function is always the array of vector lengths. The second argument, which is optional, is a reference to a ArrayHandle into which the offsets should be stored. If this offset array is not specified, an ArrayHandle will be returned from the function instead. The third argument, which is also optional, is a reference to a vtkm::Id into which the expected size of the source array is put. Having the size of the source array is often helpful, as it can be used to allocate data for the source array or check the source array’s size. It is also OK to give the expected size reference but not the offset array reference.

Example 26.28: Using ArrayHandleGroupVecVariable.

```
// Create an array of counts containing [4, 2, 3, 3]
vtkm::IdComponent countBuffer[4] = { 4, 2, 3, 3 };
vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray =
  vtkm::cont::make_ArrayHandle(countBuffer, 4);

// Convert the count array to an offset array [0, 4, 6, 9]
// Returns the number of total components: 12
vtkm::Id sourceArraySize;
using OffsetArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
OffsetArrayType offsetArray =
  vtkm::cont::ConvertNumComponentsToOffsets(countArray, sourceArraySize);

// Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
using SourceArrayType = vtkm::cont::ArrayHandleIndex;
SourceArrayType sourceArray(sourceArraySize);

// Create an array containing [(0,1,2,3), (4,5), (6,7,8), (9,10,11)]
vtkm::cont::ArrayHandleGroupVecVariable<SourceArrayType, OffsetArrayType>
  vecVariableArray(sourceArray, offsetArray);
```

The vtkm/cont/ArrayHandleGroupVecVariable.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleGroupVecVariable that takes an instance of the source array to group into Vec-like objects and the offset array.

Example 26.29: Using MakeArrayHandleGroupVecVariable.

```
    // Create an array containing [(0,1,2,3), (4,5), (6,7,8), (9,10,11)]
    vtkm::cont::make_ArrayHandleGroupVecVariable(sourceArray, offsetArray)
```

Did you know?

You can write to ArrayHandleGroupVec and ArrayHandleGroupVecVariable by, for example, using it as an output array. Writes to these arrays will go to the respective location in the source array. ArrayHandleGroupVec can also be allocated and resized (which in turn causes the source array to be allocated). However, ArrayHandleGroupVecVariable cannot be resized and the source array must be pre-allocated. You can use the source array size value returned from ConvertNumComponentsToOffsets to allocate source arrays.

Common Errors

Keep in mind that the values stored in a ArrayHandleGroupVecVariable are not actually vtkm::Vec objects. Rather, they are "Vec-like" objects, which has some subtle but important ramifications. First, the type will not match the vtkm::Vec template, and there is no automatic conversion to vtkm::Vec objects. Thus, many functions that accept vtkm::Vec objects as parameters will not accept the Vec-like object. Second, the size of Vec-like objects are not known until runtime. See Sections 4.3 and 19.6.2 for more information on the difference between vtkm::Vec and Vec-like objects.

26.13 Virtual Arrays

One of the complications that all the variations to array handle described earlier in this chapter introduces is that the actual type of the array might not be known. That can be problematic when writing functions or methods that operate on arrays. Often this issue can be resolved by simply making a templated argument that accepts any object that looks like an ArrayHandle. VTK-m provides the macro VTKM_IS_ARRAY_HANDLE to verify that a template type is in fact an array handle.

Example 26.30: Using templates for generic array handles.

```
// NOTE: There are faster ways to sum large arrays in VTK-m.
template<typename ArrayHandleType>
VTKM_CONT vtkm::Float64 SumArrayHandle(const ArrayHandleType& arrayHandle)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);

  typename ArrayHandleType::PortalConstControl portal =
    arrayHandle.GetPortalConstControl();
  vtkm::Float64 sum = 0.0;
  for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); ++index)
  {
    sum += portal.Get(index);
  }

  return sum;
}
```

However, in some cases using a template in this way is not feasible. For example, what if you are calling a virtual method, which cannot be practically templated like this? Or what if you need to store the arrays in a secondary object that cannot be practically templated on all possible array types? Or what if you need to return an array handle, but you do not know the specific type of array handle until runtime?

Example 26.31: A problem that can occur when an array handle type is not known.

```
  std::vector<vtkm::cont::ArrayHandle<vtkm::Float64>> vectorOfArrays;

  // Make basic array.
  vtkm::cont::ArrayHandle<vtkm::Float64> basicArray;
  // Fill basicArray...
  vectorOfArrays.push_back(basicArray); // Works fine
  // The previous line works fine because you are passing a standard ArrayHandle
  // to a method that expects a standard ArrayHandle of the same type.

  // Make fancy array.
  vtkm::cont::ArrayHandleCounting<vtkm::Float64> fancyArray(-1.0, 0.1, ARRAY_SIZE);
  vectorOfArrays.push_back(fancyArray); // ERROR!!!!
  // The previous line fails to compile because it is passing an ArrayHandleCounting
  // to a method that expects a standard ArrayHandle, and you cannot directly make
  // this cast.
```

To get around this problem, VTK-m provides the `vtkm::cont::ArrayHandleVirtual` class. `ArrayHandleVirtual` is a special type of array handle that can be wrapped around a `ArrayHandle`, any of the fancy array handles described this chapter, or any other possible custom array that can be created.

`ArrayHandleVirtual` can be used like any other array handle.

Example 26.32: Using an `ArrayHandleVirtual`.

```
VTKM_CONT std::vector<vtkm::Float64> SumSeveralArrayHandles(
  const std::vector<vtkm::cont::ArrayHandleVirtual<vtkm::Float64>>& vectorOfArrays)
{
  std::vector<vtkm::Float64> sums;
  for (auto&& arrayHandle : vectorOfArrays)
  {
    sums.push_back(SumArrayHandle(arrayHandle));
  }

  return sums;
}

VTKM_CONT void DoStuff()
{
  std::vector<vtkm::cont::ArrayHandleVirtual<vtkm::Float64>> vectorOfArrays;

  // Make basic array.
  vtkm::cont::ArrayHandle<vtkm::Float64> basicArray;
  // Fill basicArray...
  vectorOfArrays.push_back(basicArray);

  // Make fancy array.
  vtkm::cont::ArrayHandleCounting<vtkm::Float64> fancyArray(-1.0, 0.1, ARRAY_SIZE);
  vectorOfArrays.push_back(fancyArray);

  std::vector<vtkm::Float64> sums = SumSeveralArrayHandles(vectorOfArrays);
}
```

Did you know?

`ArrayHandleVirtual` can be used when you do not know what kind of array you are working with. However, you still need to know the type of value stored in the array (floating point, integer, vector, ect.). `vtkm::cont::VariantArrayHandle`, described in Chapter 33, can instead be used in the case where you do not know the type of values in the array.

The `ArrayHandleVirtual` class also comes with some special methods to work with types that are not known until runtime.

`NewInstance` Creates a new array of the same type.

`IsType` Given an array handle type, returns true if the array stored in the `ArrayHandleVirtual` is the same type as the one given.

`Cast` Given an array handle type, casts the array to that type and returns it. If the stored array is of the wrong type, and exception is thrown.

One common use case for querying the type stored in a `ArrayHandleVirtual` is to create "fast paths" for common types. The following example demonstrates using casting to create a fast path for a basic `ArrayHandle` but also providing a fallback using the virtual interface, which may be slower due to calling virtual methods to get values.

Example 26.33: Casting a `ArrayHandleVirtual` to a known type.

```
VTKM_CONT vtkm::Float64 SumArrayHandleVirtual(
  const vtkm::cont::ArrayHandleVirtual<vtkm::Float64>& virtualArray)
{
  if (virtualArray.IsType<vtkm::cont::ArrayHandle<vtkm::Float64>>())
  {
    // Fast path for basic array storage (direct access to memory)
    vtkm::cont::ArrayHandle<vtkm::Float64> basicArray =
      virtualArray.Cast<vtkm::cont::ArrayHandle<vtkm::Float64>>();
    return SumArrayHandle(basicArray);
  }
  else
  {
    // Slower path to go through general virtual interface
    return SumArrayHandle(virtualArray);
  }
}
```

26.13.1 Virtual Coordinates

There may be instances in which the types for a given coordinate system are not known. In this case, it is possible to use a special implementation of vtkm::cont::ArrayHandleVirtual called vtkm::cont::ArrayHandleVirtualCoordinates when interacting with 3D datasets with array types. Similar to how any vtkm::cont::ArrayHandle can be used to store point coordinates, any vtkm::cont::ArrayHandleVirtual can similarly store point coordinates of any type. The vtkm::cont::ArrayHandleVirtualCoordinates exists to support easier to understand code and better represent the underlying data.

Example 26.34: Using ArrayHandleVirtualCoordinates.

```
static constexpr vtkm::Id ARRAY_SIZE = 10;

VTKM_CONT std::vector<vtkm::FloatDefault> CountSeveralCoordinates(
  const std::vector<vtkm::cont::ArrayHandleVirtualCoordinates>& vectorOfCoordinates)
{
  std::vector<vtkm::FloatDefault> counts;
  for (auto&& arrayHandleCoordinate : vectorOfCoordinates)
  {
    counts.push_back(arrayHandleCoordinate.GetNumberOfValues());
  }
  return counts;
}

VTKM_CONT void CountArrays()
{
  using CPType = vtkm::cont::ArrayHandleCounting<vtkm::FloatDefault>;
  vtkm::cont::ArrayHandleCounting<vtkm::Vec3f> one(
    vtkm::Vec3f{ 0, 0, 0 }, vtkm::Vec3f{ 1, 1, 1 }, ARRAY_SIZE);
  vtkm::cont::ArrayHandleUniformPointCoordinates two(vtkm::Id3(10, 10, 10));
  vtkm::cont::ArrayHandleCounting<vtkm::FloatDefault> c1(0, 1, ARRAY_SIZE);
  vtkm::cont::ArrayHandleCounting<vtkm::FloatDefault> c2(0, 1, ARRAY_SIZE);
  vtkm::cont::ArrayHandleCounting<vtkm::FloatDefault> c3(0, 1, ARRAY_SIZE);
  vtkm::cont::ArrayHandleCartesianProduct<CPType, CPType, CPType> three(c1, c2, c3);

  std::vector<vtkm::cont::ArrayHandleVirtualCoordinates> vectorOfCoordinates;
  vectorOfCoordinates.emplace_back(one);
  vectorOfCoordinates.emplace_back(two);
  vectorOfCoordinates.emplace_back(three);

  std::vector<vtkm::FloatDefault> elements =
    CountSeveralCoordinates(vectorOfCoordinates);
}
```

Common Errors

The constructor for vtkm::cont::ArrayHandleVirtualCoordinates *is declared* explicit *This means that we cannot implicitly convert a type that could resolve to a* vtkm::cont::ArrayHandleVirtualCoordinates *object like the* vtkm::cont::ArrayHandleVirtual *can. All* vtkm::cont::ArrayHandleVirtualCordinates *must be explicitly constructed, as the above example shows by calling* emplace_back *as opposed to* push_back*.*

Common Errors

It should be noted that `vtkm::cont::ArrayHandleVirtualCoordinates` *is not templated to accept different types. It directly inherits from the corresponding* `vtkm::cont::ArrayHandleVirtual` *whose underlying type is a* `vtkm::Vec3f`*. Because of this, attempting to use a more precise type (like a double) with* `vtkm::cont::ArrayHandleVirtualCoordinates` *will result in a loss of precision.*

Example 26.35: Bad Cast `ArrayHandleVirtualCoordinates`.

```
VTKM_CONT void BadCast()
{
  vtkm::cont::ArrayHandleCounting<vtkm::Vec3f_64> one(
    vtkm::Vec3f_64{ 0, 0, 0 }, vtkm::Vec3f_64{ 1, 1, 1 }, ARRAY_SIZE);
  std::vector<vtkm::cont::ArrayHandleVirtualCoordinates> vectorOfCoordinates;
  vectorOfCoordinates.emplace_back(one); // OH NO! WE LOST PRECISION!
}
```

CHAPTER

TWENTYSEVEN

ACCESSING AND ALLOCATING ARRAY HANDLES

So far we have seen examples of creating `vtkm::cont::ArrayHandle` s from normal C++ arrays (Chapter 16) and creating some special arrays (Chapter 26). However, we have so far avoided discussing how to access the actual data in `ArrayHandle`s. So far we have only accessed `ArrayHandle` data indirectly through other VTK-m features such as worklets.

In this chapter we describe how to more directly access the data in an `ArrayHandle`, how to allocate space for data in an `ArrayHandle`, and how data is transferred to the execution environment to be used in a worklet.

27.1 Array Portals

An array handle defines auxiliary structures called *array portals* that provide direct access into its data. An array portal is a simple object that is somewhat functionally equivalent to an STL-type iterator, but with a much simpler interface. Array portals can be read-only (const) or read-write and they can be accessible from either the control environment or the execution environment. All these variants have similar interfaces although some features that are not applicable can be left out.

An array portal object contains each of the following:

`ValueType` The type for each item in the array.

`GetNumberOfValues` A method that returns the number of entries in the array.

`Get` A method that returns the value at a given index.

`Set` A method that changes the value at a given index. This method does not need to exist for read-only (const) array portals.

The following code example defines an array portal for a simple C array of scalar values. This definition has no practical value (it is covered by the more general `vtkm::cont::internal::ArrayPortalFromIterators`), but demonstrates the function of each component.

Example 27.1: A simple array portal implementation.

```
template<typename T>
class SimpleScalarArrayPortal
{
public:
  using ValueType = T;

  // There is no specification for creating array portals, but they generally
  // need a constructor like this to be practical.
  VTKM_EXEC_CONT
  SimpleScalarArrayPortal(ValueType* array, vtkm::Id numberOfValues)
    : Array(array)
    , NumberOfValues(numberOfValues)
  {
  }

  VTKM_EXEC_CONT
  SimpleScalarArrayPortal()
    : Array(NULL)
    , NumberOfValues(0)
  {
  }

  VTKM_EXEC_CONT
  vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }

  VTKM_EXEC_CONT
  ValueType Get(vtkm::Id index) const { return this->Array[index]; }

  VTKM_EXEC_CONT
  void Set(vtkm::Id index, ValueType value) const { this->Array[index] = value; }

private:
  ValueType* Array;
  vtkm::Id NumberOfValues;
};
```

Although array portals are simple to implement and use, and array portals' functionality is similar to iterators, there exists a great deal of code already based on STL iterators and it is often convienient to interface with an array through an iterator rather than an array portal. The vtkm::cont::ArrayPortalToIterators class can be used to convert an array portal to an STL-compatible iterator. The class is templated on the array portal type and has a constructor that accepts an instance of the array portal. It contains the following features.

IteratorType The type of an STL-compatible random-access iterator that can provide the same access as the array portal.

GetBegin A method that returns an STL-compatible iterator of type IteratorType that points to the beginning of the array.

GetEnd A method that returns an STL-compatible iterator of type IteratorType that points to the end of the array.

Example 27.2: Using ArrayPortalToIterators.

```
template<typename PortalType>
VTKM_CONT std::vector<typename PortalType::ValueType> CopyArrayPortalToVector(
  const PortalType& portal)
{
  using ValueType = typename PortalType::ValueType;
  std::vector<ValueType> result(
    static_cast<std::size_t>(portal.GetNumberOfValues()));

  vtkm::cont::ArrayPortalToIterators<PortalType> iterators(portal);

  std::copy(iterators.GetBegin(), iterators.GetEnd(), result.begin());

  return result;
}
```

As a convenience, vtkm/cont/ArrayPortalToIterators.h also defines a pair of functions named `vtkm::cont::ArrayPortalToIteratorBegin()` and `vtkm::cont::ArrayPortalToIteratorEnd()` that each take an array portal as an argument and return a begin and end iterator, respectively.

Example 27.3: Using `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd`.

```
  std::vector<vtkm::Float32> myContainer(
    static_cast<std::size_t>(portal.GetNumberOfValues()));

  std::copy(vtkm::cont::ArrayPortalToIteratorBegin(portal),
            vtkm::cont::ArrayPortalToIteratorEnd(portal),
            myContainer.begin());
```

`ArrayHandle` contains two internal type definitions for array portal types that are capable of interfacing with the underlying data in the control environment. These are `PortalControl` and `PortalConstControl`, which define read-write and read-only (const) array portals, respectively.

`ArrayHandle` also contains similar type definitions for array portals in the execution environment. Because these types are dependent on the device adapter used for execution, these type definitions are embedded in a templated class named `ExecutionTypes`. Within `ExecutionTypes` are the type definitions `Portal` and `PortalConst` defining the read-write and read-only (const) array portals, respectively, for the execution environment for the given device adapter tag.

Because `vtkm::cont::ArrayHandle` is control environment object, it provides the methods `GetPortalControl` and `GetPortalConstControl` to get the associated array portal objects. These methods also have the side effect of refreshing the control environment copy of the data as if you called `SyncControlArray`. Be aware that calling `GetPortalControl` will invalidate any copy in the execution environment, meaning that any subsequent use will cause the data to be copied back again.

Example 27.4: Using portals from an `ArrayHandle`.

```
template<typename T, typename Storage>
void SortCheckArrayHandle(vtkm::cont::ArrayHandle<T, Storage> arrayHandle)
{
  using PortalType = typename vtkm::cont::ArrayHandle<T, Storage>::PortalControl;
  using PortalConstType =
    typename vtkm::cont::ArrayHandle<T, Storage>::PortalConstControl;

  PortalType readwritePortal = arrayHandle.GetPortalControl();
  // This is actually pretty dumb. Sorting would be generally faster in
  // parallel in the execution environment using the device adapter algorithms.
  std::sort(vtkm::cont::ArrayPortalToIteratorBegin(readwritePortal),
            vtkm::cont::ArrayPortalToIteratorEnd(readwritePortal));

  PortalConstType readPortal = arrayHandle.GetPortalConstControl();
  for (vtkm::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
  {
    if (readPortal.Get(index - 1) > readPortal.Get(index))
    {
      std::cout << "Sorting is wrong!" << std::endl;
      break;
    }
  }
}
```

Did you know?

Most operations on arrays in VTK-m should really be done in the execution environment. Keep in mind that whenever doing an operation using a control array portal, that operation will likely be slow for large arrays. However, some operations, like performing file I/O, make sense in the control environment.

27.2 Allocating and Populating Array Handles

`vtkm::cont::ArrayHandle` is capable of allocating its own memory. The most straightforward way to allocate memory is to call the `ArrayHandle::Allocate` method. The `Allocate` method takes a single argument, which is the number of elements to make the array.

Example 27.5: Allocating an `ArrayHandle`.

```
vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;

const vtkm::Id ARRAY_SIZE = 50;
arrayHandle.Allocate(ARRAY_SIZE);
```

Common Errors

The ability to allocate memory is a key difference between `ArrayHandle` *and many other common forms of smart pointers. When one* `ArrayHandle` *allocates new memory, all other* `ArrayHandle`*s pointing to the same managed memory get the newly allocated memory. This can be particularly surprising when the originally managed memory is empty. For example, older versions of* `std::vector` *initialized all its values by setting them to the same object. When a* `vector` *of* `ArrayHandle`*s was created and one entry was allocated, all entries changed to the same allocation.*

Once an `ArrayHandle` is allocated, it can be populated by using the portal returned from `ArrayHandle::GetPortalControl`, as described in Section 27.1. This is roughly the method used by the readers in the I/O package (Chapter 8).

Example 27.6: Populating a newly allocated `ArrayHandle`.

```
vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;

const vtkm::Id ARRAY_SIZE = 50;
arrayHandle.Allocate(ARRAY_SIZE);

using PortalType = vtkm::cont::ArrayHandle<vtkm::Float32>::PortalControl;
PortalType portal = arrayHandle.GetPortalControl();

for (vtkm::Id index = 0; index < ARRAY_SIZE; index++)
{
  portal.Set(index, GetValueForArray(index));
}
```

27.3 Compute Array Range

It is common to need to know the minimum and/or maximum values in an array. To help find these values, VTK-m provides the vtkm::cont::ArrayRangeCompute convenience function defined in vtkm/cont/ArrayRangeCompute.h. ArrayRangeCompute simply takes an ArrayHandle on which to find the range of values.

If given an array with vtkm::Vec values, ArrayRangeCompute computes the range separately for each component of the Vec. The return value for ArrayRangeCompute is vtkm::cont::ArrayHandle <vtkm::Range >. This returned array will have one value for each component of the input array's type. So for example if you call ArrayRangeCompute on a vtkm::cont::ArrayHandle <vtkm::Id3 >, the returned array of Ranges will have 3 values in it. Of course, when ArrayRangeCompute is run on an array of scalar types, you get an array with a single value in it.

Each value of vtkm::Range holds the minimum and maximum value for that component. The Range object is documented in Section 19.4.

Example 27.7: Using ArrayRangeCompute.

```
vtkm::cont::ArrayHandle<vtkm::Range> rangeArray =
  vtkm::cont::ArrayRangeCompute(arrayHandle);
auto rangePortal = rangeArray.GetPortalConstControl();
for (vtkm::Id index = 0; index < rangePortal.GetNumberOfValues(); ++index)
{
  vtkm::Range componentRange = rangePortal.Get(index);
  std::cout << "Values for component " << index << " go from "
            << componentRange.Min << " to " << componentRange.Max << std::endl;
}
```

Did you know?

ArrayRangeCompute will compute the minimum and maximum values in parallel. If desired, you can specify the parallel hardware device used for the computation as an optional second argument to ArrayRangeCompute. You can specify the device using a runtime device tracker, which is documented in Section 12.3.

27.4 Interface to Execution Environment

One of the main functions of the array handle is to allow an array to be defined in the control environment and then be used in the execution environment. When using an ArrayHandle with filters or worklets, this transition is handled automatically. However, it is also possible to invoke the transfer for use in your own scheduled algorithms.

The ArrayHandle class manages the transition from control to execution with a set of three methods that allocate, transfer, and ready the data in one operation. These methods all start with the prefix Prepare and are meant to be called before some operation happens in the execution environment. The methods are as follows.

ArrayHandle::PrepareForInput Copies data from the control to the execution environment, if necessary, and readies the data for read-only access.

ArrayHandle::PrepareForInPlace Copies the data from the control to the execution environment, if necessary, and readies the data for both reading and writing.

ArrayHandle::PrepareForOutput Allocates space (the size of which is given as a parameter) in the execution environment, if necessary, and readies the space for writing.

The PrepareForInput and PrepareForInPlace methods each take a single argument that is the device adapter tag where execution will take place (see Section 12.1 for more information on device adapter tags). PrepareForOutput takes two arguments: the size of the space to allocate and the device adapter tag. Each of these methods returns an array portal that can be used in the execution environment. PrepareForInput returns an object of type ArrayHandle::ExecutionTypes<*DeviceAdapterTag*>::PortalConst whereas PrepareForInPlace and PrepareForOutput each return an object of type ArrayHandle::ExecutionTypes<*DeviceAdapterTag*>::Portal.

Although these Prepare methods are called in the control environment, the returned array portal can only be used in the execution environment. Thus, the portal must be passed to an invocation of the execution environment.

Most of the time, the passing of ArrayHandle data to the execution environment is handled automatically by VTK-m. The most common need to call one of these Prepare methods is to build execution objects (Chapter 29) or to construct derived array types (Section 36.2.3). Examples of using these Prepare methods are in those respective sections.

Common Errors

There are many operations on ArrayHandle *that can invalidate the array portals, so do not keep them around indefinitely. It is generally better to keep a reference to the* ArrayHandle *and use one of the* Prepare *methods each time the data are accessed in the execution environment.*

CHAPTER

TWENTYEIGHT

GLOBAL ARRAYS AND TOPOLOGY

When writing an algorithm in VTK-m by creating a worklet, the data each instance of the worklet has access to is intentionally limited. This allows VTK-m to provide safety from race conditions and other parallel programming difficulties. However, there are times when the complexity of an algorithm requires all threads to have shared global access to a global structure. This chapter describes worklet tags that can be used to pass data globally to all instances of a worklet.

28.1 Whole Arrays

A *whole array* argument to a worklet allows you to pass in an ArrayHandle. All instances of the worklet will have access to all the data in the ArrayHandle.

> **Common Errors**
>
> *The VTK-m worklet invoking mechanism performs many safety checks to prevent race conditions across concurrently running worklets. Using a whole array within a worklet circumvents this guarantee of safety, so be careful when using whole arrays, especially when writing to whole arrays.*

A whole array is declared by adding a WholeArrayIn, a WholeArrayInOut, or a WholeArrayOut to the ControlSignature of a worklet. The corresponding argument to the Invoker should be an ArrayHandle. The ArrayHandle must already be allocated in all cases, including when using WholeArrayOut. When the data are passed to the operator of the worklet, it is passed as an array portal object. (Array portals are discussed in Section 27.1.) This means that the worklet can access any entry in the array with Get and/or Set methods.

We have already seen a demonstration of using a whole array in Example 21.2 to perform a simple array copy. Here we will construct a more thorough example of building functionality that requires random array access.

Let's say we want to measure the quality of triangles in a mesh. A common method for doing this is using the equation

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where a is the area of the triangle and h_1, h_2, and h_3 are the lengths of the sides. We can easily compute this in a cell to point map, but what if we want to speed up the computations by reducing precision? After all, we probably only care if the triangle is good, reasonable, or bad. So instead, let's build a lookup table and then retrieve the triangle quality from that lookup table based on its sides.

The following example demonstrates creating such a table lookup in an array and using a worklet argument tagged with WholeArrayIn to make it accessible.

Example 28.1: Using WholeArrayIn to access a lookup table in a worklet.

```
namespace detail
{

static const vtkm::Id TRIANGLE_QUALITY_TABLE_DIMENSION = 8;
static const vtkm::Id TRIANGLE_QUALITY_TABLE_SIZE =
  TRIANGLE_QUALITY_TABLE_DIMENSION * TRIANGLE_QUALITY_TABLE_DIMENSION;

VTKM_CONT
vtkm::cont::ArrayHandle<vtkm::Float32> GetTriangleQualityTable()
{
  // Use these precomputed values for the array. A real application would
  // probably use a larger array, but we are keeping it small for demonstration
  // purposes.
  static vtkm::Float32 triangleQualityBuffer[TRIANGLE_QUALITY_TABLE_SIZE] = {
    0, 0,        0,        0,        0,        0,        0,        0,
    0, 0,        0,        0,        0,        0,        0,        0.24431f,
    0, 0,        0,        0,        0,        0,        0.43298f, 0.47059f,
    0, 0,        0,        0,        0,        0.54217f, 0.65923f, 0.66408f,
    0, 0,        0,        0,        0.57972f, 0.75425f, 0.82154f, 0.81536f,
    0, 0,        0,        0.54217f, 0.75425f, 0.87460f, 0.92567f, 0.92071f,
    0, 0,        0.43298f, 0.65923f, 0.82154f, 0.92567f, 0.97664f, 0.98100f,
    0, 0.24431f, 0.47059f, 0.66408f, 0.81536f, 0.92071f, 0.98100f, 1
  };

  return vtkm::cont::make_ArrayHandle(triangleQualityBuffer,
                                      TRIANGLE_QUALITY_TABLE_SIZE);
}

template<typename T>
VTKM_EXEC_CONT vtkm::Vec<T, 3> TriangleEdgeLengths(const vtkm::Vec<T, 3>& point1,
                                                   const vtkm::Vec<T, 3>& point2,
                                                   const vtkm::Vec<T, 3>& point3)
{
  return vtkm::make_Vec(vtkm::Magnitude(point1 - point2),
                        vtkm::Magnitude(point2 - point3),
                        vtkm::Magnitude(point3 - point1));
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PortalType, typename T>
VTKM_EXEC_CONT static vtkm::Float32 LookupTriangleQuality(
  const PortalType& triangleQualityPortal,
  const vtkm::Vec<T, 3>& point1,
  const vtkm::Vec<T, 3>& point2,
  const vtkm::Vec<T, 3>& point3)
{
  vtkm::Vec<T, 3> edgeLengths = TriangleEdgeLengths(point1, point2, point3);

  // To reduce the size of the table, we just store the quality of triangles
  // with the longest edge of size 1. The table is 2D indexed by the length
  // of the other two edges. Thus, to use the table we have to identify the
  // longest edge and scale appropriately.
  T smallEdge1 = vtkm::Min(edgeLengths[0], edgeLengths[1]);
  T tmpEdge = vtkm::Max(edgeLengths[0], edgeLengths[1]);
  T smallEdge2 = vtkm::Min(edgeLengths[2], tmpEdge);
  T largeEdge = vtkm::Max(edgeLengths[2], tmpEdge);

  smallEdge1 /= largeEdge;
  smallEdge2 /= largeEdge;

  // Find index into array.
  vtkm::Id index1 = static_cast<vtkm::Id>(
    vtkm::Floor(smallEdge1 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
  vtkm::Id index2 = static_cast<vtkm::Id>(
    vtkm::Floor(smallEdge2 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
  vtkm::Id totalIndex = index1 + index2 * TRIANGLE_QUALITY_TABLE_DIMENSION;

  return triangleQualityPortal.Get(totalIndex);
}

} // namespace detail

struct TriangleQualityWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn cells,
                                FieldInPoint pointCoordinates,
                                WholeArrayIn triangleQualityTable,
                                FieldOutCell triangleQuality);
  using ExecutionSignature = _4(CellShape, _2, _3);
  using InputDomain = _1;

  template<typename CellShape,
           typename PointCoordinatesType,
           typename TriangleQualityTablePortalType>
  VTKM_EXEC vtkm::Float32 operator()(
    CellShape shape,
    const PointCoordinatesType& pointCoordinates,
    const TriangleQualityTablePortalType& triangleQualityTable) const
  {
    if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
    {
      this->RaiseError("Only triangles are supported for triangle quality.");
      return vtkm::Nan32();
    }

    return detail::LookupTriangleQuality(triangleQualityTable,
                                         pointCoordinates[0],
                                         pointCoordinates[1],
                                         pointCoordinates[2]);
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualityTable =
      detail::GetTriangleQualityTable();

    vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;

    this->Invoke(TriangleQualityWorklet{},
                 inputDataSet.GetCellSet(),
                 inputPointCoordinatesField,
                 triangleQualityTable,
                 triangleQualities);
```

28.2 Atomic Arrays

One of the problems with writing to whole arrays is that it is difficult to coordinate the access to an array from multiple threads. If multiple threads are going to write to a common index of an array, then you will probably need to use an *atomic array*.

An atomic array allows random access into an array of data, similar to a whole array. However, the operations on the values in the atomic array allow you to perform an operation that modifies its value that is guaranteed complete without being interrupted and potentially corrupted.

> Common Errors
>
> *Due to limitations in available atomic operations, atomic arrays can currently only contain* `vtkm::Int32` *or* `vtkm::Int64` *values.*

To use an array as an atomic array, first add the `AtomicArrayInOut` tag to the worklet's `ControlSignature`. The corresponding argument to the `Invoker` should be an `ArrayHandle`, which must already be allocated and initialized with values.

When the data are passed to the operator of the worklet, it is passed in a `vtkm::exec::AtomicArrayExecutionObject` structure. `AtomicArrayExecutionObject` has two important methods:

`Add` Takes as arguments an index and a value. The entry in the array corresponding to the index will have the value added to it. If multiple threads attempt to add to the same index in the array, the requests will be serialized so that the final result is the sum of all the additions. `AtomicArrayExecutionObject::Add` returns the value that was replaced. That is, it returns the value right *before* the addition.

`CompareAndSwap` Takes as arguments an index, a new value, and an old value. If the entry in the array corresponding to the index has the same value as the "old value," then it is changed to the "new value" and the original value is return from the method. If the entry in the array is not the same as the "old value," then nothing happens to the array and the value that is actually stored in the array is returned. If multiple threads attempt to compare and swap to the same index in the array, the requests are serialized.

> Common Errors
>
> *Atomic arrays help resolve hazards in parallel algorithms, but they come at a cost. Atomic operations are more costly than non-thread-safe ones, and they can slow a parallel program immensely if used incorrectly.*

The following example uses an atomic array to count the bins in a histogram. It does this by making the array of histogram bins an atomic array and then using an atomic add. Note that this is not the fastest way to create a histogram. We gave an implementation in Section 21.4 that is generally faster (unless your histogram happens to be very sparse). VTK-m also comes with a histogram worklet that uses a similar approach.

Example 28.2: Using `AtomicArrayInOut` to count histogram bins in a worklet.

```
struct CountBins : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn data, AtomicArrayInOut histogramBins);
  using ExecutionSignature = void(_1, _2);
  using InputDomain = _1;

  vtkm::Range HistogramRange;
  vtkm::Id NumberOfBins;

  VTKM_CONT
  CountBins(const vtkm::Range& histogramRange, vtkm::Id& numBins)
    : HistogramRange(histogramRange)
    , NumberOfBins(numBins)
  {
  }

  template<typename T, typename AtomicArrayType>
  VTKM_EXEC void operator()(T value, const AtomicArrayType& histogramBins) const
  {
    vtkm::Float64 interp =
      (value - this->HistogramRange.Min) / this->HistogramRange.Length();
    vtkm::Id bin = static_cast<vtkm::Id>(interp * this->NumberOfBins);
    if (bin < 0)
    {
      bin = 0;
    }
    if (bin >= this->NumberOfBins)
    {
      bin = this->NumberOfBins - 1;
    }

    histogramBins.Add(bin, 1);
  }
};
```

28.3 Whole Cell Sets

Section 21.2 describes how to make a topology map filter that performs an operation on cell sets. The worklet has access to a single cell element (such as point or cell) and its immediate connections. But there are cases when you need more general queries on a topology. For example, you might need more detailed information than the topology map gives or you might need to trace connections from one cell to the next. To do this VTK-m allows you to provide a *whole cell set* argument to a worklet that provides random access to the entire topology.

A whole cell set is declared by adding a WholeCellSetIn to the worklet's ControlSignature. The corresponding argument to the Invoker should be a CellSet subclass or a DynamicCellSet (both of which are described in Section 7.2).

The WholeCellSetIn is templated and takes two arguments: the "visit" topology type and the "incident" topology type, respectively. These template arguments must be one of the topology element tags, but for convenience you can use Point and Cell in lieu of vtkm::TopologyElementTagPoint and vtkm::TopologyElementTagCell, respectively. The "visit" and "incident" topology types define which topological elements can be queried (visited) and which incident elements are returned. The semantics of the "visit" and "incident" topology is the same as that for the general topology maps described in Section 21.2.3. You can look up an element of the "visit" topology by index and then get all of the "incident" elements from it.

For example, a WholeCellSetIn<Cell, Point> allows you to find all the points that are incident on each cell (as well as querying the cell shape). Likewise, a WholeCellSetIn<Point, Cell> allows you to find all the cells that are incident on each point. The default parameters of WholeCellSetIn are visiting cells with incident points. That is, WholeCellSetIn<> is equivalent to WholeCellSetIn<Cell, Point>.

When the cell set is passed to the operator of the worklet, it is passed in a special connectivity object. The actual object type depends on the cell set, but vtkm::exec::CellSetStructured and are two common examples vtkm::exec::CellSetExplicit. All these connectivity objects share a common interface. First, they all declare the following public types.

CellShapeTag The tag for the cell shapes of the cell set. (Cell shape tags are described in Section 25.1.) If the connectivity potentially contains more than one type of cell shape, then this type will be vtkm::CellShapeTagGeneric.

IndicesType A Vec-like type that stores all the incident indices.

Second they all provide the following methods.

GetNumberOfElements Get the number of "to" topology elements in the cell set. All the other methods require an element index, and this represents the range of valid indices. The return type is vtkm::Id.

GetCellShape Takes an index for an element and returns a CellShapeTag object of the corresponding cell shape. If the "to" topology elements are not strictly cell, then a reasonably close shape is returned. For example, if the "to" topology elements are points, then the shape is returned as a vertex.

GetNumberOfIndices Takes an index for an element and returns the number of incident "from" elements are connected to it. The returned type is vtkm::IdComponent.

GetIndices Takes an index for an element and returns a Vec-like object of type IndicesType containing the indices of all incident "from" elements. The size of the Vec-like object is the same as that returned from GetNumberOfIndicices.

VTK-m comes with several functions to work with the shape and index information returned from these connectivity objects. Most of these methods are documented in Chapter 25.

Let us use the whole cell set feature to help us determine the "flatness" of a polygonal mesh. We will do this by summing up all the angles incident on each on each point. That is, for each point, we will find each incident polygon, then find the part of that polygon using the given point, then computing the angle at that point, and then summing for all such angles. So, for example, in the mesh fragment shown in Figure 28.1 one of the angles attached to the middle point is labeled θ_j.

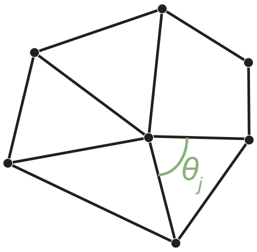


Figure 28.1: The angles incident around a point in a mesh.

We want a worklet to compute $\sum_j \theta$ for all such attached angles. This measure is related (but not the same as) the curvature of the surface. A flat surface will have a sum of 2π. Convex and concave surfaces have a value less than 2π, and saddle surfaces have a value greater than 2π.

To do this, we create a visit points with cells worklet (Section 21.2.2) that visits every point and gives the index of every incident cell. The worklet then uses a whole cell set to inspect each incident cell to measure the attached angle and sum them together.

Example 28.3: Using WholeCellSetIn to sum the angles around each point.

```
struct SumOfAngles : vtkm::worklet::WorkletVisitPointsWithCells
{
  using ControlSignature = void(CellSetIn inputCells,
                                WholeCellSetIn<>, // Same as inputCells
                                WholeArrayIn pointCoords,
                                FieldOutPoint angleSum);
  using ExecutionSignature = void(CellIndices incidentCells,
                                  InputIndex pointIndex,
                                  _2 cellSet,
                                  _3 pointCoordsPortal,
                                  _4 outSum);
  using InputDomain = _1;

  template<typename IncidentCellVecType,
           typename CellSetType,
           typename PointCoordsPortalType,
           typename SumType>
  VTKM_EXEC void operator()(const IncidentCellVecType& incidentCells,
                            vtkm::Id pointIndex,
                            const CellSetType& cellSet,
                            const PointCoordsPortalType& pointCoordsPortal,
                            SumType& outSum) const
  {
    using CoordType = typename PointCoordsPortalType::ValueType;

    CoordType thisPoint = pointCoordsPortal.Get(pointIndex);

    outSum = 0;
    for (vtkm::IdComponent incidentCellIndex = 0;
         incidentCellIndex < incidentCells.GetNumberOfComponents();
         ++incidentCellIndex)
    {
      // Get information about incident cell.
      vtkm::Id cellIndex = incidentCells[incidentCellIndex];
      typename CellSetType::CellShapeTag cellShape = cellSet.GetCellShape(cellIndex);
      typename CellSetType::IndicesType cellConnections =
        cellSet.GetIndices(cellIndex);
      vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
      vtkm::IdComponent numEdges =
        vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);

      // Iterate over all edges and find the first one with pointIndex.
      // Use that to find the first vector.
      vtkm::IdComponent edgeIndex = -1;
      CoordType vec1;
      while (true)
      {
        ++edgeIndex;
        if (edgeIndex >= numEdges)
        {
          this->RaiseError("Bad cell. Could not find two incident edges.");
          return;
        }
        auto edge =
          vtkm::make_Vec(vtkm::exec::CellEdgeLocalIndex(
                           numPointsInCell, 0, edgeIndex, cellShape, *this),
                         vtkm::exec::CellEdgeLocalIndex(
                           numPointsInCell, 1, edgeIndex, cellShape, *this));
        if (cellConnections[edge[0]] == pointIndex)
        {
          vec1 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
          break;
        }
        else if (cellConnections[edge[1]] == pointIndex)
        {
          vec1 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
          break;
        }
        else
        {
          // Continue to next iteration of loop.
        }
      }

      // Continue iteration over remaining edges and find the second one with
      // pointIndex. Use that to find the second vector.
      CoordType vec2;
      while (true)
      {
        ++edgeIndex;
        if (edgeIndex >= numEdges)
        {
          this->RaiseError("Bad cell. Could not find two incident edges.");
          return;
        }
        auto edge =
          vtkm::make_Vec(vtkm::exec::CellEdgeLocalIndex(
                           numPointsInCell, 0, edgeIndex, cellShape, *this),
                         vtkm::exec::CellEdgeLocalIndex(
                           numPointsInCell, 1, edgeIndex, cellShape, *this));
        if (cellConnections[edge[0]] == pointIndex)
        {
          vec2 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
          break;
        }
        else if (cellConnections[edge[1]] == pointIndex)
        {
          vec2 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
          break;
        }
        else
        {
          // Continue to next iteration of loop.
        }
      }

      // The dot product of two unit vectors is equal to the cosine of the
      // angle between them.
      vtkm::Normalize(vec1);
      vtkm::Normalize(vec2);
      SumType cosine = static_cast<SumType>(vtkm::Dot(vec1, vec2));

      outSum += vtkm::ACos(cosine);
    }
  }
};
```

CHAPTER

TWENTYNINE

EXECUTION OBJECTS

Although passing whole arrays and cell sets into a worklet is a convenient way to provide data to a worklet that is not divided by the input or output domain, they are sometimes not the best structures to represent data. Thus, all worklets support a another type of argument called an *execution object*, or exec object for short, that provides a user-defined object directly to each invocation of the worklet. This is defined by an `ExecObject` tag in the `ControlSignature`.

The execution object must be a subclass of `vtkm::cont::ExecutionObjectBase`. Also, it must implement a `PrepareForExecution` method declared with `VTKM_CONT`, templated on device adapter tag, and passed as an argument. The `PrepareForExecution` function creates an execution object that can be passed from the control environment to the execution environment and be usable in the execution environment, and any method of the produced object used within the worklet must be declared with `VTKM_EXEC` or `VTKM_EXEC_CONT`.

An execution object can refer to an array, but the array reference must be through an array portal for the execution environment. This can be retrieved from the `ArrayHandle::PrepareForInput` method as described in Section 27.4. Other VTK-m data objects, such as the subclasses of `vtkm::cont::CellSet`, have similar methods.

Returning to the example we have in Section 28.1, we are computing triangle quality quickly by looking up the value in a table. In Example 28.1 (page 226) the table is passed directly to the worklet as a whole array. However, there is some additional code involved to get the appropriate index into the table for a given triangle. Let us say that we want to have the ability to compute triangle quality in many different worklets. Rather than pass in a raw array, it would be better to encapsulate the functionality in an object.

We can do that by creating an execution object with a `PrepareForExecution` that creates an object that has the table stored inside and methods to compute the triangle quality. The following example uses the table built in Example 28.1 to create such an object.

Example 29.1: Using `ExecObject` to access a lookup table in a worklet.

```
template<typename DeviceAdapterTag>
class TriangleQualityTableExecutionObject
{
public:
  VTKM_CONT
  TriangleQualityTableExecutionObject()
  {
    this->TablePortal =
      detail::GetTriangleQualityTable().PrepareForInput(DeviceAdapterTag());
  }

  template<typename T>
  VTKM_EXEC vtkm::Float32 GetQuality(const vtkm::Vec<T, 3>& point1,
                                     const vtkm::Vec<T, 3>& point2,
                                     const vtkm::Vec<T, 3>& point3) const
  {
    return detail::LookupTriangleQuality(this->TablePortal, point1, point2, point3);
  }

private:
  using TableArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
  using TableArrayPortalType =
    typename TableArrayType::ExecutionTypes<DeviceAdapterTag>::PortalConst;
  TableArrayPortalType TablePortal;
};

class TriangleQualityTable : public vtkm::cont::ExecutionObjectBase
{
public:
  template<typename Device>
  VTKM_CONT TriangleQualityTableExecutionObject<Device> PrepareForExecution(
    Device) const
  {
    return TriangleQualityTableExecutionObject<Device>();
  }
};

struct TriangleQualityWorklet2 : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn cells,
                                FieldInPoint pointCoordinates,
                                ExecObject triangleQualityTable,
                                FieldOutCell triangleQuality);
  using ExecutionSignature = _4(CellShape, _2, _3);
  using InputDomain = _1;

  template<typename CellShape,
           typename PointCoordinatesType,
           typename TriangleQualityTableType>
  VTKM_EXEC vtkm::Float32 operator()(
    CellShape shape,
    const PointCoordinatesType& pointCoordinates,
    const TriangleQualityTableType& triangleQualityTable) const
  {
    if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
    {
      this->RaiseError("Only triangles are supported for triangle quality.");
      return vtkm::Nan32();
    }

    return triangleQualityTable.GetQuality(
      pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
  }
};

//
// Later in the associated Filter class...
//

    TriangleQualityTable triangleQualityTable;

    vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;

    this->Invoke(TriangleQualityWorklet2{},
                 inputDataSet.GetCellSet(),
                 inputPointCoordinatesField,
                 triangleQualityTable,
                 triangleQualities);
```

CHAPTER

THIRTY

LOCATORS

Locators are a special type of structure that allows you to take a point coordinate in space and then find a topological element that contains or is near that coordinate. VTK-m comes with multiple types of locators, which are categorized by the type of topological element that they find. For example, a *cell locator* takes a coordinate in world space and finds the cell in a `vtkm::cont::DataSet` that contains that cell. Likewise, a *point locator* takes a coordinate in world space and finds a point from a `vtkm::cont::CoordinateSystem` nearby.

Different locators differ in their interface slightly, but they all follow the same basic operation. First, they are constructed and provided with one or more elements of a `vtkm::cont::DataSet`. Then they are built with a call to an `Update` method. The locator can then be passed to a worklet as an `ExecObject`, which will cause the worklet to get a special execution version of the locator that can do the queries.

Did you know?

Other visualization libraries, like VTK-m's big sister toolkit VTK, provide similar locator structures that allow iterative building by adding one element at a time. VTK-m explicitly disallows this use case. Although iteratively adding elements to a locator is undoubtedly useful, such an operation will inevitably bottleneck a highly threaded algorithm in critical sections. This makes iterative additions to locators too costly to support in VTK-m.

30.1 Cell Locators

Cell Locators in VTK-m provide a means of building spatial search structures that can later be used to find a cell containing a certain point. This could be useful in scenarios where the application demands the cell to which a point belongs to to achieve a certain functionality. For example, while tracing a particle's path through a vector field, after every step we lookup which cell the particle has entered to interpolate the velocity at the new location to take the next step.

Using cell locators is a two step process. The first step is to build the search structure. This is done by instantiating one of the subclasses of `vtkm::cont::CellLocator`, providing a cell set and coordinate system (usually from a `vtkm::cont::DataSet`), and then updating the structure. Once the cell locator is built, it can be used in the execution environment within a filter or worklet.

30.1.1 Building a Cell Locator

All Cell Locators in VTK-m inherit from vtkm::cont::CellLocator, which provides the basic interface for the required features of cell locators. This generic interface provides methods to set the cell set (with SetCellSet and GetCellSet) and to set the coordinate system (with SetCoordinates and GetCoordinates). Once the cell set and coordinates are provided, you may call Update to construct the search structures. Although Update is called from the control environment, the search structure will be built on parallel devices.

Example 30.1: Constructing a CellLocator.

```
    vtkm::cont::CellLocatorGeneral cellLocator;
    cellLocator.SetCellSet(inDataSet.GetCellSet());
    cellLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    cellLocator.Update();
```

VTK-m currently exposes the implementations of the following Cell Locators.

vtkm::cont::CellLocatorGeneral This locator will automatically select another locator to use as its implementation. CellLocatorGeneral allows you to automatically select cell locators optimized for certain cell structures without knowing the cell set type. You can change how CellLocatorGeneral selects a CellLocator by providing a function to the SetConfigurator method. If no configurator is set, then a default one is used.

vtkm::cont::CellLocatorUniformGrid This locator is optimized for structured data that has uniform axis-aligned spacing. For this cell locator to work, it has to be given a cell set of type CellSetStructured and a coordinate system using an ArrayHandleUniformPointCoordinates for its data.

vtkm::cont::CellLocatorRectilinearGrid This locator is optimized for structured data that has nonuniform axis-aligned spacing. For this cell locator to work, it has to be given a cell set of type CellSetStructured and a coordinate system using an ArrayHandleCartesianProduct for its data.

vtkm::cont::CellLocatorUniformBins This locator builds a 2-level hierarchy of uniform bins. The first level is a coarse partitioning of the space. Each bin in the first level has a second grid who's size depends on the number of cells in the first level. The density (number of cells expected in each bin) for each level can be set with SetDensityL1 and SetDensityL2. Their default values are 32 and 2, respectively.

vtkm::cont::CellLocatorBoundingIntervalHierarchy This locator is based on the bounding interval hierarchy spatial search structure. CellLocatorBoundingIntervalHierarchy takes two parameters: the number of splitting planes used to split the cells uniformly along an axis at each level and the maximum leaf size, which determines if a node needs to be split further. These parameters can set through the SetNumberOfPlanes and SetMaxLeafSize methods.

30.1.2 Using Cell Locators in a Worklet

The vtkm::cont::CellLocator interface implements vtkm::cont::ExecutionObjectBase. This means that any CellLocator can be used in worklets as an ExecObject argument (as defined in the ControlSignature). See Chapter 29 for information on ExecObject arguments to worklets.

When a vtkm::cont::CellLocator class is passed as an ExecObject argument to a worklet Invoke, the worklet receives a pointer to a vtkm::exec::CellLocator object. vtkm::exec::CellLocator provides a FindCell method that identifies a containing cell given a point location in space.

Common Errors

Note that vtkm::cont::CellLocator *and* vtkm::exec::CellLocator *are different objects with different interfaces despite the similar names.*

The CellLocator::FindCell method takes 4 arguments. The first argument is an input query point. The second argument is used to return the id of the cell containing this point (or -1 if the point is not found in any cell). The third argument is used to return the parametric coordinates for the point within the cell (assuming it is found in any cell). The fourth argument is a reference to the calling worklet (used for error reporting purposes). The following example defines a simple worklet to get the value of a point field interpolated to a group of query point coordinates provided.

Example 30.2: Using a CellLocator in a worklet.

```
struct QueryCellsWorklet : public vtkm::worklet::WorkletMapField
{
  using ControlSignature =
    void(FieldIn, ExecObject, WholeCellSetIn<Cell, Point>, WholeArrayIn, FieldOut);
  using ExecutionSignature = void(_1, _2, _3, _4, _5);

  template<typename Point,
           typename CellLocatorExecObject,
           typename CellSet,
           typename FieldPortal,
           typename OutType>
  VTKM_EXEC void operator()(const Point& point,
                            const CellLocatorExecObject& cellLocator,
                            const CellSet& cellSet,
                            const FieldPortal& field,
                            OutType& out) const
  {
    // Use the cell locator to find the cell containing the point and the parametric
    // coordinates within that cell.
    vtkm::Id cellId;
    vtkm::Vec3f parametric;
    cellLocator->FindCell(point, cellId, parametric, *this);

    // Use this information to interpolate the point field to the given location.
    if (cellId >= 0)
    {
      // Get shape information about the cell containing the point coordinate
      auto cellShape = cellSet.GetCellShape(cellId);
      auto indices = cellSet.GetIndices(cellId);

      // Make a Vec-like containing the field data at the cell's points
      auto fieldValues = vtkm::make_VecFromPortalPermute(&indices, &field);

      // Do the interpolation
      out = vtkm::exec::CellInterpolate(fieldValues, parametric, cellShape, *this);
    }
    else
    {
      this->RaiseError("Given point outside of the cell set.");
    }
  }
};

//
// Later in the associated Filter class...
//
```

```

    vtkm::cont::CellLocatorGeneral cellLocator;
    cellLocator.SetCellSet(inDataSet.GetCellSet());
    cellLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    cellLocator.Update();

    vtkm::cont::ArrayHandle<FieldType> interpolatedField;

    this->Invoke(QueryCellsWorklet{},
                 this->QueryPoints,
                 &cellLocator,
                 inDataSet.GetCellSet(),
                 inputField,
                 interpolatedField);
```

30.2 Point Locators

Point Locators in VTK-m provide a means of building spatial search structures that can later be used to find the nearest neighbor a certain point. This could be useful in scenarios where the closest pairs of points are needed. For example, during halo finding of particles in cosmology simulations, pairs of nearest neighbors within certain linking length are used to form clusters of particles.

Using cell locators is a two step process. The first step is to build the search structure. This is done by instantiating one of the subclasses of `vtkm::cont::PointLocator`, providing a coordinate system (usually from a `vtkm::cont::DataSet`) representing the location of points that can later be found through queries, and then updating the structure. Once the cell locator is built, it can be used in the execution environment within a filter or worklet.

30.2.1 Building Point Locators

All point Locators in VTK-m inherit from `vtkm::cont::PointLocator`, which provides the basic interface for the required features of point locators. This generic interface provides methods to set the coordinate system (with `SetCoordinates` and `GetCoordinates`) of training points. Once the coordinates are provided, you may call `Update` to construct the search structures. Although `Update` is called from the control environment, the search structure will be built on parallel devices

Example 30.3: Constructing a `PointLocator`.

```
    vtkm::cont::PointLocatorUniformGrid pointLocator;
    pointLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    pointLocator.Update();
```

VTK-m currently exposes the implementations of the following Point Locators.

`vtkm::cont::PointLocatorUniformGrid` This point locator is based on the uniform grid search structure. It divides the search space into a uniform grid of bins. A search for a point near a given coordinate starts in the bin containing the search coordinates. If a candidate point is not found in that bin, points are searched in an expanding neighborhood of grid bins. The size of the grid used by the locator to partition the space can be set with `SetNumberOfBins`. By default, `PointLocatorUniformGrid` uses a 32^3 grid. It is also possible to set the physical space over which the search space is constructed with the `SetRange` method. If the range is not set, it will automatically be set to the space of the coordinates.

30.2.2 Using Point Locators in a Worklet

The vtkm::cont::PointLocator interface implements vtkm::cont::ExecutionObjectBase. This means that any PointLocator can be used in worklets as an ExecObject argument (as defined in the ControlSignature). See Chapter 29 for information on ExecObject arguments to worklets.

When a vtkm::cont::PointLocator class is passed as an ExecObject argument to a worklet Invoke, the worklet receives a pointer to a vtkm::exec::PointLocator object. vtkm::exec::PointLocator provides a FindNearestNeighbor method that identifies the nearest neighbor point given a coordinate in space.

Common Errors

Note that vtkm::cont::PointLocator *and* vtkm::exec::PointLocator *are different objects with different interfaces despite the similar names.*

The FindNearestNeighbor method takes 3 arguments. The first argument is an input query point. The second argument is used to return the id of the nearest neighbor point (or -1 if no nearby point is found, for example, in the case of an empty set of data set points). The third argument is used to return the squared distance for the query point to its nearest neighbor.

Example 30.4: Using a PointLocator in a worklet.

```
/// Worklet that generates for each input coordinate a unit vector that points
/// to the closest point in a locator.
struct PointToClosestWorklet : public vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn, ExecObject, WholeArrayIn, FieldOut);
  using ExecutionSignature = void(_1, _2, _3, _4);

  template<typename Point,
           typename PointLocatorExecObject,
           typename CoordinateSystemPortal,
           typename OutType>
  VTKM_EXEC void operator()(const Point& queryPoint,
                            const PointLocatorExecObject& pointLocator,
                            const CoordinateSystemPortal& coordinateSystem,
                            OutType& out) const
  {
    // Use the point locator to find the point in the locator closest to the point
    // given.
    vtkm::Id pointId;
    vtkm::FloatDefault distanceSquared;
    pointLocator->FindNearestNeighbor(queryPoint, pointId, distanceSquared);

    // Use this information to find the nearest point and create a unit vector
    // pointing to it.
    if (pointId >= 0)
    {
      // Get nearest point coordinate.
      auto point = coordinateSystem.Get(pointId);

      // Get the vector pointing to this point
      out = point - queryPoint;

      // Convert to unit vector (if possible)
      if (distanceSquared > vtkm::Epsilon<vtkm::FloatDefault>())
      {
        out = vtkm::RSqrt(distanceSquared) * out;
      }
    }
    else
    {
      this->RaiseError("Locator could not find closest point.");
    }
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::PointLocatorUniformGrid pointLocator;
    pointLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    pointLocator.Update();

    vtkm::cont::ArrayHandle<vtkm::Vec3f> pointDirections;

    this->Invoke(PointToClosestWorklet{},
                 this->QueryPoints,
                 &pointLocator,
                 pointLocator.GetCoordinates(),
                 pointDirections);
```

CHAPTER

THIRTYONE

WORKLET INPUT OUTPUT SEMANTICS

The default scheduling of a worklet provides a 1 to 1 mapping from the input domain to the output domain. For example, a `vtkm::worklet::WorkletMapField` gets run once for every item of the input array and produces one item for the output array. Likewise, `vtkm::worklet::WorkletVisitCellsWithPoints` gets run once for every cell in the input topology and produces one associated item for the output field.

However, there are many operations that do not fall well into this 1 to 1 mapping procedure. The operation might need to pass over elements that produce no value or the operation might need to produce multiple values for a single input element. Such non 1 to 1 mappings can be achieved by defining a scatter or a mask (or both) on a worklet.

31.1 Scatter

A *scatter* allows you to specify for each input element how many output elements should be created. For example, a scatter allows you to create two output elements for every input element. A scatter could also allow you to drop every other input element from the output. The following types of scatter are provided by VTK-m.

`vtkm::worklet::ScatterIdentity` Provides a basic 1 to 1 mapping from input to output. This is the default scatter used if none is specified.

`vtkm::worklet::ScatterUniform` Provides a 1 to many mapping from input to output with the same number of outputs for each input. A template parameter provides the number of output values to produce per input.

`vtkm::worklet::ScatterCounting` Provides a 1 to any mapping from input to output with different numbers of outputs for each input. The constructor takes an `ArrayHandle` that is the same size as the input containing the count of output values to produce for each input. Values can be zero, in which case that input will be skipped.

`vtkm::worklet::ScatterPermutation` Reorders the indices. The constructor takes a permutation `ArrayHandle` that is sized to the number of output values and maps output indices to input indices. For example, if index i of the permutation array contains j, then the worklet invocation for output i will get the j^{th} input values. The reordering does not have to be 1 to 1. Any input not referenced by the permutation array will be dropped, and any input referenced by the permutation array multiple times will be duplicated. However, unlike `ScatterCounting` VisitIndex is always 0 even if an input value happens to be duplicated.

Did you know?

Scatters are often used to create multiple outputs for a single input, but they can also be used to remove inputs from the output. In particular, if you provide a count of 0 in a ScatterCounting *count array, no outputs will be created for the associated input. To simply mask out some elements from the input, provide* ScatterCounting *with a stencil array of 0's and 1's with a 0 for every element you want to remove and a 1 for every element you want to pass. You can also mix 0's with counts larger than 1 to drop some elements and add multiple results for other elements.* ScatterPermutation *can similarly be used to remove input values by leaving them out of the permutation.*

To define a scatter procedure, the worklet must provide a type definition named ScatterType. The ScatterType must be set to one of the aforementioned Scatter* classes. It is common, but optional, to also provide a static method named MakeScatter that generates an appropriate scatter object for the worklet if you cannot use the default constructor for the scatter. This static method can be used by users of the worklet to set up the scatter for the Invoker.

Example 31.1: Declaration of a scatter type in a worklet.

```
  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CountArrayType>
  VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray)
  {
    VTKM_IS_ARRAY_HANDLE(CountArrayType);
    return ScatterType(countArray);
  }
```

When using a scatter that produces multiple outputs for a single input, the worklet is invoked multiple times with the same input values. In such an event the worklet operator needs to distinguish these calls to produce the correct associated output. This is done by declaring one of the ExecutionSignature arguments as VisitIndex. This tag will pass a vtkm::IdComponent to the worklet that identifies which invocation is being called.

It is also the case that the when a scatter can produce multiple outputs for some input that the index of the input element is not the same as the WorkIndex. If the index to the input element is needed, you can use the InputIndex tag in the ExecutionSignature. It is also good practice to use the OutputIndex tag if the index to the output element is needed.

Most Scatter objects have a state, and this state must be passed to the vtkm::cont::Invoker when invoking the worklet. In this case, the Scatter object should be passed as the second object to the call to the Invoker (after the worklet object).

Example 31.2: Invoking with a custom scatter.

```
  vtkm::worklet::ScatterCounting generateScatter =
    ClipPoints::Generate::MakeScatter(countArray);
  this->Invoke(
    ClipPoints::Generate{}, generateScatter, inField, clippedPointsArray);
```

Did you know?

A scatter object does not have to be tied to a single worklet/invoker instance. In some cases it makes sense to use the same scatter object multiple times for worklets that have the same input to output mapping. Although this is not common, it can save time by reusing the set up computations of ScatterCounting*.*

To demonstrate using scatters with worklets, we provide some contrived but illustrative examples. The first example is a worklet that takes a pair of input arrays and interleaves them so that the first, third, fifth, and so on entries come from the first array and the second, fourth, sixth, and so on entries come from the second array. We achieve this by using a vtkm::worklet::ScatterUniform of size 2 and using the VisitIndex to determine from which array to pull a value.

Example 31.3: Using ScatterUniform.

```cpp
struct InterleaveArrays : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn, FieldIn, FieldOut);
  using ExecutionSignature = void(_1, _2, _3, VisitIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterUniform<2>;

  template<typename T>
  VTKM_EXEC void operator()(const T& input0,
                            const T& input1,
                            T& output,
                            vtkm::IdComponent visitIndex) const
  {
    if (visitIndex == 0)
    {
      output = input0;
    }
    else // visitIndex == 1
    {
      output = input1;
    }
  }
};
```

The second example takes a collection of point coordinates and clips them by an axis-aligned bounding box. It does this using a vtkm::worklet::ScatterCounting with an array containing 0 for all points outside the bounds and 1 for all points inside the bounds. As is typical with this type of operation, we use another worklet with a default identity scatter to build the count array.

Example 31.4: Using ScatterCounting.

```cpp
struct ClipPoints
{
  class Count : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn points, FieldOut count);
    using ExecutionSignature = _2(_1);
    using InputDomain = _1;

    VTKM_CONT Count(const vtkm::Bounds& bounds)
      : Bounds(bounds)
    {
    }

    template<typename T>
    VTKM_EXEC vtkm::IdComponent operator()(const vtkm::Vec<T, 3>& point) const
    {
      return (this->Bounds.Contains(point) ? 1 : 0);
    }

  private:
    vtkm::Bounds Bounds;
  };

  class Generate : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn inPoints, FieldOut outPoints);
    using ExecutionSignature = void(_1, _2);
    using InputDomain = _1;

    using ScatterType = vtkm::worklet::ScatterCounting;

    template<typename CountArrayType>
    VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray)
    {
      VTKM_IS_ARRAY_HANDLE(CountArrayType);
      return ScatterType(countArray);
    }

    template<typename InType, typename OutType>
    VTKM_EXEC void operator()(const vtkm::Vec<InType, 3>& inPoint,
                              vtkm::Vec<OutType, 3>& outPoint) const
    {
      // The scatter ensures that this method is only called for input points
      // that are passed to the output (where the count was 1). Thus, in this
      // case we know that we just need to copy the input to the output.
      outPoint = vtkm::Vec<OutType, 3>(inPoint[0], inPoint[1], inPoint[2]);
    }
  };
};

//
// Later in the associated Filter class...
//

    vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray;

    this->Invoke(ClipPoints::Count(this->Bounds), inField, countArray);

    vtkm::cont::ArrayHandle<T> clippedPointsArray;

    vtkm::worklet::ScatterCounting generateScatter =
      ClipPoints::Generate::MakeScatter(countArray);
    this->Invoke(
      ClipPoints::Generate{}, generateScatter, inField, clippedPointsArray);
```

The third example takes an input array and reverses the ordering. It does this using a vtkm::worklet::ScatterPermutation with a permutation array generated from a vtkm::cont::ArrayHandleCounting counting down from the input array size to 0.

Example 31.5: Using ScatterPermutation.

```
struct ReverseArrayWorklet : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn inputArray, FieldOut outputArray);
  using ExecutionSignature = void(_1, _2);
  using InputDomain = _1;

  using ArrayStorageTag =
    typename vtkm::cont::ArrayHandleCounting<vtkm::Id>::StorageTag;
  using ScatterType = vtkm::worklet::ScatterPermutation<ArrayStorageTag>;

  VTKM_CONT
  static ScatterType MakeScatter(vtkm::Id arraySize)
  {
    return ScatterType(
      vtkm::cont::ArrayHandleCounting<vtkm::Id>(arraySize - 1, -1, arraySize));
  }

  template<typename FieldType>
  VTKM_EXEC void operator()(FieldType inputArrayField,
                            FieldType& outputArrayField) const
  {
    outputArrayField = inputArrayField;
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::ArrayHandle<T> outputField;
    this->Invoke(ReverseArrayWorklet{},
                 ReverseArrayWorklet::MakeScatter(inputField.GetNumberOfValues()),
                 inputField,
                 outputField);
```

Did you know?

A vtkm::worklet::ScatterPermutation *can have less memory usage than a* vtkm::worklet::ScatterCounting *when zeroing indices. By default, a* vtkm::worklet::ScatterPermutation *will omit all fields that are not specified in the input permutation, whereas* vtkm::worklet::ScatterCounting *requires 0 values. If mapping an input to an output that omits fields, consider using a* vtkm::worklet::ScatterPermutation *to save memory.*

Common Errors

A permutation array provided to vtkm::worklet::ScatterPermutation *can be filled with arbitrary id values. If an input permutation id exceeds the bounds of an input provided to a* worklet*, an out of bounds error will occur in the worklet functor. To prevent this kind of error, you should ensure that ids in the* vtkm::worklet::ScatterPermutation *do not exceed the bounds of provided inputs.*

CHAPTER

THIRTYTWO

GENERATING CELL SETS

This chapter describes techniques for designing algorithms in VTK-m that generate cell sets to be inserted in a `vtkm::cont::DataSet`. Although Chapter 7 on data sets describes how to create a data set, including defining its set of cells, these are serial functions run in the control environment that are not designed for computing geometric structures. Rather, they are designed for specifying data sets built from existing data arrays, from inherently slow processes (such as file I/O), or for small test data. In this chapter we discuss how to write worklets that create new mesh topologies by writing data that can be incorporated into a `vtkm::cont::CellSet`.

This chapter is constructed as a set of patterns that are commonly employed to build cell sets. These techniques apply the worklet structures documented in Chapter 21. Although it is possible for these worklets to generate data of its own, the algorithms described here follow the more common use case of deriving one topology from another input data set. This chapter is not (and cannot be) completely comprehensive by covering every possible mechanism for building cell sets. Instead, we provide the basic and common patterns used in scientific visualization.

32.1 Single Cell Type

For our first example of algorithms that generate cell sets is one that creates a set of cells in which all the cells are of the same shape and have the same number of points. Our motivating example is an algorithm that will extract all the edges from a cell set. The resulting cell set will comprise a collection of line cells that represent the edges from the original cell set. Since all cell edges can be represented as lines with two endpoints, we know all the output cells will be of the same type. As we will see later in the example, we can use a `vtkm::cont::CellSetSingleType` to represent the data.

It is rare that an algorithm generating a cell set will generate exactly one output cell for each input cell. Thus, the first step in an algorithm generating a cell set is to count the number of cells each input item will create. In our motivating example, this is the the number of edges for each input cell.

Example 32.1: A simple worklet to count the number of edges on each cell.

```
struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
  using ExecutionSignature = _2(CellShape, PointCount);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC_CONT vtkm::IdComponent operator()(
    CellShapeTag cellShape,
    vtkm::IdComponent numPointsInCell) const
  {
    return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
  }
};
```

This count array generated in Example 32.1 can be used in a vtkm::worklet::ScatterCounting of a subsequent worklet that generates the output cells. (See Section 31.1 for information on using a scatter with a worklet.) We will see this momentarily.

Did you know?

If you happen to have an operation that you know will have the same count for every input cell, then you can skip the count step and use a vtkm::worklet::ScatterUniform *instead of* ScatterCount*. Doing so will simplify the code and skip some computation. We cannot use* ScatterUniform *in this example because different cell shapes have different numbers of edges and therefore different counts. However, if we were theoretically to make an optimization for 3D structured grids, we know that each cell is a hexahedron with 12 edges and could use a* ScatterUniform<12> *for that.*

The second and final worklet we need to generate our wireframe cells is one that outputs the indices of an edge. The worklet parenthesis' operator takes information about the input cell (shape and point indices) and an index of which edge to output. The aforementioned ScatterCounting provides a VisitIndex that signals which edge to output. The worklet parenthesis operator returns the two indices for the line in, naturally enough, a vtkm::Id2.

Example 32.2: A worklet to generate indices for line cells.

```
class EdgeIndicesWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut connectivityOut);
  using ExecutionSignature = void(CellShape, PointIndices, _2, VisitIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::Id2& connectivityOut,
                            vtkm::IdComponent edgeIndex) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();

    vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 0, edgeIndex, cellShape, *this);
    vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 1, edgeIndex, cellShape, *this);

    connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
    connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
  }
};
```

Our ultimate goal is to fill a vtkm::cont::CellSetSingleType object with the generated line cells. A CellSetSingleType requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively. The last item, the array of connection indices, is what we are creating with the worklet in Example 32.2.

However, there is a complication. The connectivity array for CellSetSingleType is expected to be a flat array of vtkm::Id indices, not an array of Vec objects. We could jump through some hoops adjusting the ScatterCounting to allow the worklet to output only one index of one cell rather than all indices of one cell. But that would be overly complicated and inefficient.

A simpler approach is to use the vtkm::cont::ArrayHandleGroupVec fancy array handle (described in Section 26.12) to make a flat array of indices look like an array of Vec objects. The following example shows what the DoExecute method in the associated filter would look like. Note the use make_ArrayHandleGroupVec when calling Invoke on line 20 to make this conversion.

Example 32.3: Invoking worklets to extract edges from a cell set.

```
 1 template<typename Policy>
 2 inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
 3   const vtkm::cont::DataSet& inData,
 4   vtkm::filter::PolicyBase<Policy> policy)
 5 {
 6
 7   const vtkm::cont::DynamicCellSet& inCellSet =
 8     vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), policy);
 9
10   // Count number of edges in each cell.
11   vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
12   this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);
13
14   // Build the scatter object (for non 1-to-1 mapping of input to output)
15   vtkm::worklet::ScatterCounting scatter(edgeCounts);
16   this->OutputToInputCellMap =
17     scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
18
19   vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
20   this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
21                scatter,
22                inCellSet,
23                vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));
24
25   vtkm::cont::CellSetSingleType<> outCellSet;
26   outCellSet.Fill(
27     inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);
28
29   vtkm::cont::DataSet outData;
30
31   outData.SetCellSet(outCellSet);
32
33   for (vtkm::IdComponent coordSystemIndex = 0;
34        coordSystemIndex < inData.GetNumberOfCoordinateSystems();
35        ++coordSystemIndex)
36   {
37     outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
38   }
39
40   return outData;
41 }
```

Another feature to note in Example 32.3 is that the method calls GetOutputToInputMap on the Scatter object it creates and squirrels the map array away for later use. The reason for this behavior is to implement mapping fields that are attached on the input cells to the indices of the output. In practice, DoExecute is called on DataSet objects to create new DataSet objects. The method in Example 32.3 creates a new CellSet, but we also need a method to transform the Fields on the data set. The saved OutputToInputCellMap array allows us to transform input fields to output fields.

The following example shows the implementation of DoMapField in the associated filter that takes this saved OutputToInputCellMap array and converts an array from an input cell field to an output cell field array. Note that we can do this by using the OutputToInputCellMap as an index array in a vtkm::cont::ArrayHandlePermutation.

Example 32.4: Converting cell fields using a simple permutation.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool ExtractEdges::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& inputArray,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field outputField;

  if (fieldMeta.IsPointField())
  {
    outputField = fieldMeta.AsField(inputArray); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    vtkm::cont::ArrayHandle<T> outputCellArray;
    vtkm::cont::ArrayCopy(vtkm::cont::make_ArrayHandlePermutation(
                            this->OutputToInputCellMap, inputArray),
                          outputCellArray);
    outputField = fieldMeta.AsField(outputCellArray);
  }
  else
  {
    return false;
  }

  result.AddField(outputField);

  return true;
}
```

32.2 Combining Like Elements

Our motivating example in Section 32.1 created a cell set with a line element representing each edge in some input data set. However, on close inspection there is a problem with our algorithm: it is generating a lot of duplicate elements. The cells in a typical mesh are connected to each other. As such, they share edges with each other. That is, the edge of one cell is likely to also be part of one or more other cells. When multiple cells contain the same edge, the algorithm we created in Section 32.1 will create multiple overlapping lines, one for each cell using the edge, as demonstrated in Figure 32.1. What we really want is to have one line for every edge in the mesh rather than many overlapping lines.

In this section we will re-implement the algorithm to generate a wireframe by creating a line for each edge, but this time we will merge duplicate edges together. Our first step is the same as before. We need to count the number of edges in each input cell and use those counts to create a vtkm::worklet::ScatterCounting for subsequent worklets. Counting the edges is a simple worklet.

Example 32.5: A simple worklet to count the number of edges on each cell.

```
struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
{
```

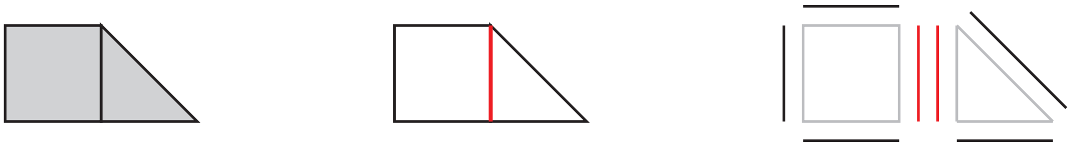


Figure 32.1: Duplicate lines from extracted edges. Consider the small mesh at the left comprising a square and a triangle. If we count the edges in this mesh, we would expect to get 6. However, our naïve implementation in Section 32.1 generates 7 because the shared edge (highlighted in red in the wireframe in the middle) is duplicated. As seen in the exploded view at right, one line is created for the square and one for the triangle.

```
  using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
  using ExecutionSignature = _2(CellShape, PointCount);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC_CONT vtkm::IdComponent operator()(
    CellShapeTag cellShape,
    vtkm::IdComponent numPointsInCell) const
  {
    return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
  }
};
```

In our previous version, we used the count to directly write out the lines. However, before we do that, we want to identify all the unique edges and identify which cells share this edge. This grouping is exactly the function that the reduce by key worklet type (described in Section 21.4) is designed to accomplish. The principal idea is to write a "key" that uniquely identifies the edge. The reduce by key worklet can then group the edges by the key and allow you to combine the data for the edge.

Thus, our goal of finding duplicate edges hinges on producing a key where two keys are identical if and only if the edges are the same. One straightforward key is to use the coordinates in 3D space by, say, computing the midpoint of the edge. The main problem with using this point coordinates approach is that a computer can hold a point coordinate only with floating point numbers of limited precision. Computer floating point computations are notorious for providing slightly different answers when the results should be the same. For example, if an edge has endpoints at p_1 and p_2 and two different cells compute the midpoint as $(p_1 + p_2)/2$ and $(p_2 + p_1)/2$, respectively, the answer is likely to be slightly different. When this happens, the keys will not be the same and we will still produce 2 edges in the output.

Fortunately, there is a better choice for keys based on the observation that in the original cell set each edge is specified by endpoints that each have unique indices. We can combine these 2 point indices to form a "canonical" descriptor of an edge (correcting for order).[1] VTK-m comes with a helper function, vtkm::exec::CellEdgeCanonicalId, defined in vtkm/exec/CellEdge.h, to produce these unique edge keys as vtkm::Id2 s. Our second worklet produces these canonical edge identifiers.

Example 32.6: Worklet generating canonical edge identifiers.

```
class EdgeIdsWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut canonicalIds);
  using ExecutionSignature = void(CellShape cellShape,
                                  PointIndices globalPointIndices,
                                  VisitIndex localEdgeIndex,
                                  _2 canonicalIdOut);
```

[1] Using indices to find common mesh elements is described by Miller et al. in "Finely-Threaded History-Based Topology Computation" (in *Eurographics Symposium on Parallel Graphics and Visualization*, June 2014).

```
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::IdComponent localEdgeIndex,
                            vtkm::Id2& canonicalIdOut) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();

    canonicalIdOut = vtkm::exec::CellEdgeCanonicalId(
      numPointsInCell, localEdgeIndex, cellShape, globalPointIndicesForCell, *this);
  }
};
```

Our third and final worklet generates the line cells by outputting the indices of each edge. As hinted at earlier, this worklet is a reduce by key worklet (inheriting from vtkm::worklet::WorkletReduceByKey). When the worklet is invoked, VTK-m will collect the unique keys and call the worklet once for each unique edge. Because there is no longer a consistent mapping from the generated lines to the elements of the input cell set, we need pairs of indices identifying the cells/edges from which the edge information comes. We use these indices along with a connectivity structure produced by a WholeCellSetIn to find the information about the edge. As shown later, these indices of cells and edges can be extracted from the ScatterCounting used to execute the worklet back in Example 32.6.

As we did in Section 32.1, this worklet writes out the edge information in a vtkm::Id2 (which in some following code will be created with an ArrayHandleGroupVec).

Example 32.7: A worklet to generate indices for line cells from combined edges.

```
class EdgeIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originEdges,
                                ReducedValuesOut connectivityOut);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originEdge,
                                  _5 connectivityOut);
  using InputDomain = _1;

  template<typename CellSetType, typename OriginCellsType, typename OriginEdgesType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginEdgesType& originEdges,
                            vtkm::Id2& connectivityOut) const
  {
    // Regardless of how many cells are sharing the edge we are generating, we
    // know that each cell/edge given to us by the reduce-by-key refers to the
    // same edge, so we can just look at the first cell to get the edge.
    vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(originCells[0]);
    vtkm::IdComponent edgeIndex = originEdges[0];
    auto cellShape = cellSet.GetCellShape(originCells[0]);

    vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 0, edgeIndex, cellShape, *this);
    vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 1, edgeIndex, cellShape, *this);

    auto globalPointIndicesForCell = cellSet.GetIndices(originCells[0]);
    connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
    connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
  }
};
```

Did you know?

It so happens that the vtkm::Id2 *s generated by* CellEdgeCanonicalId *contain the point indices of the two endpoints, which is enough information to create the edge. Thus, in this example it would be possible to forgo the steps of looking up indices through the cell set. That said, this is more often not the case, so for the purposes of this example we show how to construct cells without depending on the structure of the keys.*

With these 3 worklets, it is now possible to generate all the information we need to fill a vtkm::cont::CellSetSingleType object. A CellSetSingleType requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively.

The last item, the array of connection indices, is what we are creating with the worklet in Example 32.7. The connectivity array for CellSetSingleType is expected to be a flat array of vtkm::Id indices, but the worklet needs to provide groups of indices for each cell (in this case as a Vec object). To reconcile what the worklet provides and what the connectivity array must look like, we use the vtkm::cont::ArrayHandleGroupVec fancy array handle (described in Section 26.12) to make a flat array of indices look like an array of Vec objects. The following example shows what the DoExecute method in the associated filter would look like. Note the use of make_ArrayHandleGroupVec when calling Invoke on line 30 to make this conversion.

Example 32.8: Invoking worklets to extract unique edges from a cell set.

```
 1 template<typename Policy>
 2 inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
 3   const vtkm::cont::DataSet& inData,
 4   vtkm::filter::PolicyBase<Policy> policy)
 5 {
 6   const vtkm::cont::DynamicCellSet& inCellSet =
 7     vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), policy);
 8
 9   // First, count the edges in each cell.
10   vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
11   this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);
12
13   // Second, using these counts build a scatter that repeats a cell's visit
14   // for each edge in the cell.
15   vtkm::worklet::ScatterCounting scatter(edgeCounts);
16   this->OutputToInputCellMap =
17     scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
18   vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
19     scatter.GetVisitArray(inCellSet.GetNumberOfCells());
20
21   // Third, for each edge, extract a canonical id.
22   vtkm::cont::ArrayHandle<vtkm::Id2> canonicalIds;
23   this->Invoke(vtkm::worklet::EdgeIdsWorklet{}, scatter, inCellSet, canonicalIds);
24
25   // Fourth, construct a Keys object to combine all like edge ids.
26   this->CellToEdgeKeys = vtkm::worklet::Keys<vtkm::Id2>(canonicalIds);
```

```

  // Fifth, use a reduce-by-key to extract indices for each unique edge.
  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
               this->CellToEdgeKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputEdgeMap,
               vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  // Sixth, use the created connectivity array to build a cell set.
  vtkm::cont::CellSetSingleType<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  vtkm::cont::DataSet outData;

  outData.SetCellSet(outCellSet);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

Another feature to note in Example 32.8 is that the method calls GetOutputToInputMap on the Scatter object it creates and squirrels it away for later use. It also saves the vtkm::worklet::Keys object created for later use. The reason for this behavior is to implement mapping fields that are attached on the input cells to the indices of the output. In practice, DoExecute is called on DataSet objects to create new DataSet objects. The method in Example 32.8 creates a new CellSet, but we also need a method to transform the Fields on the data set. The saved OutputToInputCellMap array and Keys object allow us to transform input fields to output fields.

The following example shows the implementation of DoMapField in the associated filter that takes these saved objects and converts an array from an input cell field to an output cell field array. Because in general there are several cells that contribute to each edge/line in the output, we need a method to combine all these cell values to one. The most appropriate combination is likely an average of all the values. Because this is a common operation, VTK-m provides the vtkm::worklet::AverageByKey class to help perform exactly this operation. AverageByKey provides a Run method that takes a Keys object, an array of in values, and a device adapter tag and produces and array of values averaged by key.

Example 32.9: Converting cell fields that average collected values.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool ExtractEdges::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& inputArray,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field outputField;

  if (fieldMeta.IsPointField())
  {
    outputField = fieldMeta.AsField(inputArray); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    auto outputCellArray =
      vtkm::worklet::AverageByKey::Run(this->CellToEdgeKeys,
                                       vtkm::cont::make_ArrayHandlePermutation(
                                         this->OutputToInputCellMap, inputArray));
    outputField = fieldMeta.AsField(outputCellArray);
  }
  else
  {
    return false;
  }

  result.AddField(outputField);

  return true;
}
```

32.3 Faster Combining Like Elements with Hashes

In the previous two sections we constructed worklets that took a cell set and created a new set of cells that represented the edges of the original cell set, which can provide a wireframe of the mesh. In Section 32.1 we provided a pair of worklets that generate one line per edge per cell. In Section 32.2 we improved on this behavior by using a reduce by key worklet to find and merge shared edges.

If we were to time all the operations run in the later implementation to generate the wireframe (i.e. the operations in Example 32.8), we would find that the vast majority of the time is not spent in the actual worklets. Rather, the majority of the time is spent in collecting the like keys, which happens in the constructor of the vtkm::worklet::Keys object. Internally, keys are collected by sorting them. The most fruitful way to improve the performance of this algorithm is to improve the sorting behavior.

The details of how the sort works is dependent on the inner workings of the device adapter. It turns out that the performance of the sort of the keys is highly dependent on the data type of the keys. For example, sorting numbers stored in a 32-bit integer is often much faster than sorting groups of 2 or 3 64-bit integer. This is particularly true when the sort is capable of performing a radix-based sort.

An easy way to convert collections of indices like those returned from vtkm::exec::CellEdgeCanonicalId to a 32-bit integer is to use a hash function. To facilitate the creation of hash values, VTK-m comes with a simple vtkm::Hash function (in the vtkm/Hash.h header file). Hash takes a Vec or Vec-like object of integers and returns a value of type vtkm::HashType (an alias for a 32-bit integer). This hash function uses the FNV-1a algorithm that is designed to create hash values that are quasi-random but deterministic. This means that hash values of two different identifiers are unlikely to be the same.

That said, hash collisions can happen and become increasingly likely on larger data sets. Therefore, if we wish to use hash values, we also have to add conditions that manage collisions when they happen. Resolving hash value collisions adds overhead, but the time saved in faster sorting of hash values generally outweighs the overhead added by resolving collisions.[2] In this section we will improve on the implementation given in Section 32.2 by using hash values for keys and resolving for collisions.

As always, our first step is to count the number of edges in each input cell. These counts are used to create a vtkm::worklet::ScatterCounting for subsequent worklets.

Example 32.10: A simple worklet to count the number of edges on each cell.

```
struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
{
```

[2] A comparison of the time required for completely unique keys and hash keys with collisions is studied by Lessley, et al. in "Techniques for Data-Parallel Searching for Duplicate Elements" (in *IEEE Symposium on Large Data Analysis and Visualization*, October 2017).

```
  using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
  using ExecutionSignature = _2(CellShape, PointCount);
  using InputDomain = _1;

  template<typename CellShapeTag>
  VTKM_EXEC_CONT vtkm::IdComponent operator()(
    CellShapeTag cellShape,
    vtkm::IdComponent numPointsInCell) const
  {
    return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
  }
};
```

Our next step is to generate keys that can be used to find like elements. As before, we will use the vtkm::exec::CellEdgeCanonicalId function to create a unique representation for each edge. However, rather than directly use the value from CellEdgeCanonicalId, which is a vtkm::Id2, we will instead use that to generate a hash value.

Example 32.11: Worklet generating hash values.

```
class EdgeHashesWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut hashValues);
  using ExecutionSignature = _2(CellShape cellShape,
                                PointIndices globalPointIndices,
                                VisitIndex localEdgeIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC vtkm::HashType operator()(
    CellShapeTag cellShape,
    const PointIndexVecType& globalPointIndicesForCell,
    vtkm::IdComponent localEdgeIndex) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();
    return vtkm::Hash(vtkm::exec::CellEdgeCanonicalId(
      numPointsInCell, localEdgeIndex, cellShape, globalPointIndicesForCell, *this));
  }
};
```

The hash values generated by the worklet in Example 32.11 will be the same for two identical edges. However, it is no longer guaranteed that two distinct edges will have different keys, and collisions of this nature become increasingly common for larger cell sets. Thus, our next step is to resolve any such collisions.

The following example provides a worklet that goes through each group of edges associated with the same hash value (using a reduce by key worklet). It identifies which edges are actually the same as which other edges, marks a local identifier for each unique edge group, and returns the number of unique edges associated with the hash value.

Example 32.12: Worklet to resolve hash collisions occurring on edge identifiers.

```
class EdgeHashCollisionsWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originEdges,
                                ValuesOut localEdgeIndices,
                                ReducedValuesOut numEdges);
  using ExecutionSignature = _6(_2 inputCells,
                                _3 originCells,
                                _4 originEdges,
                                _5 localEdgeIndices);
  using InputDomain = _1;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginEdgesType,
           typename localEdgeIndicesType>
  VTKM_EXEC vtkm::IdComponent operator()(
    const CellSetType& cellSet,
    const OriginCellsType& originCells,
    const OriginEdgesType& originEdges,
    localEdgeIndicesType& localEdgeIndices) const
  {
    vtkm::IdComponent numEdgesInHash = localEdgeIndices.GetNumberOfComponents();

    // Sanity checks.
    VTKM_ASSERT(originCells.GetNumberOfComponents() == numEdgesInHash);
    VTKM_ASSERT(originEdges.GetNumberOfComponents() == numEdgesInHash);

    // Clear out localEdgeIndices
    for (vtkm::IdComponent index = 0; index < numEdgesInHash; ++index)
    {
      localEdgeIndices[index] = -1;
    }

    // Count how many unique edges there are and create an id for each;
    vtkm::IdComponent numUniqueEdges = 0;
    for (vtkm::IdComponent firstEdgeIndex = 0; firstEdgeIndex < numEdgesInHash;
         ++firstEdgeIndex)
    {
      if (localEdgeIndices[firstEdgeIndex] == -1)
      {
        vtkm::IdComponent edgeId = numUniqueEdges;
        localEdgeIndices[firstEdgeIndex] = edgeId;
        // Find all matching edges.
        vtkm::Id firstCellIndex = originCells[firstEdgeIndex];
        vtkm::Id2 canonicalEdgeId =
          vtkm::exec::CellEdgeCanonicalId(cellSet.GetNumberOfIndices(firstCellIndex),
                                          originEdges[firstEdgeIndex],
                                          cellSet.GetCellShape(firstCellIndex),
                                          cellSet.GetIndices(firstCellIndex),
                                          *this);
        for (vtkm::IdComponent laterEdgeIndex = firstEdgeIndex + 1;
             laterEdgeIndex < numEdgesInHash;
             ++laterEdgeIndex)
        {
          vtkm::Id laterCellIndex = originCells[laterEdgeIndex];
          vtkm::Id2 otherCanonicalEdgeId = vtkm::exec::CellEdgeCanonicalId(
            cellSet.GetNumberOfIndices(laterCellIndex),
            originEdges[laterEdgeIndex],
            cellSet.GetCellShape(laterCellIndex),
            cellSet.GetIndices(laterCellIndex),
            *this);
          if (canonicalEdgeId == otherCanonicalEdgeId)
          {
            localEdgeIndices[laterEdgeIndex] = edgeId;
          }
        }
        ++numUniqueEdges;
      }
    }

    return numUniqueEdges;
  }
};
```

With all hash collisions correctly identified, we are ready to generate the connectivity array for the line elements. This worklet uses a reduce by key worklet like the previous example, but this time we use a ScatterCounting to run the worklet multiple times for hash values that contain multiple unique edges. The worklet takes all the information it needs to reference back to the edges in the original mesh including a WholeCellSetIn, look back indices for the cells and respective edges, and the unique edge group indicators produced by Example 32.11.

As in the previous sections, this worklet writes out the edge information in a vtkm::Id2 (which in some following code will be created with an ArrayHandleGroupVec).

Example 32.13: A worklet to generate indices for line cells from combined edges and potential collisions.

```
class EdgeIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originEdges,
                                ValuesIn localEdgeIndices,
                                ReducedValuesOut connectivityOut);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originEdge,
                                  _5 localEdgeIndices,
                                  VisitIndex localEdgeIndex,
                                  _6 connectivityOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginEdgesType,
           typename LocalEdgeIndicesType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginEdgesType& originEdges,
                            const LocalEdgeIndicesType& localEdgeIndices,
                            vtkm::IdComponent localEdgeIndex,
                            vtkm::Id2& connectivityOut) const
  {
    // Find the first edge that matches the index given and return it.
    for (vtkm::IdComponent edgeIndex = 0;; ++edgeIndex)
    {
      if (localEdgeIndices[edgeIndex] == localEdgeIndex)
      {
        vtkm::Id cellIndex = originCells[edgeIndex];
        vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
        vtkm::IdComponent edgeInCellIndex = originEdges[edgeIndex];
        auto cellShape = cellSet.GetCellShape(cellIndex);

        vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
          numPointsInCell, 0, edgeInCellIndex, cellShape, *this);
        vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
          numPointsInCell, 1, edgeInCellIndex, cellShape, *this);

        auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
        connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
        connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];

        break;
      }
    }
  }
};
```

With these 3 worklets, it is now possible to generate all the information we need to fill a vtkm::cont::CellSetSingleType object. A CellSetSingleType requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively.

The last item, the array of connection indices, is what we are creating with the worklet in Example 32.13. The connectivity array for CellSetSingleType is expected to be a flat array of vtkm::Id indices, but the worklet needs to provide groups of indices for each cell (in this case as a Vec object). To reconcile what the worklet provides and what the connectivity array must look like, we use the vtkm::cont::ArrayHandleGroupVec fancy array handle (described in Section 26.12) to make a flat array of indices look like an array of Vec objects.

The following example shows what the DoExecute method in the associated filter would look like.

Example 32.14: Invoking worklets to extract unique edges from a cell set using hash values.

```
template<typename Policy>
inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
  const vtkm::cont::DataSet& inData,
  vtkm::filter::PolicyBase<Policy> policy)
{
  const vtkm::cont::DynamicCellSet& inCellSet =
    vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), policy);

  // First, count the edges in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);

  // Second, using these counts build a scatter that repeats a cell's visit
  // for each edge in the cell.
  vtkm::worklet::ScatterCounting scatter(edgeCounts);
  this->OutputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
  vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
    scatter.GetVisitArray(inCellSet.GetNumberOfCells());

  // Third, for each edge, extract a hash.
  vtkm::cont::ArrayHandle<vtkm::HashType> hashValues;
  this->Invoke(vtkm::worklet::EdgeHashesWorklet{}, scatter, inCellSet, hashValues);

  // Fourth, use a Keys object to combine all like hashes.
  this->CellToEdgeKeys = vtkm::worklet::Keys<vtkm::HashType>(hashValues);

  // Fifth, use a reduce-by-key to collect like hash values, resolve collisions,
  // and count the number of unique edges associated with each hash.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> numUniqueEdgesInEachHash;
  this->Invoke(vtkm::worklet::EdgeHashCollisionsWorklet{},
               this->CellToEdgeKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputEdgeMap,
               this->LocalEdgeIndices,
               numUniqueEdgesInEachHash);

  // Sixth, use a reduce-by-key to extract indices for each unique edge.
  // (Note that HashCollisionScatter is an std::shared_ptr because we cannot copy
  // ScatterCounting objects.)
  this->HashCollisionScatter.reset(
    new vtkm::worklet::ScatterCounting(numUniqueEdgesInEachHash));

  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
               *this->HashCollisionScatter,
               this->CellToEdgeKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputEdgeMap,
               this->LocalEdgeIndices,
               vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  // Seventh, use the created connectivity array to build a cell set.
  vtkm::cont::CellSetSingleType<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  vtkm::cont::DataSet outData;

  outData.SetCellSet(outCellSet);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

As noted in Section 32.2, in practice `DoExecute` is called on `DataSet` objects to create new `DataSet` objects. Although Example 32.14 creates a new `CellSet`, we also need a method to transform the `Field`s on the data set. To do this, we need to save some information. This includes the map from the edge of each cell to its origin cell (`OutputToInputCellMap`), a `Keys` object on the hash values (`CellToEdgeKeys`), an array of indices resolving collisions (`LocalEdgeIndices`), and a `ScatterCounting` to repeat edge outputs when unique edges collide on a hash (`HashCollisionScatter`).

Common Errors

Note that `HashCollisionScatter` *is an object of type* `vtkm::worklet::ScatterCounting`*, which does not have a default constructor. That can be a problem since you do not have the data to construct* `HashCollisionScatter` *until* `DoExecute` *is called. To get around this chicken-and-egg issue, it is best to store the* `ScatterCounter` *as a pointer. It is best practice to use a smart pointer, like* `std::shared_ptr` *to manage it.*

In Section 32.2 we used a convenience method to average a field attached to cells on the input to each unique edge in the output. Unfortunately, that function does not take into account the collisions that can occur on the keys. Instead we need a custom worklet to average those values that match the same unique edge.

Example 32.15: A worklet to average values with the same key, resolving for collisions.

```
class AverageCellEdgesFieldWorklet : public vtkm::worklet::WorkletReduceByKey
{
```

```
public:
  using ControlSignature = void(KeysIn keys,
                                ValuesIn inFieldValues,
                                ValuesIn localEdgeIndices,
                                ReducedValuesOut averagedField);
  using ExecutionSignature = _4(_2 inFieldValues,
                                _3 localEdgeIndices,
                                VisitIndex localEdgeIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename InFieldValuesType, typename LocalEdgeIndicesType>
  VTKM_EXEC typename InFieldValuesType::ComponentType operator()(
    const InFieldValuesType& inFieldValues,
    const LocalEdgeIndicesType& localEdgeIndices,
    vtkm::IdComponent localEdgeIndex) const
  {
    using FieldType = typename InFieldValuesType::ComponentType;

    FieldType averageField = FieldType(0);
    vtkm::IdComponent numValues = 0;
    for (vtkm::IdComponent reduceIndex = 0;
         reduceIndex < inFieldValues.GetNumberOfComponents();
         ++reduceIndex)
    {
      if (localEdgeIndices[reduceIndex] == localEdgeIndex)
      {
        FieldType fieldValue = inFieldValues[reduceIndex];
        averageField = averageField + fieldValue;
        ++numValues;
      }
    }
    VTKM_ASSERT(numValues > 0);
    return static_cast<FieldType>(averageField / numValues);
  }
};
```

With this worklet, it is straightforward to process cell fields.

Example 32.16: Invoking the worklet to process cell fields, resolving for collisions.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool ExtractEdges::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& inputArray,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field outputField;

  if (fieldMeta.IsPointField())
  {
    outputField = fieldMeta.AsField(inputArray); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    vtkm::cont::ArrayHandle<T> averageField;
    this->Invoke(vtkm::worklet::AverageCellEdgesFieldWorklet{},
                 *this->HashCollisionScatter,
                 this->CellToEdgeKeys,
                 vtkm::cont::make_ArrayHandlePermutation(this->OutputToInputCellMap,
                                                         inputArray),
                 this->LocalEdgeIndices,
                 averageField);
    outputField = fieldMeta.AsField(averageField);
  }
  else
  {
    return false;
  }

  result.AddField(outputField);

  return true;
}
```

32.4 Variable Cell Types

So far in our previous examples we have demonstrated creating a cell set where every cell is the same shape and number of points (i.e. a CellSetSingleType). However, it can also be the case where an algorithm must create cells of a different type (into a vtkm::cont::CellSetExplicit). The procedure for generating cells of different shapes is similar to that of creating a single shape. There is, however, an added step of counting the size (in number of points) of each shape to build the appropriate structure for storing the cell connectivity.

Our motivating example is a filter that extracts all the unique faces in a cell set and stores them in a cell set of polygons. This problem is similar to the one addressed in Sections 32.1, 32.2, and 32.3. In both cases it is necessary to find all subelements of each cell (in this case the faces instead of the edges). It is also the case that we expect many faces to be shared among cells in the same way edges are shared among cells. We will use the hash-based approach demonstrated in Section 32.3 except this time applied to faces instead of edges.

The main difference between the two extraction tasks is that whereas all edges are lines with two points, faces can come in different sizes. A tetrahedron has triangular faces whereas a hexahedron has quadrilateral faces. Pyramid and wedge cells have both triangular and quadrilateral faces. Thus, in general the algorithm must be capable of outputing multiple cell types.

Our algorithm for extracting unique cell faces follows the same algorithm as that in Section 32.3. We first need three worklets (used in succession) to count the number of faces in each cell, to generate a hash value for each face, and to resolve hash collisions. These are essentially the same as Examples 32.10, 32.11, and 32.12, respectively, with superficial changes made (like changing Edge to Face). To make it simpler to follow the discussion, the code is not repeated here.

When extracting edges, these worklets provide everything necessary to write out line elements. However, before we can write out polygons of different sizes, we first need to count the number of points in each polygon. The following example does just that. This worklet also writes out the identifier for the shape of the face, which we will eventually require to build a CellSetExplicit. Also recall that we have to work with the information returned from the collision resolution to report on the appropriate unique cell face.

Example 32.17: A worklet to count the points in the final cells of extracted faces

```
class CountPointsInFaceWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originFaces,
                                ValuesIn localFaceIndices,
                                ReducedValuesOut faceShape,
                                ReducedValuesOut numPointsInEachFace);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originFace,
                                  _5 localFaceIndices,
                                  VisitIndex localFaceIndex,
                                  _6 faceShape,
                                  _7 numPointsInFace);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginFacesType,
           typename LocalFaceIndicesType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginFacesType& originFaces,
                            const LocalFaceIndicesType& localFaceIndices,
                            vtkm::IdComponent localFaceIndex,
                            vtkm::UInt8& faceShape,
                            vtkm::IdComponent& numPointsInFace) const
  {
    // Find the first face that matches the index given.
    for (vtkm::IdComponent faceIndex = 0;; ++faceIndex)
    {
      if (localFaceIndices[faceIndex] == localFaceIndex)
      {
        vtkm::Id cellIndex = originCells[faceIndex];
        faceShape = vtkm::exec::CellFaceShape(
          originFaces[faceIndex], cellSet.GetCellShape(cellIndex), *this);
        numPointsInFace = vtkm::exec::CellFaceNumberOfPoints(
          originFaces[faceIndex], cellSet.GetCellShape(cellIndex), *this);
        break;
      }
    }
  }
};
```

When extracting edges, we converted a flat array of connectivity information to an array of Vecs using an ArrayHandleGroupVec. However, ArrayHandleGroupVec can only create Vecs of a constant size. Instead, for this use case we need to use vtkm::cont::ArrayHandleGroupVecVariable. As described in Section 26.12, ArrayHandleGroupVecVariable takes a flat array of values and an index array of offsets that points to the beginning of each group to represent as a Vec-like. The worklet in Example 32.17 does not actually give us the array of offsets we need. Rather, it gives us the count of each group. We can get the offsets from the counts by using the vtkm::cont::ConvertNumIndicesToOffsets convenience function.

Example 32.18: Converting counts of connectivity groups to offsets for ArrayHandleGroupVecVariable.

```
  vtkm::cont::ArrayHandle<vtkm::Id> offsetsExtended;
  vtkm::Id connectivityArraySize;
  vtkm::cont::ConvertNumIndicesToOffsets(
    numPointsInEachFace, offsetsExtended, connectivityArraySize);

  auto offsetsExclusive = vtkm::cont::make_ArrayHandleView(
    offsetsExtended, 0, offsetsExtended.GetNumberOfValues() - 1);

  vtkm::cont::CellSetExplicit<>::ConnectivityArrayType connectivityArray;
  connectivityArray.Allocate(connectivityArraySize);
  auto connectivityArrayVecs = vtkm::cont::make_ArrayHandleGroupVecVariable(
    connectivityArray, offsetsExclusive);
```

Once we have created an ArrayHandleGroupVecVariable, we can pass that to a worklet that produces the point connections for each output polygon. The worklet is very similar to the one for creating edge lines (shown in

Example 32.13), but we have to correctly handle the Vec-like of unknown type and size.

Example 32.19: A worklet to generate indices for polygon cells of different sizes from combined edges and potential collisions.

```
class FaceIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originFaces,
                                ValuesIn localFaceIndices,
                                ReducedValuesOut connectivityOut);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originFace,
                                  _5 localFaceIndices,
                                  VisitIndex localFaceIndex,
                                  _6 connectivityOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginFacesType,
           typename LocalFaceIndicesType,
           typename ConnectivityVecType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginFacesType& originFaces,
                            const LocalFaceIndicesType& localFaceIndices,
                            vtkm::IdComponent localFaceIndex,
                            ConnectivityVecType& connectivityOut) const
  {
    // Find the first face that matches the index given and return it.
    for (vtkm::IdComponent faceIndex = 0;; ++faceIndex)
    {
      if (localFaceIndices[faceIndex] == localFaceIndex)
      {
        vtkm::Id cellIndex = originCells[faceIndex];
        vtkm::IdComponent faceInCellIndex = originFaces[faceIndex];
        auto cellShape = cellSet.GetCellShape(cellIndex);
        vtkm::IdComponent numPointsInFace = connectivityOut.GetNumberOfComponents();

        VTKM_ASSERT(
          numPointsInFace ==
          vtkm::exec::CellFaceNumberOfPoints(faceInCellIndex, cellShape, *this));

        auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
        for (vtkm::IdComponent localPointI = 0; localPointI < numPointsInFace;
             ++localPointI)
        {
          vtkm::IdComponent pointInCellIndex = vtkm::exec::CellFaceLocalIndex(
            localPointI, faceInCellIndex, cellShape, *this);
          connectivityOut[localPointI] = globalPointIndicesForCell[pointInCellIndex];
        }

        break;
      }
    }
  }
};
```

With these worklets in place, we can implement a filter DoExecute as follows.

Example 32.20: Invoking worklets to extract unique faces from a cell set.

```
template<typename Policy>
inline VTKM_CONT vtkm::cont::DataSet ExtractFaces::DoExecute(
  const vtkm::cont::DataSet& inData,
  vtkm::filter::PolicyBase<Policy> policy)
{

  const vtkm::cont::DynamicCellSet& inCellSet =
    vtkm::filter::ApplyPolicyCellSet(inData.GetCellSet(), policy);

  // First, count the faces in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> faceCounts;
  this->Invoke(vtkm::worklet::CountFacesWorklet{}, inCellSet, faceCounts);

  // Second, using these counts build a scatter that repeats a cell’s visit
  // for each edge in the cell.
  vtkm::worklet::ScatterCounting scatter(faceCounts);
  this->OutputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
  vtkm::worklet::ScatterCounting::VisitArrayType outputToInputFaceMap =
    scatter.GetVisitArray(inCellSet.GetNumberOfCells());

  // Third, for each face, extract a hash.
  vtkm::cont::ArrayHandle<vtkm::HashType> hashValues;
  this->Invoke(vtkm::worklet::FaceHashesWorklet{}, scatter, inCellSet, hashValues);

  // Fourth, use a Keys object to combine all like hashes.
  this->CellToFaceKeys = vtkm::worklet::Keys<vtkm::HashType>(hashValues);

  // Fifth, use a reduce-by-key to collect like hash values, resolve collisions,
  // and count the number of unique faces associated with each hash.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> numUniqueFacesInEachHash;
  this->Invoke(vtkm::worklet::FaceHashCollisionsWorklet{},
               this->CellToFaceKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputFaceMap,
               this->LocalFaceIndices,
               numUniqueFacesInEachHash);

  // Sixth, use a reduce-by-key to count the number of points in each unique face.
  // Also identify the shape of each face.
  this->HashCollisionScatter.reset(
    new vtkm::worklet::ScatterCounting(numUniqueFacesInEachHash));

  vtkm::cont::CellSetExplicit<>::ShapesArrayType shapeArray;
  vtkm::cont::ArrayHandle<vtkm::IdComponent> numPointsInEachFace;

  this->Invoke(vtkm::worklet::CountPointsInFaceWorklet{},
               *this->HashCollisionScatter,
               this->CellToFaceKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputFaceMap,
               this->LocalFaceIndices,
               shapeArray,
               numPointsInEachFace);

  // Seventh, convert the numPointsInEachFace array to an offsets array and use that
  // to create an ArrayHandleGroupVecVariable.
  vtkm::cont::ArrayHandle<vtkm::Id> offsetsExtended;
  vtkm::Id connectivityArraySize;
  vtkm::cont::ConvertNumIndicesToOffsets(
    numPointsInEachFace, offsetsExtended, connectivityArraySize);

  auto offsetsExclusive = vtkm::cont::make_ArrayHandleView(
    offsetsExtended, 0, offsetsExtended.GetNumberOfValues() - 1);

  vtkm::cont::CellSetExplicit<>::ConnectivityArrayType connectivityArray;
  connectivityArray.Allocate(connectivityArraySize);
  auto connectivityArrayVecs = vtkm::cont::make_ArrayHandleGroupVecVariable(
    connectivityArray, offsetsExclusive);

  // Eigth, use a reduce-by-key to extract indices for each unique face.
  this->Invoke(vtkm::worklet::FaceIndicesWorklet{},
               *this->HashCollisionScatter,
               this->CellToFaceKeys,
               inCellSet,
               this->OutputToInputCellMap,
               outputToInputFaceMap,
               this->LocalFaceIndices,
               connectivityArrayVecs);

  // Ninth, use the created connectivity array and others to build a cell set.
  vtkm::cont::CellSetExplicit<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), shapeArray, connectivityArray, offsetsExtended);

  vtkm::cont::DataSet outData;

  outData.SetCellSet(outCellSet);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

As noted previously, in practice `DoExecute` is called on `DataSet` objects to create new `DataSet` objects. The process for doing so is no different from our previous algorithm as described at the end of Section 32.3 (Examples 32.15 and 32.16).

CHAPTER

THIRTYTHREE

VARIANT ARRAY HANDLES

The ArrayHandle class uses templating to make very efficient and type-safe access to data. However, it is sometimes inconvenient or impossible to specify the element type and storage at run-time. The VariantArrayHandle class provides a mechanism to manage arrays of data with unspecified types.

vtkm::cont::VariantArrayHandle holds a reference to an array. Unlike ArrayHandle, VariantArrayHandle is *not* templated. Instead, it uses C++ run-type type information to store the array without type and cast it when appropriate.

A VariantArrayHandle can be established by constructing it with or assigning it to an ArrayHandle. The following example demonstrates how a VariantArrayHandle might be used to load an array whose type is not known until run-time.

Example 33.1: Creating a VariantArrayHandle.

```
VTKM_CONT
vtkm::cont::VariantArrayHandle LoadVariantArray(const void* buffer,
                                                vtkm::Id length,
                                                std::string type)
{
  vtkm::cont::VariantArrayHandle handle;
  if (type == "float")
  {
    vtkm::cont::ArrayHandle<vtkm::Float32> concreteArray =
      vtkm::cont::make_ArrayHandle(reinterpret_cast<const vtkm::Float32*>(buffer),
                                   length);
    handle = concreteArray;
  }
  else if (type == "int")
  {
    vtkm::cont::ArrayHandle<vtkm::Int32> concreteArray =
      vtkm::cont::make_ArrayHandle(reinterpret_cast<const vtkm::Int32*>(buffer),
                                   length);
    handle = concreteArray;
  }
  return handle;
}
```

33.1 Querying and Casting

Data pointed to by a VariantArrayHandle is not directly accessible. However, there are a few generic queries you can make without directly knowing the data type. The GetNumberOfValues method returns the length of the array with respect to its base data type. It is also common in VTK-m to use data types, such as vtkm::Vec, with multiple components per value. The GetNumberOfComponents method returns the number of components in a vector-like type (or 1 for scalars).

Example 33.2: Non type-specific queries on VariantArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::VariantArrayHandle scalarDynamicHandle(
  vtkm::cont::make_ArrayHandle(scalarBuffer));

// This returns 10.
vtkm::Id scalarArraySize = scalarDynamicHandle.GetNumberOfValues();

// This returns 1.
vtkm::IdComponent scalarComponents = scalarDynamicHandle.GetNumberOfComponents();

std::vector<vtkm::Vec3f_32> vectorBuffer(20);
vtkm::cont::VariantArrayHandle vectorDynamicHandle(
  vtkm::cont::make_ArrayHandle(vectorBuffer));

// This returns 20.
vtkm::Id vectorArraySize = vectorDynamicHandle.GetNumberOfValues();

// This returns 3.
vtkm::IdComponent vectorComponents = vectorDynamicHandle.GetNumberOfComponents();
```

It is also often desirable to create a new array based on the underlying type of a VariantArrayHandle. For example, when a filter creates a field, it is common to make this output field the same type as the input. To satisfy this use case, VariantArrayHandle has a method named NewInstance that creates a new empty array with the same underlying type as the original array.

Example 33.3: Using NewInstance.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::VariantArrayHandle variantHandle(
  vtkm::cont::make_ArrayHandle(scalarBuffer));

// This creates a new empty array of type Float32.
vtkm::cont::VariantArrayHandle newVariantArray = variantHandle.NewInstance();
```

Before the data with a VariantArrayHandle can be accessed, the type of the array must be established. This is usually done internally within VTK-m when a worklet or filter is invoked and the VariantArrayHandle is converted into an ArrayHandleVirtual. However, it is also possible to query the types and cast to a concrete ArrayHandle.

You can query the component type using the VariantArrayHandle::IsValueType method. IsValueType takes a value type and returns whether that matches the underlying array. You can query the component type and storage type using the VariantArrayHandle::IsType method. IsType takes an example array handle type and returns whether the underlying array matches the given static array type.

Example 33.4: Querying the component and storage types of a VariantArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::ArrayHandle<vtkm::Float32> concreteHandle =
  vtkm::cont::make_ArrayHandle(scalarBuffer);
vtkm::cont::VariantArrayHandle variantHandle(concreteHandle);

// This returns true
bool isFloat32Array = variantHandle.IsType<decltype(concreteHandle)>();

// This returns false
bool isIdArray = variantHandle.IsType<vtkm::cont::ArrayHandle<vtkm::Id>>();
```

Once the type of the VariantArrayHandle is known, it can be cast to either to ArrayHandleVirtual or a concrete ArrayHandle, which has access to the data as described in Chapter 27. The easiest ways to do this is to use AsVirtual when desiring an ArrayHandleVirtual or CopyTo method when wanting a concrete ArrayHandle.

The VariantArrayHandle::AsVirtual templated method takes a value type as a template parameter and returns an array handle virtual that points to the array in VariantArrayHandle. If the given types are incorrect, then AsVirtual throws a vtkm::cont::ErrorControlBadValue exception.

Example 33.5: Casting a VariantArrayHandle to a virtual ArrayHandle.

```
vtkm::cont::ArrayHandleVirtual<vtkm::Float32> virtualArray =
  variantHandle.AsVirtual<vtkm::Float32>();
```

The VariantArrayHandle::CopyTo templated method takes a reference to an ArrayHandle as an argument and sets that array handle to point to the array in VariantArrayHandle. If the given types are incorrect, then CopyTo throws a vtkm::cont::ErrorControlBadValue exception.

Example 33.6: Casting a VariantArrayHandle to a concrete ArrayHandle.

```
variantHandle.CopyTo(concreteHandle);
```

Common Errors

Remember that ArrayHandle and VariantArrayHandle represent pointers to the data, so this "copy" is a shallow copy. There is still only one copy of the data, and if you change the data in one array handle that change is reflected in the other.

33.2 Casting to Unknown Types

Using AsVirtual, and CopyTo are fine as long as the correct types are known, but often times they are not. For this use case VariantArrayHandle has a method named CastAndCall that attempts to cast the array to some set of types.

The CastAndCall method accepts a functor to run on the appropriately cast array. The functor must have an overloaded const parentheses operator that accepts an ArrayHandle of the appropriate type.

Example 33.7: Operating on VariantArrayHandle with CastAndCall.

```
struct PrintArrayContentsFunctor
{
  template<typename T, typename S>
  VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<T, S>& array) const
  {
    this->PrintArrayPortal(array.GetPortalConstControl());
  }

private:
  template<typename PortalType>
  VTKM_CONT void PrintArrayPortal(const PortalType& portal) const
  {
    for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
    {
      // All ArrayPortal objects have ValueType for the type of each value.
      using ValueType = typename PortalType::ValueType;

      ValueType value = portal.Get(index);

      vtkm::IdComponent numComponents =
        vtkm::VecTraits<ValueType>::GetNumberOfComponents(value);
      for (vtkm::IdComponent componentIndex = 0; componentIndex < numComponents;
           componentIndex++)
      {
        std::cout << " "
                  << vtkm::VecTraits<ValueType>::GetComponent(value, componentIndex);
      }
      std::cout << std::endl;
    }
  }
};

template<typename VariantArrayType>
void PrintArrayContents(const VariantArrayType& array)
{
  array.CastAndCall(PrintArrayContentsFunctor());
}
```

Common Errors

It is possible to store any form of `ArrayHandle` *in a* `VariantArrayHandle`*, but it is not possible for* `CastAndCall` *to check every possible form of* `ArrayHandle`*. If* `CastAndCall` *cannot determine the* `ArrayHandle` *type, then an* `ErrorControlBadValue` *is thrown. The following section describes how to specify the forms of* `ArrayHandle` *to try.*

33.3 Specifying Cast Lists

The `CastAndCall` method can only check a finite number of value types. The default form of `CastAndCall` uses a default set of common types. These default lists can be overridden using the VTK-m list tags facility, which is discussed at length in Section 19.7.

Common type lists for value are defined in vtkm/TypeListTag.h and are documented in Section 19.7.2. This header also defines `VTKM_DEFAULT_TYPE_LIST_TAG`, which defines the default list of value types tried in `CastAndCall`.

There is a form of `CastAndCall` that accepts a tag for the list of component types. This can be used when the specific list is known at the time of the call. However, when creating generic operations like the `PrintArrayContents` function in Example 33.7, passing these tag is inconvenient at best.

To address this use case, `VariantArrayHandle` has a method named `ResetTypes`. This method returns a new object that behaves just like a `VariantArrayHandle` with identical state except that the cast and call functionality uses the specified component types instead of the default. (Note that `PrintArrayContents` in Example 33.7 is templated on the type of `VariantArrayHandle`. This is to accommodate using the objects from the `ResetTypes` method, which have the same behavior but different type names.)

So the default component type list contains a subset of the basic VTK-m types. If you wanted to accommodate more types, you could use `ResetTypes`.

Example 33.8: Trying all component types in a `VariantArrayHandle`.

```
  PrintArrayContents(dynamicArray.ResetTypes(vtkm::TypeListTagAll()));
```

Likewise, if you happen to know a particular type of the variant array, that can be specified to reduce the amount of object code created by templates in the compiler.

Example 33.9: Specifying a single component type in a VariantArrayHandle.

```
  PrintArrayContents(dynamicArray.ResetTypes(vtkm::TypeListTagId()));
```

Common Errors

VariantArrayHandle::ResetTypes *does not change the object. Rather, it returns a new object with different type information. This method has no effect unless you do something with the returned value.*

ResetTypes works by returning a vtkm::cont::VariantArrayHandleBase object. VariantArrayHandleBase specifies the value tag as a template argument and otherwise behaves just like VariantArrayHandle.

Did you know?

I lied earlier when I said at the beginning of this chapter that VariantArrayHandle *is a class that is not templated. This symbol is really just a type alias of* VariantArrayHandleBase*. Because the* VariantArrayHandle *fully specifies the template arguments, it behaves like a class, but if you get a compiler error it will show up as* VariantArrayHandleBase*.*

Most code does not need to worry about working directly with VariantArrayHandleBase. However, it is sometimes useful to declare it in templated functions that accept variant array handles so that works with every type list. The function in Example 33.7 did this by making the variant array handle class itself the template argument. This will work, but it is prone to error because the template will resolve to any type of argument. When passing objects that are not variant array handles will result in strange and hard to diagnose errors. Instead, we can define the same function using VariantArrayHandleBase so that the template will only match variant array handle types.

Example 33.10: Using VariantArrayHandleBase to accept generic variant array handles.

```
template<typename TypeList>
void PrintArrayContents(const vtkm::cont::VariantArrayHandleBase<TypeList>& array)
{
  array.CastAndCall(PrintArrayContentsFunctor());
}
```

CHAPTER

THIRTYFOUR

DEVICE ALGORITHMS

As described in Chapter 15, VTK-m is built around the concept of a *device adapter* that encapsulates the necessary features of each device on which VTK-m can run. At the core of the device adapter is a collection of basic algorithms optimized for the specific device. Many features of VTK-m, such as worklets, are built on top of these device algorithms. Using these higher level structures simplifies programming.

However, it is sometimes desirable to run directly run these algorithms provided by the device adapter. VTK-m comes with the templated class `vtkm::cont::Algorithm` that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. All algorithms also accept an optional device adapter argument.

`Algorithm` contains no state. It only has a set of static methods that implement its algorithms. The following methods are available.

> Did you know?
>
> *Many of the following device adapter algorithms take input and output* `ArrayHandle`*s, and these functions will handle their own memory management. This means that it is unnecessary to allocate output arrays. For example, it is unnecessary to call* `ArrayHandle::Allocate` *for the output array passed to the* `Algorithm::Copy` *method.*

34.1 Copy

The `Algorithm::Copy` method copies data from an input array to an output array. The copy takes place in the execution environment.

Example 34.1: Using the `Copy` algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::Algorithm::Copy(input, output);

// output has { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 }
```

34.2 CopyIf

The Algorithm::CopyIf method selectively removes values from an array. The *copy if* algorithm is also sometimes referred to as *stream compact*. The first argument, the input, is an ArrayHandle to be compacted (by removing elements). The second argument, the stencil, is an ArrayHandle of equal size with flags indicating whether the corresponding input value is to be copied to the output. The third argument is an output ArrayHandle whose length is set to the number of true flags in the stencil and the passed values are put in order to the output array.

Algorithm::CopyIf also accepts an optional fourth argument that is a unary predicate to determine what values in the stencil (second argument) should be considered true. See Section 34.18 for more information on unary predicates. The unary predicate determines the true/false value of the stencil that determines whether a given entry is copied. If no unary predicate is given, then CopyIf will copy all values whose stencil value is not equal to 0 (or the closest equivalent to it). More specifically, it copies values not equal to vtkm::TypeTraits::ZeroInitialization.

Example 34.2: Using the CopyIf algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
std::vector<vtkm::UInt8> stencilBuffer{ 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);
vtkm::cont::ArrayHandle<vtkm::UInt8> stencil =
  vtkm::cont::make_ArrayHandle(stencilBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::Algorithm::CopyIf(input, stencil, output);


// output has { 0, 5, 3, 8, 3 }

struct LessThan5
{
  VTKM_EXEC_CONT bool operator()(vtkm::Int32 x) const { return x < 5; }
};

vtkm::cont::Algorithm::CopyIf(input, input, output, LessThan5());

// output has { 0, 1, 1, 4, 3, 3 }
```

34.3 CopySubRange

The Algorithm::CopySubRange method copies the contents of a section of one ArrayHandle to another. The first argument is the input ArrayHandle. The second argument is the index from which to start copying data. The third argument is the number of values to copy from the input to the output. The fourth argument is the output ArrayHandle, which will be grown if it is not large enough. The fifth argument, which is optional, is the index in the output array to start copying data to. If the output index is not specified, data are copied to the beginning of the output array.

Example 34.3: Using the CopySubRange algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> output;
```

```

    vtkm::cont::Algorithm::CopySubRange(input, 1, 7, output);

    // output has { 0, 1, 1, 5, 5, 4, 3 }
```

34.4 LowerBounds

The Algorithm::LowerBounds method takes three arguments. The first argument is an ArrayHandle of sorted values. The second argument is another ArrayHandle of items to find in the first array. LowerBounds find the index of the first item that is greater than or equal to the target value, much like the std::lower_bound STL algorithm. The results are returned in an ArrayHandle given in the third argument.

There are two specializations of Algorithm::LowerBounds. The first takes an additional comparison function that defines the less-than operation. The second specialization takes only two parameters. The first is an ArrayHandle of sorted vtkm::Id s and the second is an ArrayHandle of vtkm::Id to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

Example 34.4: Using the LowerBounds algorithm.

```
    std::vector<vtkm::Int32> sortedBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 };
    std::vector<vtkm::Int32> valuesBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

    vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
      vtkm::cont::make_ArrayHandle(sortedBuffer);
    vtkm::cont::ArrayHandle<vtkm::Int32> values =
      vtkm::cont::make_ArrayHandle(valuesBuffer);

    vtkm::cont::ArrayHandle<vtkm::Id> output;

    vtkm::cont::Algorithm::LowerBounds(sorted, values, output);

    // output has { 8, 0, 1, 1, 6, 6, 5, 3, 8, 10, 11, 3 }

    std::vector<vtkm::Int32> revSortedBuffer{ 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 };
    vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
      vtkm::cont::make_ArrayHandle(revSortedBuffer);

    vtkm::cont::Algorithm::LowerBounds(
      reverseSorted, values, output, vtkm::SortGreater());

    // output has { 2, 11, 9, 9, 4, 4, 6, 7, 2, 1, 0, 7 }
```

34.5 Reduce

The Algorithm::Reduce method takes an input array, initial value, and a binary function and computes a "total" of applying the binary function to all entries in the array. The provided binary function must be associative (but it need not be commutative). There is a specialization of Reduce that does not take a binary function and computes the sum.

Example 34.5: Using the Reduce algorithm.

```
    std::vector<vtkm::Int32> inputBuffer{ 1, 1, 5, 5 };
    vtkm::cont::ArrayHandle<vtkm::Int32> input =
      vtkm::cont::make_ArrayHandle(inputBuffer);

    vtkm::Int32 sum = vtkm::cont::Algorithm::Reduce(input, 0);
```

```

  // sum is 12

  vtkm::Int32 product = vtkm::cont::Algorithm::Reduce(input, 1, vtkm::Multiply());
  // product is 25
```

34.6 ReduceByKey

The Algorithm::ReduceByKey method works similarly to the Reduce method except that it takes an additional array of keys, which must be the same length as the values being reduced. The arrays are partitioned into segments that have identical adjacent keys, and a separate reduction is performed on each partition. The unique keys and reduced values are returned in separate arrays.

Example 34.6: Using the ReduceByKey algorithm.

```
  std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

  vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

  vtkm::cont::ArrayHandle<vtkm::Id> uniqueKeys;
  vtkm::cont::ArrayHandle<vtkm::Int32> sums;

  vtkm::cont::Algorithm::ReduceByKey(keys, input, uniqueKeys, sums, vtkm::Add());

  // uniqueKeys is { 0, 3, 5, 6 }
  // sums is { 7, 12, 4, 30 }

  vtkm::cont::ArrayHandle<vtkm::Int32> products;

  vtkm::cont::Algorithm::ReduceByKey(
    keys, input, uniqueKeys, products, vtkm::Multiply());

  // products is { 0, 25, 4, 4536 }
```

34.7 ScanExclusive

The Algorithm::ScanExclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is always 0. The second value in the output is the same as the first value in the input. The third value in the output is the sum of the first two values in the input. The fourth value in the output is the sum of the first three values of the input, and so on. ScanExclusive returns the sum of all values in the input. There are two forms of ScanExclusive. The first performs the sum using addition. The second form accepts a custom binary functor to use as the "sum" operator and a custom initial value (instead of 0).

Example 34.7: Using the ScanExclusive algorithm.

```
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

  vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;

  vtkm::cont::Algorithm::ScanExclusive(input, runningSum);

  // runningSum is { 0, 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50 }

  vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;

  vtkm::cont::Algorithm::ScanExclusive(input, runningMax, vtkm::Maximum(), -1);

  // runningMax is { -1, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9 }
```

34.8 ScanExclusiveByKey

The Algorithm::ScanExclusiveByKey method works similarly to the ScanExclusive method except that it takes an additional array of keys, which must be the same length as the values being scanned. The arrays are partitioned into segments that have identical adjacent keys, and a separate scan is performed on each partition. Only the scanned values are returned.

Example 34.8: Using ScanExclusiveByKey algorithm.

```
  std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

  vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

  vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;

  vtkm::cont::Algorithm::ScanExclusiveByKey(keys, input, runningSums);

  // runningSums is { 0, 7, 0, 1, 2, 7, 0, 0, 3, 10, 18, 27 }

  vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;

  vtkm::cont::Algorithm::ScanExclusiveByKey(
    keys, input, runningMaxes, -1, vtkm::Maximum());

  // runningMax is { -1, 7, -1, 1, 1, 5, -1, -1, 3, 7, 8, 9 }
```

34.9 ScanInclusive

The Algorithm::ScanInclusive method takes an input and an output ArrayHandle and performs a running sum on the input array. The first value in the output is the same as the first value in the input. The second value in the output is the sum of the first two values in the input. The third value in the output is the sum of the first three values of the input, and so on. ScanInclusive returns the sum of all values in the input. There are two forms of ScanInclusive: one performs the sum using addition whereas the other accepts a custom binary function to use as the sum operator.

Example 34.9: Using the ScanInclusive algorithm.

```
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

  vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;

  vtkm::cont::Algorithm::ScanInclusive(input, runningSum);

  // runningSum is { 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50, 53 }

  vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;

  vtkm::cont::Algorithm::ScanInclusive(input, runningMax, vtkm::Maximum());

  // runningMax is { 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9 }
```

34.10 ScanInclusiveByKey

The Algorithm::ScanInclusiveByKey method works similarly to the ScanInclusive method except that it takes an additional array of keys, which must be the same length as the values being scanned. The arrays are partitioned into segments that have identical adjacent keys, and a separate scan is performed on each partition. Only the scanned values are returned.

Example 34.10: Using the ScanInclusiveByKey algorithm.

```
  std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

  vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

  vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;

  vtkm::cont::Algorithm::ScanInclusiveByKey(keys, input, runningSums);

  // runningSums is { 7, 7, 1, 2, 7, 12, 4, 3, 10, 18, 27, 30 }

  vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;

  vtkm::cont::Algorithm::ScanInclusiveByKey(
    keys, input, runningMaxes, vtkm::Maximum());

  // runningMax is { 7, 7, 1, 1, 5, 5, 4, 3, 7, 8, 9, 9 }
```

34.11 Schedule

The Algorithm::Schedule method takes a functor as its first argument and invokes it a number of times specified by the second argument. It should be assumed that each invocation of the functor occurs on a separate thread although in practice there could be some thread sharing.

There are two versions of the Schedule method. The first version takes a vtkm::Id and invokes the functor that number of times. The second version takes a vtkm::Id3 and invokes the functor once for every entry in a 3D array of the given dimensions.

The functor is expected to be an object with a const overloaded parentheses operator. The operator takes as a parameter the index of the invocation, which is either a vtkm::Id or a vtkm::Id3 depending on what version of Schedule is being used. The functor must also subclass vtkm::exec::FunctorBase, which provides the error handling facilities for the execution environment. FunctorBase contains a public method named RaiseError that takes a message and will cause a vtkm::cont::ErrorExecution exception to be thrown in the control environment.

34.12 Sort

The Algorithm::Sort method provides an unstable sort of an array. There are two forms of the Sort method. The first takes an ArrayHandle and sorts the values in place. The second takes an additional argument that is a functor that provides the comparison operation for the sort.

Example 34.11: Using the Sort algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> array =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::Algorithm::Sort(array);

// array has { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 }

vtkm::cont::Algorithm::Sort(array, vtkm::SortGreater());

// array has { 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 }
```

34.13 SortByKey

The Algorithm::SortByKey method works similarly to the Sort method except that it takes two ArrayHandles: an array of keys and a corresponding array of values. The sort orders the array of keys in ascending values and also reorders the values so they remain paired with the same key. Like Sort, SortByKey has a version that sorts by the default less-than operator and a version that accepts a custom comparison functor.

Example 34.12: Using the SortByKey algorithm.

```
std::vector<vtkm::Int32> keyBuffer{ 7, 0, 1, 5, 4, 8, 9, 3 };
std::vector<vtkm::Id> valueBuffer{ 0, 1, 2, 3, 4, 5, 6, 7 };

vtkm::cont::ArrayHandle<vtkm::Int32> keys =
  vtkm::cont::make_ArrayHandle(keyBuffer);
vtkm::cont::ArrayHandle<vtkm::Id> values =
  vtkm::cont::make_ArrayHandle(valueBuffer);

vtkm::cont::Algorithm::SortByKey(keys, values);

// keys has   { 0, 1, 3, 4, 5, 7, 8, 9 }
// values has { 1, 2, 7, 4, 3, 0, 5, 6 }

vtkm::cont::Algorithm::SortByKey(keys, values, vtkm::SortGreater());

// keys has   { 9, 8, 7, 5, 4, 3, 1, 0 }
// values has { 6, 5, 0, 3, 4, 7, 2, 1 }
```

34.14 Synchronize

The Synchronize method waits for any asynchronous operations running on the device to complete and then returns.

34.15 Unique

The Algorithm::Unique method removes all duplicate values in an ArrayHandle. The method will only find duplicates if they are adjacent to each other in the array. The easiest way to ensure that duplicate values are adjacent is to sort the array first.

There are two versions of Unique. The first uses the equals operator to compare entries. The second accepts a binary functor to perform the comparisons.

Example 34.13: Using the Unique algorithm.

```
std::vector<vtkm::Int32> valuesBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 7, 9 };
vtkm::cont::ArrayHandle<vtkm::Int32> values =
  vtkm::cont::make_ArrayHandle(valuesBuffer);

vtkm::cont::Algorithm::Unique(values);

// values has {0, 1, 3, 4, 5, 7, 9}

std::vector<vtkm::Float64> fvaluesBuffer{ 0.0, 0.001, 0.0, 1.5, 1.499, 2.0 };
vtkm::cont::ArrayHandle<vtkm::Float64> fvalues =
  vtkm::cont::make_ArrayHandle(fvaluesBuffer);

struct AlmostEqualFunctor
{
  VTKM_EXEC_CONT bool operator()(vtkm::Float64 x, vtkm::Float64 y) const
  {
    return (vtkm::Abs(x - y) < 0.1);
  }
};

vtkm::cont::Algorithm::Unique(fvalues, AlmostEqualFunctor());

// values has {0.0, 1.5, 2.0}
```

34.16 UpperBounds

The Algorithm::UpperBounds method takes three arguments. The first argument is an ArrayHandle of sorted values. The second argument is another ArrayHandle of items to find in the first array. UpperBounds find the index of the first item that is greater than to the target value, much like the std::upper_bound STL algorithm. The results are returned in an ArrayHandle given in the third argument.

There are two specializations of UpperBounds. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an ArrayHandle of sorted vtkm::Id s and the second is an ArrayHandle of vtkm::Id s to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

Example 34.14: Using the UpperBounds algorithm.

```
std::vector<vtkm::Int32> sortedBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 };
std::vector<vtkm::Int32> valuesBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
  vtkm::cont::make_ArrayHandle(sortedBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> values =
  vtkm::cont::make_ArrayHandle(valuesBuffer);

vtkm::cont::ArrayHandle<vtkm::Id> output;

```

```
vtkm::cont::Algorithm::UpperBounds(sorted, values, output);

// output has { 10, 1, 3, 3, 8, 8, 6, 5, 10, 11, 12, 5 }

std::vector<vtkm::Int32> revSortedBuffer{ 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 };
vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
  vtkm::cont::make_ArrayHandle(revSortedBuffer);

vtkm::cont::Algorithm::UpperBounds(
  reverseSorted, values, output, vtkm::SortGreater());

// output has { 4, 12, 11, 11, 6, 6, 7, 9, 4, 2, 1, 9 }
```

34.17 Specifying the Device Adapter

When you call a method in `vtkm::cont::Algorithm`, a device is automatically specified based on available hardware and the VTK-m state. However, if you want to use a specific device, you can specify that device by passing either a `vtkm::cont::DeviceAdapterId` or a device adapter tag as the first argument to any of these methods.

Example 34.15: Using the `DeviceAdapter` with `vtkm::cont::Algorithm`.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> output_no_device_specified;

vtkm::cont::ArrayHandle<vtkm::Int32> output_device_specified;

vtkm::cont::Algorithm::Copy(input, output_no_device_specified);

//optional we can pass the device or int id number
vtkm::cont::Algorithm::Copy(
  vtkm::cont::DeviceAdapterTagSerial(), input, output_device_specified);

// output has { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 }
```

34.18 Predicates and Operators

VTK-m follows certain design philosophies consistent with the functional programming paradigm. This assists in making implementations device agnostic and ensuring that various functions operate correctly and efficiently in multiple environments. Many basic operations, such as binary and unary comparisons and predicates, are implemented as templated functors. These are mostly re-implementations of basic C++ STL functors that can be used in the VTK-m execution environment.

Strictly using a functor by itself adds little in the way of functionality to the code. Their use is demonstrated more when used as parameters to one of the `vtkm::cont::Algorithm` methods discussed earlier in this chapter. Currenly, VTK-m provides 3 categories of functors: `Unary Predicates`, `Binary Predicates`, and `Binary Operators`.

34.18.1 Unary Predicates

`Unary Predicates` are functors that take a single parameter and return a Boolean value. These types of functors are useful in determining if values have been initialized or zeroed out correctly.

`vtkm::IsZeroInitialized` Returns True if argument is the identity of its type.

`vtkm::NotZeroInitialized` Returns True if the argument is not the identify of its type.

`vtkm::LogicalNot` Returns True iff the argument is False. Requires that the argument type is convertible to a Boolean or implements the `!` operator.

Example 34.16: Basic Unary Predicate.

```
vtkm::IsZeroInitialized zero_initialized;
vtkm::NotZeroInitialized not_zero_initialized;
vtkm::LogicalNot logical_not;

bool zeroed = zero_initialized(vtkm::TypeTraits<vtkm::Id>::ZeroInitialization());
bool notZeroed = not_zero_initialized(vtkm::Id(1));
bool logicalNot = logical_not(false);
```

34.18.2 Binary Predicates

`Binary Predicates` take two parameters and return a single Boolean value. These types of functors are used when comparing two different parameters for some sort of condition.

`vtkm::Equal` Returns True iff the first argument is equal to the second argument. Requires that the argument type implements the `==` operator.

`vtkm::NotEqual` Returns True iff the first argument is not equal to the second argument. Requires that the argument type implements the `!=` operator.

`vtkm::SortLess` Returns True iff the first argument is less than the second argument. Requires that the argument type implements the `<` operator.

`vtkm::SortGreater` Returns True iff the first argument is greater than the second argument. Requires that the argument type implements the `<` operator (the comparison is inverted internally).

`vtkm::LogicalAnd` Returns True iff the first argument and the second argument are True. Requires that the argument type is convertible to a Boolean or implements the `&&` operator.

`vtkm::LogicalOr` Returns True iff the first argument or the second argument is True. Requires that the argument type is convertible to a Boolean or implements the `||` operator.

Example 34.17: Basic Binary Predicate.

```
vtkm::Equal equal_;
vtkm::NotEqual not_equal;
vtkm::SortLess sort_less;
vtkm::SortGreater sort_greater;
vtkm::LogicalAnd logical_and;
vtkm::LogicalOr logical_or;

bool equal = equal_(vtkm::Id(1), vtkm::Id(1));
bool notEqual = not_equal(vtkm::Id(1), vtkm::Id(2));
```

```
  bool sortLess = sort_less(vtkm::Id(1), vtkm::Id(2));
  bool sortGreater = sort_greater(vtkm::Id(2), vtkm::Id(1));
  bool logicalAnd = logical_and(true, true);
  bool logicalOr = logical_or(true, false);
```

34.18.3 Binary Operators

`Binary Operators` take two parameters and return a single value (usually of the same type as the input arguments). These types of functors are useful when performing reductions or transformations of a dataset.

`vtkm::Sum` Returns the sum of two arguments. Requires that the argument type implements the + operator.

`vtkm::Product` Returns the product (multiplication) of two arguments. Requires that the argument type implements the * operator.

`vtkm::Maximum` Returns the larger of two arguments. Requires that the argument type implements the < operator.

`vtkm::Minimum` Returns the smaller of two arguments. Requires that the argument type implements the < operator.

`vtkm::MinAndMax` Returns a `vtkm::Vec <T,2>` that represents the minimum and maximum values. Requires that the argument type implements the `vtkm::Min` and `vtkm::Max` functions.

`vtkm::BitwiseAnd` Returns the bitwise and of two arguments. Requires that the argument type implements the & operator.

`vtkm::BitwiseOr` Returns the bitwise or of two arguments. Requires that the argument type implements the | operator.

`vtkm::BitwiseXor` Returns the bitwise xor of two arguments. Requires that the argument type implements the ˆ operator.

Example 34.18: Basic Binary Operator.

```
  vtkm::Sum sum_;
  vtkm::Product product_;
  vtkm::Maximum maximum_;
  vtkm::Minimum minimum_;
  vtkm::MinAndMax<vtkm::Id> min_and_max;
  vtkm::BitwiseAnd bitwise_and;
  vtkm::BitwiseOr bitwise_or;
  vtkm::BitwiseXor bitwise_xor;

  vtkm::Id sum = sum_(vtkm::Id(1), vtkm::Id(1));
  vtkm::Id product = product_(vtkm::Id(2), vtkm::Id(2));
  vtkm::Id max = maximum_(vtkm::Id(1), vtkm::Id(2));
  vtkm::Id min = minimum_(vtkm::Id(1), vtkm::Id(2));
  vtkm::Id2 minAndMax = min_and_max(vtkm::Id(3), vtkm::Id(4));
  vtkm::Id bitwiseAnd = bitwise_and(vtkm::Id(1), vtkm::Id(3));
  vtkm::Id bitwiseOr = bitwise_or(vtkm::Id(1), vtkm::Id(2));
  vtkm::Id bitwiseXor = bitwise_xor(vtkm::Id(7), vtkm::Id(4));
```

34.18.4 Creating Custom Comparators

In addition to using the built in operators and predicates, it is possible to create your own custom functors to be used in one of the vtkm::cont::Algorithm. Custom operator and predicate functors can be used to apply specific logic used to manipulate your data. The following example creates a unary predicate that checks if the input is a power of 2.

Example 34.19: Custom Unary Predicate Implementation.

```
struct PowerOfTwo
{
  VTKM_EXEC_CONT bool operator()(const vtkm::Id& x) const
  {
    if (x <= 0)
    {
      return false;
    }
    vtkm::BitwiseAnd bitwise_and;
    return bitwise_and(x, vtkm::Id(x - 1)) == 0;
  }
};
```

Example 34.20: Custom Unary Predicate Usage.

```
  PowerOfTwo power_of_two;

  bool powerOfTwo = power_of_two(vtkm::Id(4)); // returns true
  powerOfTwo = power_of_two(vtkm::Id(5)); // returns false
```

CHAPTER

THIRTYFIVE

VIRTUAL OBJECTS

VTK-m contains a base class called vtkm::VirtualObjectBase. Any class built in VTK-m that has virtual methods and is intended to work in both the control and execution environment should inherit from vtkm::VirtualObjectBase. Hierarchies under vtkm::VirtualObjectBase can be used in conjuction with vtkm::cont::VirtualObjectHandle to transfer from the control environment (where they are set up) to the execution environment (where they are used). For a refresher on the differences between the control and execution environments, please look back to Chapter 15.

In addition to inheriting from vtkm::VirtualObjectBase, virtual objects have to satisfy 2 other conditions to work correctly. First, they have to be a plain old data type that can be copied with memcpy (with the exception of the virtual table, which vtkm::cont::VirtualObjectHandle will take care of). Second, if the object changes its state in the control environment, it should call Modified on itself so the vtkm::cont::VirtualObjectHandle will know it update the object in the execution environment. The following example shows a simple implementation of a vtkm::VirtualObjectBase that describes 2D shapes and determines whether a provided point is within a given shape.

Example 35.1: Using VirtualObjectBase.

```
class VTKM_ALWAYS_EXPORT ShapeType : public vtkm::VirtualObjectBase
{
public:
  using Point = vtkm::Vec<vtkm::FloatDefault, 2>;

  VTKM_EXEC_CONT virtual bool Inside(const Point& point) const = 0;
  VTKM_EXEC_CONT bool Inside(vtkm::FloatDefault x, vtkm::FloatDefault y) const
  {
    return this->Inside(vtkm::Vec<vtkm::FloatDefault, 2>(x, y));
  }
};

class VTKM_ALWAYS_EXPORT Square : public ShapeType
{
public:
  VTKM_EXEC_CONT Square()
    : LowerLeft(Point(vtkm::FloatDefault(0)))
    , UpperRight(Point(vtkm::FloatDefault(1)))
  {
  }

  VTKM_EXEC_CONT Square(const Point& lowerLeft, const Point& upperRight)
    : LowerLeft(lowerLeft)
    , UpperRight(upperRight)
  {
  }

  VTKM_CONT void SetLowerLeft(const Point& point)
  {
    this->LowerLeft = point;
    this->Modified();
  }

  VTKM_CONT void SetUpperRight(const Point& point)
  {
    this->UpperRight = point;
    this->Modified();
  }

  VTKM_EXEC_CONT const Point& GetLowerLeft() const { return this->LowerLeft; }
  VTKM_EXEC_CONT const Point& GetUpperRight() const { return this->UpperRight; }

  VTKM_EXEC_CONT bool Inside(const Point& point) const final
  {
    if (point[0] > LowerLeft[0] && point[0] < UpperRight[0] &&
        point[1] > LowerLeft[1] && point[1] < UpperRight[1])
    {
      return true;
    }
    return false;
  }

private:
  Point LowerLeft;
  Point UpperRight;
};
```

Common Errors

Certain devices will still correctly process a vtkm::VirtaulObjectBase *derived object without calling* Modified *in the control environment to indicate that the data has changed. This is because certain devices utilize the same memory space for the control/execution environments and updates to the object in one space are refelected in the other. However, this will not work for all device types. To implement code that will work for all device types it is important to call* Modified *whenever the control environment object's state is changed.*

Virtual Objects implement the virtual methods, but a given vtkm::cont::VirtualObjectHandle is what transfers those virtual method implementations into the execution environment. Each vtkm::cont::VirtualObjectHandle takes a template parameter, VirtualBaseType, which serves as the base class that acts as the interface. This base class must inherit from vtkm::VirtualObjectBase.

A derived object can be bound to the handle either during construction or using the Reset function on previously constructed handles. These functions accepts a control side pointer to the derived class object, a boolean flag stating if the handle should acquire ownership of the object (i.e. manage the lifetime), and a type-list of device adapter tags where the object is expected to be used.

Once the handle is constructed, you can get an execution side pointer by calling the PrepareForExecution function. The device adapter passed to this function should be one of the devices passed in during the set up of the handle. The following example shows how to implement a vtkm::cont::VirtualObjectHandle that utilizes the ShapeType vtkm::VirtualObjectBase extension implemented above.

Example 35.2: Using VirtualObjectBase.

```
class VTKM_ALWAYS_EXPORT ShapeHandle
```

```
  : public vtkm::cont::VirtualObjectHandle<ShapeType>
{
private:
  using Superclass = vtkm::cont::VirtualObjectHandle<ShapeType>;

public:
  ShapeHandle() = default;

  template<typename Shape,
           typename DeviceAdapterList = VTKM_DEFAULT_DEVICE_ADAPTER_LIST_TAG>
  explicit ShapeHandle(Shape* function,
                       bool acquireOwnership = true,
                       DeviceAdapterList devices = DeviceAdapterList())
    : Superclass(function, acquireOwnership, devices)
  {
  }
};
```

Did you know?

Virtual methods are implemented in c++ using what is known as a virtual table. When a class is compiled, each class that implements virtual methods is given a virtual table pointer. This pointer references a structure that describes the physical address of the function that should be called for a corresponding virtual function call. It becomes difficult to maintain this virtual to physical linkage when transfering data to different devices since the physcial addresses will differ on the other device. Thankfully, you don't have to worry about these details. VTK-m implements custom transfer functions for each device in order to support seamlessly moving virtual tables from the control environment to the execution environment. You can read Chapter 38 for more information.

We can put the above concepts into practice by implementing a worklet that will utilize our `VirtualObject` implementations in an execution environment.

Example 35.3: Using `VirtualObjects`.

```
class EvaluateShape : public vtkm::worklet::WorkletMapField
{
public:
  using ControlSignature = void(FieldIn, FieldOut);
  using ExecutionSignature = void(_1, _2);

  EvaluateShape(const ShapeType* shape)
    : Shape(shape)
  {
  }

  VTKM_EXEC void operator()(vtkm::Vec<vtkm::FloatDefault, 2>& point,
                            bool& inside) const
  {
    inside = this->Shape->Inside(point);
  }

private:
  const ShapeType* Shape;
};


VTKM_CONT void GetPointsInSquare(
  vtkm::cont::ArrayHandle<bool>& inside,
  vtkm::cont::ArrayHandleVirtual<vtkm::Vec<vtkm::FloatDefault, 2>> points)
{
  Square square;
  ShapeHandle squareHandle(&square, false);

  vtkm::cont::Invoker invoker;
  EvaluateShape eval(
    squareHandle.PrepareForExecution(vtkm::cont::DeviceAdapterTagSerial()));
  invoker(eval, points, inside);
}
```

CHAPTER

THIRTYSIX

CUSTOM ARRAY STORAGE

Chapters 16, 26, and 27 introduce the `vtkm::cont::ArrayHandle` class. In them, we learned how an `ArrayHandle` manages the memory allocation of an array, provides access to the data via array portals, and supervises the movement of data between the control and execution environments.

In addition to these data management features, `ArrayHandle` also provides a configurable *storage* mechanism that allows you, through efficient template configuration, to redefine how data are stored and retrieved. The storage object provides an encapsulated interface around the data so that any necessary strides, offsets, or other access patterns may be handled internally. The relationship between array handles and their storage object is shown in Figure 36.1.

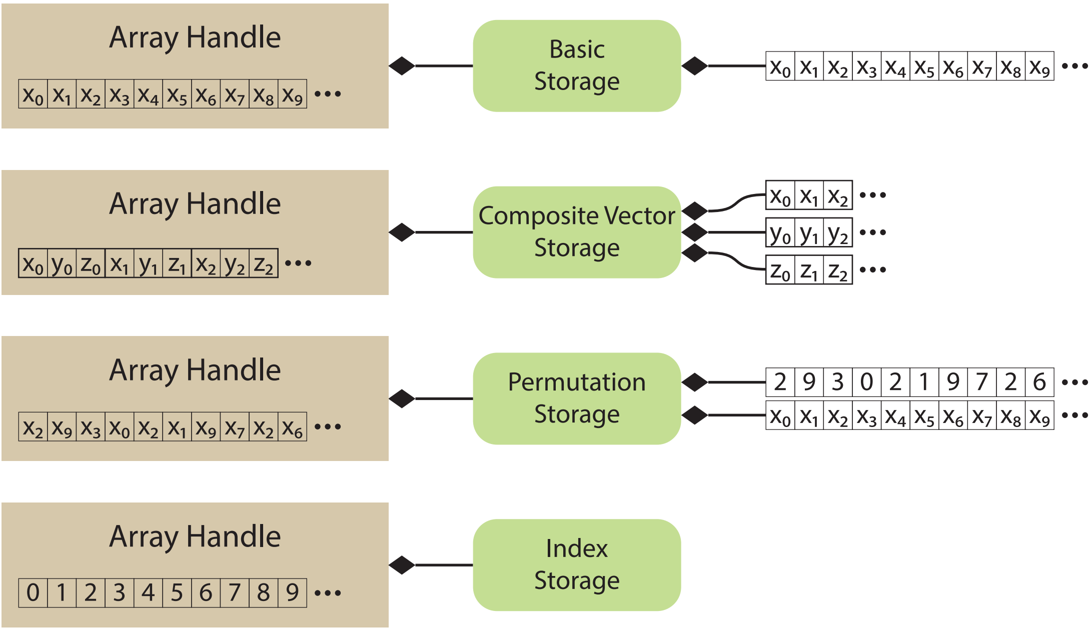

Array Handle
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 •••
Basic Storage
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 •••
Array Handle
x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2 •••
Composite Vector Storage
x_0 x_1 x_2 •••
y_0 y_1 y_2 •••
z_0 z_1 z_2 •••
Array Handle
x_2 x_9 x_3 x_0 x_2 x_1 x_9 x_7 x_2 x_6 •••
Permutation Storage
2 9 3 0 2 1 9 7 2 6 •••
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 •••
Array Handle
0 1 2 3 4 5 6 7 8 9 •••
Index Storage

Figure 36.1: Array handles, storage objects, and the underlying data source.

One interesting consequence of using a generic storage object to manage data within an array handle is that the storage can be defined functionally rather than point to data stored in physical memory. Thus, implicit array handles are easily created by adapting to functional "storage." For example, the point coordinates of a uniform rectilinear grid are implicit based on the topological position of the point. Thus, the point coordinates for uniform rectilinear grids can be implemented as an implicit array with the same interface as explicit arrays (where unstructured grid points would be stored). In this chapter we explore the many ways you can manipulate the ArrayHandle storage.

36.1 Basic Storage

As previously discussed in Chapter 16, vtkm::cont::ArrayHandle takes two template arguments.

Example 36.1: Declaration of the vtkm::cont::ArrayHandle templated class (again).

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
class ArrayHandle;
```

The first argument is the only one required and has been demonstrated multiple times before. The second (optional) argument specifies something called a storage, which provides the interface between the generic vtkm::cont::ArrayHandle class and a specific storage mechanism in the control environment. If the storage parameter is not explicitly defined, it is set to VTKM_DEFAULT_STORAGE_TAG, which is a macro that resolves to vtkm::cont::StorageTagBasic.

The default storage can always be overridden by specifying an array storage tag. Here is an example of specifying the storage type when declaring an array handle.

Example 36.2: Specifying the storage type for an ArrayHandle.

```
  vtkm::cont::ArrayHandle<vtkm::Float32, vtkm::cont::StorageTagBasic> arrayHandle;
```

Although setting an ArrayHandle's storage explicitly to StorageBasic as in Example 36.2 is seldom useful (since this is the default value), setting the storage is a good way to propagate the storage mechanism through template parameters. The remainder of this chapter uses the storage mechanism to customize the representation of arrays.

36.2 Implementing Fancy Arrays

Although the behavior of fancy arrays might seem complicated, they are actually straightforward to implement. VTK-m provides several mechanisms to implement fancy arrays.

36.2.1 Implicit Array Handles

The generic array handle and storage templating in VTK-m allows for any type of operations to retrieve a particular value. Typically this is used to convert an index to some location or locations in memory. However, it is also possible to compute a value directly from an index rather than look up some value in memory. Such an array is completely functional and requires no storage in memory at all. Such a functional array is called an *implicit array handle*. Implicit arrays are an example of *fancy array handles*, which are array handles that behave like regular arrays but do special processing under the covers to provide values.

Specifying a functional or implicit array in VTK-m is straightforward. VTK-m has a special class named vtkm::cont::ArrayHandleImplicit that makes an implicit array containing values generated by a user-specified *functor*. A functor is simply a C++ class or struct that contains an overloaded parenthesis operator so that it can be used syntactically like a function.

To demonstrate the use of ArrayHandleImplicit, let us say we want an array of even numbers. The array has the values $[0,2,4,6,\ldots]$ (double the index) up to some given size. Although we could easily create this array in memory, we can save space and possibly time by computing these values on demand.

Did you know?

VTK-m already comes with an implicit array handle named vtkm::cont::ArrayHandleCounting *that can make implicit even numbers as well as other more general counts. (See Section 26.3 for details.) So in practice you would not have to create a special implicit array, but we are doing so here for demonstrative purposes.*

The first step to using ArrayHandleImplicit is to declare a functor. The functor's parenthesis operator should accept a single argument of type vtkm::Id and return a value appropriate for that index. The parenthesis operator should also be declared const because it is not allowed to change the class' state.

Example 36.3: Functor that doubles an index.

```
struct DoubleIndexFunctor
{
  VTKM_EXEC_CONT
  vtkm::Id operator()(vtkm::Id index) const { return 2 * index; }
};
```

Once the functor is defined, an implicit array can be declared using the templated vtkm::cont::ArrayHandleImplicit class. The single template argument is the functor's type.

Example 36.4: Declaring a ArrayHandleImplicit.

```
  vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor> implicitArray(
    DoubleIndexFunctor(), 50);
```

For convenience, vtkm/cont/ArrayHandleImplicit.h also declares the vtkm::cont::make_ArrayHandleImplicit function. This function takes a functor and the size of the array and returns the implicit array.

Example 36.5: Using make_ArrayHandleImplicit.

```
  vtkm::cont::make_ArrayHandleImplicit(DoubleIndexFunctor(), 50);
```

If the implicit array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of vtkm::cont::ArrayHandleImplicit for your array.

Example 36.6: Custom implicit array handle for even numbers.

```
#include <vtkm/cont/ArrayHandleImplicit.h>

class ArrayHandleDoubleIndex
  : public vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
    ArrayHandleDoubleIndex,
    (vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor>));

  VTKM_CONT
  ArrayHandleDoubleIndex(vtkm::Id numberOfValues)
    : Superclass(DoubleIndexFunctor(), numberOfValues)
  {
  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the types Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 36.6 is not templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. This macro takes two parameters. The first parameter is the name of the subclass where the macro is defined and the second parameter is the immediate superclass including the full template specification. The second parameter of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification. (The other macro is described in Section 36.2.2 on page 293).

36.2.2 Transformed Arrays

Another type of fancy array handle is the transformed array. A transformed array takes another array and applies a function to all of the elements to produce a new array. A transformed array behaves much like a map operation except that a map operation writes its values to a new memory location whereas the transformed array handle produces its values on demand so that no additional storage is required.

Specifying a transformed array in VTK-m is straightforward. VTK-m has a special class named vtkm::cont::ArrayHandleTransform that takes an array handle and a functor and provides an interface to a new array comprising values of the first array applied to the functor.

To demonstrate the use of ArrayHandleTransform, let us say that we want to scale and bias all of the values in a target array. That is, each value in the target array is going to be multiplied by a given scale and then offset by adding a bias value. (The scale and bias are uniform across all entries.) We could, of course, easily create a worklet to apply this scale and bias to each entry in the target array and save the result in a new array, but we can save space and possibly time by computing these values on demand.

The first step to using ArrayHandleTransform is to declare a functor. The functor's parenthesis operator should accept a single argument of the type of the target array and return the transformed value. For more generally applicable transform functors, it is often useful to make the parenthesis operator a template. The parenthesis operator should also be declared const because it is not allowed to change the class' state.

Example 36.7: Functor to scale and bias a value.

```
template<typename T>
struct ScaleBiasFunctor
{
  VTKM_EXEC_CONT
  ScaleBiasFunctor(T scale = T(1), T bias = T(0))
    : Scale(scale)
    , Bias(bias)
  {
  }

  VTKM_EXEC_CONT
  T operator()(T x) const { return this->Scale * x + this->Bias; }

  T Scale;
  T Bias;
};
```

Once the functor is defined, a transformed array can be declared using the templated vtkm::cont::ArrayHandleTransform class. The first template argument is the type of array being transformed. The second template argument is the type of functor used for the transformation. The third template argument, which is optional, is the type for an inverse functor that provides the inverse operation of the functor in the second argument. This inverse functor is used for writing values into the array. For arrays that will only be read from, there is no need to supply this inverse functor.

That said, it is generally easier to use the vtkm::cont::make_ArrayHandleTransform convenience function. This function takes an array and a functor (and optionally an inverse functor) and returns a transformed array.

Example 36.8: Using make_ArrayHandleTransform.

```
    vtkm::cont::make_ArrayHandleTransform(array,
                                          ScaleBiasFunctor<vtkm::Float32>(2, 3))
```

If the transformed array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of vtkm::cont::ArrayHandleTransform or convenience make_ArrayHandle* function for your array.

Example 36.9: Custom transform array handle for scale and bias.

```
#include <vtkm/cont/ArrayHandleTransform.h>

template<typename ArrayHandleType>
class ArrayHandleScaleBias
  : public vtkm::cont::ArrayHandleTransform<
      ArrayHandleType,
      ScaleBiasFunctor<typename ArrayHandleType::ValueType>>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleScaleBias,
    (ArrayHandleScaleBias<ArrayHandleType>),
    (vtkm::cont::ArrayHandleTransform<
      ArrayHandleType,
      ScaleBiasFunctor<typename ArrayHandleType::ValueType>>));

  VTKM_CONT
  ArrayHandleScaleBias(const ArrayHandleType& array, ValueType scale, ValueType bias)
    : Superclass(array, ScaleBiasFunctor<ValueType>(scale, bias))
  {
  }
};

template<typename ArrayHandleType>
VTKM_CONT ArrayHandleScaleBias<ArrayHandleType> make_ArrayHandleScaleBias(
  const ArrayHandleType& array,
  typename ArrayHandleType::ValueType scale,
  typename ArrayHandleType::ValueType bias)
{
  return ArrayHandleScaleBias<ArrayHandleType>(array, scale, bias);
}
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the types Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 36.9 is templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS macro. This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification. (The other macro is described in Section 36.2.1 on page 292).

36.2.3 Derived Storage

A *derived storage* is a type of fancy array that takes one or more other arrays and changes their behavior in some way. A transformed array (Section 36.2.2) is a specific type of derived array with a simple mapping. In this section we will demonstrate the steps required to create a more general derived storage. When applicable, it is much easier to create a derived array as a transformed array or using the other fancy arrays than to create your own derived storage. However, if these pre-existing fancy arrays do not work work, for example if your derivation uses multiple arrays or requires general lookups, you can do so by creating your own derived storage. For the purposes of the example in this section, let us say we want 2 array handles to behave as one array with the contents concatenated together. We could of course actually copy the data, but we can also do it in place.

The first step to creating a derived storage is to build an array portal that will take portals from arrays being derived. The portal must work in both the control and execution environment (or have a separate version for control and execution).

Example 36.10: Derived array portal for concatenated arrays.

```cpp
#include <vtkm/cont/Algorithm.h>
#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/ArrayPortal.h>
template<typename P1, typename P2>
class ArrayPortalConcatenate
{
public:
  using PortalType1 = P1;
  using PortalType2 = P2;
  using ValueType = typename PortalType1::ValueType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  ArrayPortalConcatenate()
    : Portal1()
    , Portal2()
  {
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  ArrayPortalConcatenate(const PortalType1& portal1, const PortalType2 portal2)
    : Portal1(portal1)
    , Portal2(portal2)
  {
  }

  /// Copy constructor for any other ArrayPortalConcatenate with a portal type
  /// that can be copied to this portal type. This allows us to do any type
  /// casting that the portals do (like the non-const to const cast).
  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OtherP1, typename OtherP2>
  VTKM_EXEC_CONT ArrayPortalConcatenate(
    const ArrayPortalConcatenate<OtherP1, OtherP2>& src)
    : Portal1(src.GetPortal1())
    , Portal2(src.GetPortal2())
  {
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  ValueType Get(vtkm::Id index) const
  {
    if (index < this->Portal1.GetNumberOfValues())
    {
      return this->Portal1.Get(index);
    }
    else
    {
      return this->Portal2.Get(index - this->Portal1.GetNumberOfValues());
    }
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  void Set(vtkm::Id index, const ValueType& value) const
  {
    if (index < this->Portal1.GetNumberOfValues())
    {
      this->Portal1.Set(index, value);
    }
    else
    {
      this->Portal2.Set(index - this->Portal1.GetNumberOfValues(), value);
    }
  }

  VTKM_EXEC_CONT
  const PortalType1& GetPortal1() const { return this->Portal1; }
  VTKM_EXEC_CONT
  const PortalType2& GetPortal2() const { return this->Portal2; }

private:
  PortalType1 Portal1;
  PortalType2 Portal2;
};
```

Like in an adapter storage, the next step in creating a derived storage is to define a tag for the adapter. We shall call ours `StorageTagConcatenate` and it will be templated on the two array handle types that we are deriving. Then, we need to create a specialization of the templated `vtkm::cont::internal::Storage` class. The implementation for a `Storage` for a derived storage is usually trivial compared to an adapter storage because the majority of the work is deferred to the derived arrays.

Example 36.11: `Storage` for derived container of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
struct StorageTagConcatenate
{
};

namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename ArrayHandleType1, typename ArrayHandleType2>
class Storage<typename ArrayHandleType1::ValueType,
              StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>
{
public:
  using ValueType = typename ArrayHandleType1::ValueType;

  using PortalType =
    ArrayPortalConcatenate<typename ArrayHandleType1::PortalControl,
                           typename ArrayHandleType2::PortalControl>;
  using PortalConstType =
    ArrayPortalConcatenate<typename ArrayHandleType1::PortalConstControl,
                           typename ArrayHandleType2::PortalConstControl>;

  VTKM_CONT
  Storage()
    : Valid(false)
  {
  }

  VTKM_CONT
  Storage(const ArrayHandleType1 array1, const ArrayHandleType2 array2)
    : Array1(array1)
    , Array2(array2)
    , Valid(true)
  {
  }

  VTKM_CONT
  PortalType GetPortal()
  {
    VTKM_ASSERT(this->Valid);
    return PortalType(this->Array1.GetPortalControl(),
                      this->Array2.GetPortalControl());
  }

  VTKM_CONT
  PortalConstType GetPortalConst() const
  {
    VTKM_ASSERT(this->Valid);
    return PortalConstType(this->Array1.GetPortalConstControl(),
                           this->Array2.GetPortalConstControl());
  }

  VTKM_CONT
  vtkm::Id GetNumberOfValues() const
  {
    VTKM_ASSERT(this->Valid);
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  VTKM_CONT
  void Allocate(vtkm::Id numberOfValues)
  {
    VTKM_ASSERT(this->Valid);
    // This implementation of allocate, which allocates the same amount in both
    // arrays, is arbitrary. It could, for example, leave the size of Array1
    // alone and change the size of Array2. Or, probably most likely, it could
    // simply throw an error and state that this operation is invalid.
    vtkm::Id half = numberOfValues / 2;
    this->Array1.Allocate(numberOfValues - half);
    this->Array2.Allocate(half);
  }

  VTKM_CONT
  void Shrink(vtkm::Id numberOfValues)
  {
    VTKM_ASSERT(this->Valid);
    if (numberOfValues < this->Array1.GetNumberOfValues())
    {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
    }
    else
    {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
    }
  }

  VTKM_CONT
  void ReleaseResources()
  {
    VTKM_ASSERT(this->Valid);
    this->Array1.ReleaseResources();
    this->Array2.ReleaseResources();
  }

  // Requried for later use in ArrayTransfer class.
  VTKM_CONT
  const ArrayHandleType1& GetArray1() const
  {
    VTKM_ASSERT(this->Valid);
    return this->Array1;
  }
  VTKM_CONT
  const ArrayHandleType2& GetArray2() const
  {
    VTKM_ASSERT(this->Valid);
    return this->Array2;
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
  bool Valid;
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

One of the responsibilities of an array handle is to copy data between the control and execution environments. The default behavior is to request the device adapter to copy data items from one environment to another. This might involve transferring data between a host and device. For an array of data resting in memory, this is necessary. However, implicit storage (described in the previous section) overrides this behavior to pass nothing but the functional array portal. Likewise, it is undesirable to do a raw transfer of data with derived storage. The underlying arrays being derived may be used in other contexts, and it would be good to share the data wherever possible. It is also sometimes more efficient to copy data independently from the arrays being derived than from the derived storage itself.

The mechanism that controls how a particular storage gets transferred to and from the execution environment is encapsulated in the templated vtkm::cont::internal::ArrayTransfer class. By creating a specialization of vtkm::cont::internal::ArrayTransfer, we can modify the transfer behavior to instead transfer the arrays being derived and use the respective copies in the control and execution environments.

vtkm::cont::internal::ArrayTransfer has three template arguments: the base type of the array, the storage tag, and the device adapter tag.

Example 36.12: Prototype for vtkm::cont::internal::ArrayTransfer.

```
namespace vtkm
```

```
{
namespace cont
{
namespace internal
{

template<typename T, typename StorageTag, typename DeviceAdapterTag>
class ArrayTransfer;
}
} // namespace cont
} // namespace vtkm
```

All vtkm::cont::internal::ArrayTransfer implementations must have a constructor method that accepts a pointer to a vtkm::cont::internal::Storage object templated to the same base type and storage tag as the ArrayTransfer object. Assuming that an ArrayHandle is templated using the parameters in Example 36.12, the prototype for the constructor must be equivalent to the following.

Example 36.13: Prototype for ArrayTransfer constructor.

```
ArrayTransfer(vtkm::cont::internal::Storage<T, StorageTag> *storage);
```

Typically the constructor either saves the Storage pointer or other relevant objects from the Storage for later use in the methods.

In addition to this non-default constructor, the vtkm::cont::internal::ArrayTransfer specialization must define the following items.

`ValueType` The type for each item in the array. This is the same type as the first template argument.

`PortalControl` The type of an array portal that is used to access the underlying data in the control environment.

`PortalConstControl` A read-only (const) version of `PortalControl`.

`PortalExecution` The type of an array portal that is used to access the underlying data in the execution environment.

`PortalConstExecution` A read-only (const) version of `PortalExecution`.

`GetNumberOfValues` A method that returns the number of values currently allocated in the execution environment. The results may be undefined if none of the load or allocate methods have yet been called.

`PrepareForInput` A method responsible for transferring data from the control to the execution for input. `PrepareForInput` has one Boolean argument that controls whether this transfer should actually take place. When true, data from the Storage object given in the constructor should be transferred to the execution environment; otherwise the data should not be copied. An ArrayTransfer for a derived array typically ignores this parameter since the arrays being derived manages this transfer already. Regardless of the Boolean flag, a `PortalConstExecution` is returned.

`PrepareForInPlace` A method that behaves just like `PrepareForInput` except that the data in the execution environment is used for both reading and writing so the method returns a PortalExecution. If the array is considered read-only, which is common for derived arrays, then this method should throw a vtkm::cont::ErrorControlBadValue.

`PrepareForOutput` A method that takes a size (in a vtkm::Id) and allocates an array in the execution environment of the specified size. The initial memory can be uninitialized. The method returns a `PortalExecution` for the allocated data. If the array is considered read-only, which is common for derived arrays, then this method should throw a vtkm::cont::ErrorControlBadValue.

`RetrieveOutputData` This method takes an array storage pointer (which is the same as that passed to the constructor, but provided for convenience), allocates memory in the control environment, and copies data from the execution environment into it. If the derived array is considered read-only and both `PrepareForInPlace` and `PrepareForOutput` throw exceptions, then this method should never be called. If it is, then that is probably a bug in `ArrayHandle`, and it is OK to throw `vtkm::cont::ErrorControlInternal`.

`Shrink` A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked. If the derived array is considered read-only, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`ReleaseResources` A method that frees any resources (typically memory) in the execution environment.

Continuing our example derived storage that concatenates two arrays started in Examples 36.10 and 36.11, the following provides an `ArrayTransfer` appropriate for the derived storage.

Example 36.14: `ArrayTransfer` for derived storage of concatenated arrays.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename ArrayHandleType1, typename ArrayHandleType2, typename Device>
class ArrayTransfer<typename ArrayHandleType1::ValueType,
                    StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>,
                    Device>
{
public:
  using ValueType = typename ArrayHandleType1::ValueType;

private:
  using StorageTag = StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>;
  using StorageType = vtkm::cont::internal::Storage<ValueType, StorageTag>;

public:
  using PortalControl = typename StorageType::PortalType;
  using PortalConstControl = typename StorageType::PortalConstType;

  using PortalExecution = ArrayPortalConcatenate<
    typename ArrayHandleType1::template ExecutionTypes<Device>::Portal,
    typename ArrayHandleType2::template ExecutionTypes<Device>::Portal>;
  using PortalConstExecution = ArrayPortalConcatenate<
    typename ArrayHandleType1::template ExecutionTypes<Device>::PortalConst,
    typename ArrayHandleType2::template ExecutionTypes<Device>::PortalConst>;

  VTKM_CONT
  ArrayTransfer(StorageType* storage)
    : Array1(storage->GetArray1())
    , Array2(storage->GetArray2())
  {
  }

  VTKM_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  VTKM_CONT
  PortalConstExecution PrepareForInput(bool vtkmNotUsed(updateData))
  {
    return PortalConstExecution(this->Array1.PrepareForInput(Device()),
                                this->Array2.PrepareForInput(Device()));
  }

  VTKM_CONT
  PortalExecution PrepareForInPlace(bool vtkmNotUsed(updateData))
  {
    return PortalExecution(this->Array1.PrepareForInPlace(Device()),
                           this->Array2.PrepareForInPlace(Device()));
  }

  VTKM_CONT
  PortalExecution PrepareForOutput(vtkm::Id numberOfValues)
  {
    // This implementation of allocate, which allocates the same amount in both
    // arrays, is arbitrary. It could, for example, leave the size of Array1
    // alone and change the size of Array2. Or, probably most likely, it could
    // simply throw an error and state that this operation is invalid.
    vtkm::Id half = numberOfValues / 2;
    return PortalExecution(
      this->Array1.PrepareForOutput(numberOfValues - half, Device()),
      this->Array2.PrepareForOutput(half, Device()));
  }

  VTKM_CONT
  void RetrieveOutputData(StorageType* vtkmNotUsed(storage)) const
  {
    // Implementation of this method should be unnecessary. The internal
    // array handles should automatically retrieve the output data as
    // necessary.
  }

  VTKM_CONT
  void Shrink(vtkm::Id numberOfValues)
  {
    if (numberOfValues < this->Array1.GetNumberOfValues())
    {
      this->Array1.Shrink(numberOfValues);
      this->Array2.Shrink(0);
    }
    else
    {
      this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
    }
  }

  VTKM_CONT
  void ReleaseResources()
  {
    this->Array1.ReleaseResourcesExecution();
    this->Array2.ReleaseResourcesExecution();
  }

private:
  ArrayHandleType1 Array1;
  ArrayHandleType2 Array2;
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

The final step to make a derived storage is to create a mechanism to construct an ArrayHandle with a storage derived from the desired arrays. This can be done by creating a trivial subclass of vtkm::cont::ArrayHandle that simply constructs the array handle to the state of an existing storage. It uses a protected constructor of vtkm::cont::ArrayHandle that accepts a constructed storage.

Example 36.15: ArrayHandle for derived storage of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
class ArrayHandleConcatenate
  : public vtkm::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleConcatenate,
    (ArrayHandleConcatenate<ArrayHandleType1, ArrayHandleType2>),
    (vtkm::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>));

private:
  using StorageType = vtkm::cont::internal::Storage<ValueType, StorageTag>;

public:
  VTKM_CONT
  ArrayHandleConcatenate(const ArrayHandleType1& array1,
                         const ArrayHandleType2& array2)
    : Superclass(StorageType(array1, array2))
  {
  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the types Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 36.15 is templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS macro. (The other macro is described in Section 36.2.1 on page 292). This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

vtkm::cont::ArrayHandleCompositeVector is an example of a derived array handle provided by VTK-m. It references some fixed number of other arrays, pulls a specified component out of each, and produces a new component that is a tuple of these retrieved components.

36.3 Adapting Data Structures

The intention of the storage parameter for vtkm::cont::ArrayHandle is to implement the strategy design pattern to enable VTK-m to interface directly with the data of any third party code source. VTK-m is designed to work with data originating in other libraries or applications. By creating a new type of storage, VTK-m can be entirely adapted to new kinds of data structures.

Common Errors

Keep in mind that memory layout used can have an effect on the running time of algorithms in VTK-m. Different data layouts and memory access can change cache performance and introduce memory affinity problems. The example code given in this section will likely have poorer cache performance than the basic storage provided by VTK-m. However, that might be an acceptable penalty to avoid data copies.

In this section we demonstrate the steps required to adapt the array handle to a data structure provided by a third party. For the purposes of the example, let us say that some fictitious library named "foo" has a simple structure named `FooFields` that holds the field values for a particular part of a mesh, and then maintain the field values for all locations in a mesh in a `std::deque` object.

Example 36.16: Fictitious field storage used in custom array storage examples.

```
#include <deque>

struct FooFields
{
  float Pressure;
  float Temperature;
  float Velocity[3];
  // And so on...
};

using FooFieldsDeque = std::deque<FooFields>;
```

VTK-m expects separate arrays for each of the fields rather than a single array containing a structure holding all of the fields. However, rather than copy each field to its own array, we can create a storage for each field that points directly to the data in a `FooFieldsDeque` object.

The first step in creating an adapter storage is to create a control environment array portal to the data. This is described in more detail in Section 27.1 and is generally straightforward for simple containers like this. Here is an example implementation for our `FooFieldsDeque` container.

Example 36.17: Array portal to adapt a third-party container to VTK-m.

```
#include <vtkm/Assert.h>
#include <vtkm/cont/internal/IteratorFromArrayPortal.h>

// DequeType expected to be either FooFieldsDeque or const FooFieldsDeque
template<typename DequeType>
class ArrayPortalFooPressure
{
public:
  using ValueType = float;

  VTKM_CONT
  ArrayPortalFooPressure()
    : Container(NULL)
  {
  }

  VTKM_CONT
  ArrayPortalFooPressure(DequeType* container)
    : Container(container)
  {
  }

  // Required to copy compatible types of ArrayPortalFooPressure. Really needed
  // to copy from non-const to const versions of array portals.
  template<typename OtherDequeType>
  VTKM_CONT ArrayPortalFooPressure(
    const ArrayPortalFooPressure<OtherDequeType>& other)
    : Container(other.GetContainer())
  {
  }

  VTKM_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT
  ValueType Get(vtkm::Id index) const
  {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    return (*this->Container)[index].Pressure;
  }

  VTKM_CONT
  void Set(vtkm::Id index, ValueType value) const
  {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    (*this->Container)[static_cast<std::size_t>(index)].Pressure = value;
  }

  // Here for the copy constructor.
  VTKM_CONT
  DequeType* GetContainer() const { return this->Container; }

private:
  DequeType* Container;
};
```

The next step in creating an adapter storage is to define a tag for the adapter. We shall call ours StorageTagFooPressure. Then, we need to create a specialization of the templated vtkm::cont::internal::Storage class. The ArrayHandle will instantiate an object using the array container tag we give it, and we define our own specialization so that it runs our interface into the code.

vtkm::cont::internal::Storage has two template arguments: the base type of the array and the storage tag.

Example 36.18: Prototype for vtkm::cont::internal::Storage.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, class StorageTag>
class Storage;
}
} // namespace cont
} // namespace vtkm
```

The vtkm::cont::internal::Storage must define the following items.

ValueType The type of each item in the array. This is the same type as the first template argument.

`PortalType` The type of an array portal that can be used to access the underlying data. This array portal needs to work only in the control environment.

`PortalConstType` A read-only (const) version of `PortalType`.

`GetPortal` A method that returns an array portal of type `PortalType` that can be used to access the data manged in this storage.

`GetPortalConst` Same as `GetPortal` except it returns a read-only (const) array portal.

`GetNumberOfValues` A method that returns the number of values the storage is currently allocated for.

`Allocate` A method that allocates the array to a given size. All values stored in the previous allocation may be destroyed.

`Shrink` A method like `Allocate` with two differences. First, the size of the allocation must be smaller than the existing allocation when the method is called. Second, any values currently stored in the array will be valid after the array is resized. This constrained form of allocation allows the array to be resized and values valid without ever having to copy data.

`ReleaseResources` A method that instructs the storage to free all of its memory.

The following provides an example implementation of our adapter to a `FooFieldsDeque`. It relies on the `ArrayPortalFooPressure` provided in Example 36.17.

Example 36.19: Storage to adapt a third-party container to VTK-m.

```
// Includes or definition for ArrayPortalFooPressure

struct StorageTagFooPressure
{
};

namespace vtkm
{
namespace cont
{
namespace internal
{

template<>
class Storage<float, StorageTagFooPressure>
{
public:
  using ValueType = float;

  using PortalType = ArrayPortalFooPressure<FooFieldsDeque>;
  using PortalConstType = ArrayPortalFooPressure<const FooFieldsDeque>;

  VTKM_CONT
  Storage()
    : Container(NULL)
  {
  }

  VTKM_CONT
  Storage(FooFieldsDeque* container)
    : Container(container)
  {
  }

  VTKM_CONT
  PortalType GetPortal() { return PortalType(this->Container); }

  VTKM_CONT
  PortalConstType GetPortalConst() const { return PortalConstType(this->Container); }

  VTKM_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT
  void Allocate(vtkm::Id numberOfValues)
  {
    this->Container->resize(static_cast<std::size_t>(numberOfValues));
  }

  VTKM_CONT
  void Shrink(vtkm::Id numberOfValues)
  {
    this->Container->resize(static_cast<std::size_t>(numberOfValues));
  }

  VTKM_CONT
  void ReleaseResources() { this->Container->clear(); }

private:
  FooFieldsDeque* Container;
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

The final step to make a storage adapter is to make a mechanism to construct an ArrayHandle that points to a particular storage. This can be done by creating a trivial subclass of vtkm::cont::ArrayHandle that simply constructs the array handle to the state of an existing container.

Example 36.20: Array handle to adapt a third-party container to VTK-m.

```
class ArrayHandleFooPressure
  : public vtkm::cont::ArrayHandle<float, StorageTagFooPressure>
{
private:
  using StorageType = vtkm::cont::internal::Storage<float, StorageTagFooPressure>;

public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
    ArrayHandleFooPressure,
    (vtkm::cont::ArrayHandle<float, StorageTagFooPressure>));

  VTKM_CONT
  ArrayHandleFooPressure(FooFieldsDeque* container)
    : Superclass(StorageType(container))
  {
  }
};
```

Subclasses of ArrayHandle provide constructors that establish the state of the array handle. All array handle subclasses must also use either the VTKM_ARRAY_HANDLE_SUBCLASS macro or the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. Both of these macros define the types Superclass, ValueType, and StorageTag as well as a set of constructors and operators expected of all ArrayHandle classes. The difference between these two macros is that VTKM_ARRAY_HANDLE_SUBCLASS is used in templated classes whereas VTKM_ARRAY_HANDLE_SUBCLASS_NT is used in non-templated classes.

The ArrayHandle subclass in Example 36.20 is not templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. (The other macro is described in Section 36.2.2 on page 293). This macro takes two parameters. The first parameter is the name of the subclass where the macro is defined and the second parameter is the immediate superclass including the full template specification. The second parameter of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

With this new version of ArrayHandle, VTK-m can now read to and write from the FooFieldsDeque structure directly.

Example 36.21: Using an ArrayHandle with custom container.

```
VTKM_CONT
void GetElevationAirPressure(vtkm::cont::DataSet grid, FooFieldsDeque* fields)
{
  // Make an array handle that points to the pressure values in the fields.
  ArrayHandleFooPressure pressureHandle(fields);

  // Use the elevation worklet to estimate atmospheric pressure based on the
  // height of the point coordinates. Atmospheric pressure is 101325 Pa at
  // sea level and drops about 12 Pa per meter.
  vtkm::worklet::PointElevation elevation;
  elevation.SetLowPoint(vtkm::make_Vec(0.0, 0.0, 0.0));
  elevation.SetHighPoint(vtkm::make_Vec(0.0, 0.0, 2000.0));
  elevation.SetRange(101325.0, 77325.0);

  vtkm::cont::Invoker invoke;
  invoke(elevation, grid.GetCoordinateSystem().GetData(), pressureHandle);

  // Make sure the values are flushed back to the control environment.
  pressureHandle.SyncControlArray();

  // Now the pressure field is in the fields container.
}
```

Common Errors

When using an ArrayHandle *in VTK-m some code may be executed in an execution environment with a different memory space. In these cases data written to an* ArrayHandle *with a custom storage will not be written directly to the storage system you defined. Rather, they will be written to a separate array in the execution environment. If you need to access data in your custom data structure, make sure you call* SyncControlArray *on the* ArrayHandle*, as is demonstrated in Example 36.21.*

Most of the code in VTK-m will create ArrayHandles using the default storage, which is set to the basic storage if not otherwise specified. If you wish to replace the default storage used, then set the VTKM_STORAGE macro to VTKM_STORAGE_UNDEFINED and set the VTKM_DEFAULT_STORAGE_TAG to your tag class. These definitions have to happen *before* including any VTK-m header files. You will also have to declare the tag class (or at least a prototype of it) before including VTK-m header files.

Example 36.22: Redefining the default array handle storage.

```
#define VTKM_STORAGE VTKM_STORAGE_UNDEFINED
#define VTKM_DEFAULT_STORAGE_TAG StorageTagFooPressure

struct StorageTagFooPressure;
```

Common Errors

`ArrayHandle`*s are often stored in dynamic objects like variant arrays (Chapter 33) or data sets (Chapter 7). When this happens, the array's type information, including the storage used, is lost. VTK-m will execute algorithms using the* `ArrayHandleVirtual` *interface. For hot path types and storages for filters it is good to specify custom sets in the policy when executing filters.*

Part V

Core Development

CHAPTER

THIRTYSEVEN

TRY EXECUTE

Most operations in VTK-m do not require specifying on which device to run. For example, you may have noticed that when using vtkm::cont::Invoker to execute a worklet, you do not need to specify a device; it chooses a device for you. Internally, the Invoker has a mechanism to automatically select a device, try it, and fall back to other devices if the first one fails. We saw this at work in the implementation of filters in Chapter 22.

The Invoker is internally using a function named vtkm::cont::TryExecute to choose a device. This TryExecute function can be also be used in other instances where a specific device needs to be chosen.

TryExecute is a simple, generic mechanism to run an algorithm that requires a device adapter without directly specifying a device adapter. vtkm::cont::TryExecute is a templated function. The first argument is a functor object whose parenthesis operator takes a device adapter tag and returns a bool that is true if the call succeeds on the given device. If any further arguments are given to TryExecute, they are passed on to the functor. Thus, the parenthesis operator on the functor should take a device adapter tag as its first argument and any remaining arguments must match those passed to TryExecute.

To demonstrate the operation of TryExecute, consider an operation to find the average value of an array. Doing so with a given device adapter is a straightforward use of the reduction operator.

Example 37.1: A function to find the average value of an array in parallel.

```
template<typename T, typename Storage, typename Device>
VTKM_CONT T ArrayAverage(const vtkm::cont::ArrayHandle<T, Storage>& array, Device)
{
  T sum = vtkm::cont::Algorithm::Reduce(array, T(0));
  return sum / T(array.GetNumberOfValues());
}
```

The function in Example 37.1 requires a device adapter. We want to make an alternate version of this function that does not need a specific device adapter but rather finds one to use. To do this, we first make a functor as described earlier. It takes a device adapter tag as an argument, calls the version of the function shown in Example 37.1, and returns true when the operation succeeds. We then create a new version of the array average function that does not need a specific device adapter tag and calls TryExecute with the aforementioned functor.

Example 37.2: Using TryExecute.

```
namespace detail
{

struct ArrayAverageFunctor
{
  template<typename Device, typename T, typename Storage>
  VTKM_CONT bool operator()(Device,
                            const vtkm::cont::ArrayHandle<T, Storage>& inArray,
                            T& outValue) const
  {
    // Call the version of ArrayAverage that takes a DeviceAdapter.
    outValue = ArrayAverage(inArray, Device());

    return true;
  }
};

} // namespace detail

template<typename T, typename Storage>
VTKM_CONT T ArrayAverage(const vtkm::cont::ArrayHandle<T, Storage>& array)
{
  T outValue;

  bool foundAverage =
    vtkm::cont::TryExecute(detail::ArrayAverageFunctor{}, array, outValue);

  if (!foundAverage)
  {
    throw vtkm::cont::ErrorExecution("Could not compute array average.");
  }

  return outValue;
}
```

Common Errors

When `TryExecute` *calls the operation of your functor, it will catch any exceptions that the functor might throw.* `TryExecute` *will interpret any thrown exception as a failure on that device and try another device. If all devices fail,* `TryExecute` *will return a false value rather than throw its own exception. This means if you want to have an exception thrown from a call to* `TryExecute`*, you will need to check the return value and throw the exception yourself.*

CHAPTER

THIRTYEIGHT

IMPLEMENTING DEVICE ADAPTERS

VTK-m comes with several implementations of device adapters so that it may be ported to a variety of platforms. It is also possible to provide new device adapters to support yet more devices, compilers, and libraries. A new device adapter provides a tag, a class to manage arrays in the execution environment, a class to establish virtual objects in the execution environment, a collection of algorithms that run in the execution environment, and (optionally) a timer.

Most device adapters are associated with some type of device or library, and all source code related directly to that device is placed in a subdirectory of vtkm/cont. For example, files associated with CUDA are in vtkm/cont/cuda, files associated with the Intel Threading Building Blocks (TBB) are located in vtkm/cont/tbb, and files associated with OpenMP are in vtkm/cont/openmp. The documentation here assumes that you are adding a device adapter to the VTK-m source code and following these file conventions.

For the purposes of discussion in this section, we will give a simple example of implementing a device adapter using the `std::thread` class provided by C++11. We will call our device `Cxx11Thread` and place it in the directory vtkm/cont/cxx11.

By convention the implementation of device adapters within VTK-m are divided into 6 header files with the names DeviceAdapterTag∗.h, DeviceAdapterRuntimeDetector∗.h, ArrayManagerExecution∗.h, VirtualObjectTransfer∗.h, AtomicInterfaceExecution∗.h and DeviceAdapterAlgorithm∗.h, which are hidden in internal directories. The DeviceAdapter∗.h that most code includes is a trivial header that simply includes these other 6 files. For our example `std::thread` device, we will create the base header at vtkm/cont/cxx11/DeviceAdapterCxx11Thread.h. The contents are the following (with minutia like include guards removed).

Example 38.1: Contents of the base header for a device adapter.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/DeviceAdapterRuntimeDetectorCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/VirtualObjectTransferCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/AtomicInterfaceExecutionCxx11Thread.h>
#include <vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h>
```

The reason VTK-m breaks up the code for its device adapters this way is that there is an interdependence between the implementation of each device adapter and the mechanism to pick a default device adapter. Breaking up the device adapter code in this way maintains an acyclic dependence among header files.

38.1 Tag

The device adapter tag, as described in Section 12.1 is a simple empty type that is used as a template parameter to identify the device adapter. Every device adapter implementation provides one. The device adapter tag is typically defined in an internal header file with a prefix of DeviceAdapterTag.

The device adapter tag should be created with the macro VTKM_VALID_DEVICE_ADAPTER. This adapter takes an abbreviated name that it will append to DeviceAdapterTag to make the tag structure. It will also create some support classes that allow VTK-m to introspect the device adapter. The macro also expects a unique integer identifier that is usually stored in a macro prefixed with VTKM_DEVICE_ADAPTER_. These identifiers for the device adapters provided by the core VTK-m are declared in vtkm/cont/internal/DeviceAdapterTag.h.

The following example gives the implementation of our custom device adapter, which by convention would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h header file.

Example 38.2: Implementation of a device adapter tag.

```
#include <vtkm/cont/DeviceAdapterTag.h>

// If this device adapter were to be contributed to VTK-m, then this macro
// declaration should be moved to DeviceAdapterTag.h and given a unique
// number. It also has te be less than VTK_MAX_DEVICE_ADAPTER_ID
#define VTKM_DEVICE_ADAPTER_CXX11_THREAD 7

VTKM_VALID_DEVICE_ADAPTER(Cxx11Thread, VTKM_DEVICE_ADAPTER_CXX11_THREAD);
```

38.2 Runtime Detector

VTK-m defines a template named vtkm::cont::DeviceAdapterRuntimeDetector that provides the ability to detect whether a given device is available on the current system. DeviceAdapterRuntimeDetector has a single template argument that is the device adapter tag.

Example 38.3: Prototype for DeviceAdapterRuntimeDetector.

```
namespace vtkm
{
namespace cont
{

template<typename DeviceAdapterTag>
class DeviceAdapterRuntimeDetector;
}
} // namespace vtkm
```

All device adapter implementations must create a specialization of DeviceAdapterRuntimeDetector. They must contain a method named DeviceAdapterRuntimeDetector::Exists that returns a true or false value to indicate whether the device is available on the current runtime system. For our simple C++ threading example, the C++ threading is always available (even if only one such processing element exists) so our implementation simply returns true if the device has been compiled.

Example 38.4: Implementation of DeviceAdapterRuntimeDetector specialization

```
namespace vtkm
{
namespace cont
{

template<>
class DeviceAdapterRuntimeDetector<vtkm::cont::DeviceAdapterTagCxx11Thread>
{
public:
  VTKM_CONT bool Exists() const
  {
    return vtkm::cont::DeviceAdapterTagCxx11Thread::IsEnabled;
  }
};

} // namespace cont
} // namespace vtkm
```

38.3 Array Manager Execution

VTK-m defines a template named vtkm::cont::internal::ArrayManagerExecution that is responsible for allocating memory in the execution environment and copying data between the control and execution environment. ArrayManagerExecution is also paired with two helper classes, vtkm::cont::internal::ExecutionPortalFactoryBasic and vtkm::cont::internal::ExecutionArrayInterfaceBasic, which provide operations for creating and operating on and manipulating data in standard C arrays. All 3 class specializations are typically defined in an internal header file with a prefix of ArrayManagerExecution. The following subsections describe each of these classes.

38.3.1 ArrayManagerExecution

Example 38.5: Prototype for vtkm::cont::internal::ArrayManagerExecution.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, typename StorageTag, typename DeviceAdapterTag>
class ArrayManagerExecution;
}
} // namespace cont
} // namespace vtkm
```

A device adapter must provide a partial specialization of vtkm::cont::internal::ArrayManagerExecution for its device adapter tag. The implementation for ArrayManagerExecution is expected to manage the resources for a single array. All ArrayManagerExecution specializations must have a constructor that takes a pointer to a vtkm::cont::internal::Storage object. The ArrayManagerExecution should store a reference to this Storage object and use it to pass data between control and execution environments. Additionally, ArrayManagerExecution must provide the following elements.

ValueType The type for each item in the array. This is the same type as the first template argument.

PortalType The type of an array portal that can be used in the execution environment to access the array.

PortalConstType A read-only (const) version of PortalType.

GetNumberOfValues A method that returns the number of values stored in the array. The results are undefined if the data has not been loaded or allocated.

PrepareForInput A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a bool flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a PortalConstType that points to the data.

PrepareForInPlace A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a bool flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a PortalType that points to the data.

PrepareForOutput A method that takes an array size and allocates an array in the execution environment of the specified size. The initial memory may be uninitialized. The method returns a PortalType to the data.

RetrieveOutputData This method takes a storage object, allocates memory in the control environment, and copies data from the execution environment into it. If the control and execution environments share arrays, then this can be a no-operation.

CopyInto This method takes an STL-compatible iterator and copies data from the execution environment into it.

Shrink A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked.

ReleaseResources A method that frees any resources (typically memory) in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying memory in methods such as PrepareForInput and RetrieveOutputData. This might require creating buffers in the control environment to efficiently move data from control array portals.

However, if the control and execution environments share the same memory space, the execution array manager can, and should, delegate all of its operations to the Storage it is constructed with. VTK-m comes with a class called vtkm::cont::internal::ArrayManagerExecutionShareWithControl that provides the implementation for an execution array manager that shares a memory space with the control environment. In this case, making the ArrayManagerExecution specialization be a trivial subclass is sufficient.

Continuing our example of a device adapter based on C++11's std::thread class, here is the implementation of ArrayManagerExecution, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 38.6: Specialization of ArrayManagerExecution.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecution.h>
#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, typename StorageTag>
class ArrayManagerExecution<T, StorageTag, vtkm::cont::DeviceAdapterTagCxx11Thread>
  : public vtkm::cont::internal::ArrayManagerExecutionShareWithControl<T, StorageTag>
{
  using Superclass =
    vtkm::cont::internal::ArrayManagerExecutionShareWithControl<T, StorageTag>;

public:
  VTKM_CONT
  ArrayManagerExecution(typename Superclass::StorageType* storage)
    : Superclass(storage)
  {
  }
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

38.3.2 ExecutionPortalFactoryBasic

Example 38.7: Prototype for vtkm::cont::internal::ExecutionPortalFactoryBasic.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, typename DeviceAdapterTag>
struct ExecutionPortalFactoryBasic;
}
} // namespace cont
} // namespace vtkm
```

A device adapter must provide a partial specialization of vtkm::cont::internal::ExecutionPortalFactoryBasic for its device adapter tag. The implementation for ExecutionPortalFactoryBasic is capable of taking pointers to an array in the execution environment and returning an array portal to the data in that array. ExecutionPortalFactoryBasic has no state and all of its methods are static. ExecutionPortalFactoryBasic provides the following elements.

ValueType The type for each item in the array. This is the same type as the first template argument.

PortalType The type for the read/write portals created by the class.

PortalConstType The type for read-only portals created by the class.

CreatePortal A static method that takes two pointers of type ValueType* that point to the beginning (first element) and end (one past the last element) of the array to create a portal for the array. Returns a portal of type PortalType that works in the execution environment.

CreatePortalConst A static method that takes two pointers of type const ValueType* that point to the beginning (first element) and end (one past the last element) of the array to create a portal for the array. Returns a portal of type PortalConstType that works in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments share the same memory space, then the execution array manager can, and should, return a simple vtkm::cont::internal::ArrayPortalFromIterators. VTK-m comes with a class called vtkm::cont::internal::ExecutionPortalFactoryBasicShareWithControl that provides the implementation for a basic execution portal factory that shares a memory space with the control environment. In this case, making the ExecutionPortalFactoryBasic specialization be a trivial subclass is sufficient.

However, if the control and execution environments have separate memory spaces, then the typical vtkm::cont::internal::ArrayPortalFromIterators class might not work in the execution environment. In this case a custom array portal class will have to be implemented and created within the ExecutionPortalFactoryBasic.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `ExecutionPortalFactoryBasic`, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 38.8: Specialization of `ExecutionPortalFactoryBasic`.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T>
struct ExecutionPortalFactoryBasic<T, vtkm::cont::DeviceAdapterTagCxx11Thread>
  : public vtkm::cont::internal::ExecutionPortalFactoryBasicShareWithControl<T>
{
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

38.3.3 `ExecutionArrayInterfaceBasic`

Example 38.9: Prototype for `vtkm::cont::internal::ExecutionArrayInterfaceBasic`.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename DeviceAdapterTag>
struct ExecutionArrayInterfaceBasic;
}
} // namespace cont
} // namespace vtkm
```

A device adapter must provide a partial specialization of `vtkm::cont::internal::ExecutionArrayInterfaceBasic` for its device adapter tag. The implementation for `ExecutionArrayInterfaceBasic` is expected to allow allocation of basic C arrays in the execution environment and to copy data between control and execution environments. All implementations of `ExecutionArrayInterfaceBasic` are expected to inherit from `vtkm::cont::internal::ExecutionArrayInterfaceBasicBase` and implement the pure virtual methods therein. All implementations are also expected to have a constructor that takes a reference to a `vtkm::cont::internal::StorageBasicBase`, which should subsequently be passed to the `ExecutionArrayInterfaceBasicBase`. The methods that `vtkm::cont::internal::ExecutionArrayInterfaceBasic` must override are the following.

`GetDeviceId` Returns a `vtkm::cont::DeviceAdapterId` integer representing a unique identifier for the device associated with this implementation. This number should be the same as the `VTKM_DEVICE_ADAPTER_` macro described in Section 38.1.

`Allocate` Takes a reference to a `vtkm::cont::internal::TypelessExecutionArray`, which holds a collection of array pointer references, and a size of allocation in bytes. The method should allocate an array of the given size in the execution environment and return the resulting pointers in the given TypelessExecutionArray reference.

`Free` Takes a reference to a `vtkm::cont::internal::TypelessExecutionArray` created in a previous call to `Allocate` and frees the memory. The array references in the `TypelessExecutionArray` should be set to `nullptr`.

`CopyFromControl` Takes a `const void*` pointer for an array in the control environment, a `void*` for an array in the execution environment, and a size of the arrays in bytes. The method copies the data from the control array to the execution array.

`CopyToControl` Takes a `const void*` pointer for an array in the execution environment, a `void*` for an array in the control environment, and a size of the arrays in bytes. The method copies the data from the execution array to the control array.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by allocating arrays on the device and copying data between the main CPU and the device.

However, if the control and execution environments share the same memory space, then the execution array interface can, and should, use the storage on the control environment to allocate arrays and do simple copies (shallow where possible). VTK-m comes with a class called `vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl` that provides the implementation for a basic execution array interface that shares a memory space with the control environment. In this case, it is best to make the `ExecutionArrayInterfaceBasic` specialization a subclass of `ExecutionArrayInterfaceBasicShareWithControl`. Note that in this case you still need to define a constructor that takes a reference to `vtkm::cont::internal::StorageBasicBase` (which is passed straight to the superclass) and an implementation of the `GetDeviceId` method.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `ExecutionArrayInterfaceBasic`, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 38.10: Specialization of `ExecutionArrayInterfaceBasic`.

```cpp
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<>
struct ExecutionArrayInterfaceBasic<vtkm::cont::DeviceAdapterTagCxx11Thread>
  : public vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl
{
public:
  using Superclass =
    vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl;

  VTKM_CONT
  ExecutionArrayInterfaceBasic(vtkm::cont::internal::StorageBasicBase& storage)
    : Superclass(storage)
  {
  }

  VTKM_CONT
  virtual vtkm::cont::DeviceAdapterId GetDeviceId() const final override
  {
    return vtkm::cont::DeviceAdapterTagCxx11Thread();
  }
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

38.4 Virtual Object Transfer

VTK-m defines a template named vtkm::cont::internal::VirtualObjectTransfer that is responsible for instantiating virtual objects in the execution environment. Chapter 35 discusses how to design and implement virtual objects. The VirtualObjectTransfer class is the internal mechanism that allocates space for the object and sets up the virtual method tables for them. This class has two template parameters. The first parameter is the concrete derived type of the virtual object to be transferred to the execution environment. It is assumed that after the object is copied to the execution environment, a pointer to a base superclass of this concrete derived type will be used. The second template argument is the device adapter on which to put the object.

Example 38.11: Prototype for vtkm::cont::internal::VirtualObjectTransfer.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename VirtualDerivedType, typename DeviceAdapter>
struct VirtualObjectTransfer;
}
} // namespace cont
} // namespace vtkm
```

A device adapter must provide a partial specialization of VirtualObjectTransfer for its device adapter tag. This partial specialization is typically defined in an internal header file with a prefix of VirtualObjectTransfer. The implementation for VirtualObjectTransfer can establish a virtual object in the execution environment based on an object in the control environment, update the state of said object, and release all the resources for the object. VirtualObjectTransfer must provide the following methods.

VirtualObjectTransfer (constructor) A VirtualObjectTransfer has a constructor that takes a pointer to the derived type that (eventually) gets transferred to the execution environment of the given device adapter. The object provided must stay valid for the lifespan of the VirtualObjectTransfer object.

PrepareForExecution Transfers the virtual object (given in the constructor) to the execution environment and returns a pointer to the object that can be used in the execution environment. The returned object may not be valid in the control environment and should not be used there. PrepareForExecution takes a single bool argument. If the argument is false and PrepareForExecution was called previously, then the method can return the same data as the last call without any updates. If the argument is true, then the data in the execution environment is always updated regardless of whether data was copied in a previous call to PrepareForExecution. This argument is used to tell the VirtualObjectTransfer whether the object in the control environment has changed and has to be updated in the execution environment.

`ReleaseResources` Frees up any resources in the execution environment. Any previously returned virtual object from `PrepareForExecution` becomes invalid. (The destructor for `VirtualObjectTransfer` should also release the resources.)

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying the concrete control object to the execution environment (where the virtual table will be invalid) and a new object is created in the execution environment by copying the object from the control environment. It can be assumed that the object can be trivially copied (with the exception of the virtual method table).

Did you know?

For some devices, like CUDA, it is either only possible or more efficient to allocate data from the host (the control environment). To avoid having to allocate data from the device (the execution environment), implement `PrepareForExecution` *by first allocating data from the host and then running code on the device that does a "placement new" to create and copy the object in the pre-allocated space.*

However, if the control and execution environments share the same memory space, the virtual object transfer can, and should, just bind directly with the target concrete object. VTK-m comes with a class called `vtkm::cont::internal::VirtualObjectTransferShareWithControl` that provides the implementation for a virtual object transfer that shares a memory space with the control environment. In this case, making the `VirtualObjectTransfer` specialization be a trivial subclass is sufficient. Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `VirtualObjectTransfer`, which by convention would be placed in the vtkm/cont/cxx11/internal/VirtualObjectTransferCxx11Thread.h header file.

Example 38.12: Specialization of `VirtualObjectTransfer`.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/VirtualObjectTransfer.h>
#include <vtkm/cont/internal/VirtualObjectTransferShareWithControl.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename VirtualDerivedType>
struct VirtualObjectTransfer<VirtualDerivedType,
                             vtkm::cont::DeviceAdapterTagCxx11Thread>
  : VirtualObjectTransferShareWithControl<VirtualDerivedType>
{
  VTKM_CONT VirtualObjectTransfer(const VirtualDerivedType* virtualObject)
    : VirtualObjectTransferShareWithControl<VirtualDerivedType>(virtualObject)
  {
  }
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

38.5 Atomic Interface Execution

VTK-m defines a template named vtkm::cont::internal::AtomicInterfaceExecution that is responsible for performing atomic operations on raw addresses. vtkm::cont::internal::AtomicInterfaceExecution defines a WordTypePreferred member that is the fastest available for bitwise operations of the given device. At minimum, the interface must support operations on WordTypePreferred and vtkm::WordTypeDefault, which may be the same. A full list of the supported word types is advertised in the type list stored in WordTypes. vtkm::cont::internal::AtomicInterfaceExecution must provide the following methods for inputs of both pointer types WordTypePrefered and vtkm::WordTypeDefault:

Load Atomically load a value from memory while enforcing, at a minimum, ”acquire” memory ordering.

Store Atomically write a value to memory while enforcing, at a minimum ”release” memory ordering.

Not Perform a bitwise atomic not operation on the word at the provided address.

And Perform a bitwise atomic and operation on the word at the provided address.

Or Perform a bitwise atomic or operation on the word at the provided address.

Xor Perform a bitwise atomic xor operation on the word at the provided address.

CompareAndSwap Perform a bitwise atomic Compare and Swap operation on the word at the provided address.

VTK-m provides a default implementation of the above operations that can be used on devices that share the control environment. In order to use this implementation, simply subclass vtkm::cont::internal::AtomicInterfaceControl from a template specialization of vtkm::cont::internal::AtomicInterfaceExecution that takes a template argument of the new device adapter tag.

Example 38.13: Specialization of AtomicInterfaceExecution.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/AtomicInterfaceControl.h>
#include <vtkm/cont/internal/AtomicInterfaceExecution.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<>
class AtomicInterfaceExecution<DeviceAdapterTagCxx11Thread>
  : public AtomicInterfaceControl
{
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

38.6 Algorithms

A device adapter implementation must also provide a specialization of vtkm::cont::DeviceAdapterAlgorithm, which provides the underlying implementation of the algorithms described in Chapter 34. The implementation for the device adapter algorithms is typically placed in a header file with a prefix of DeviceAdapterAlgorithm.

Although there are many methods in DeviceAdapterAlgorithm, it is seldom necessary to implement them all. Instead, VTK-m comes with vtkm::cont::internal::DeviceAdapterAlgorithmGeneral that provides generic implementation for most of the required algorithms. By deriving the specialization of DeviceAdapterAlgorithm from DeviceAdapterAlgorithmGeneral, only the implementations for Schedule and Synchronize need to be implemented. All other algorithms can be derived from those.

That said, not all of the algorithms implemented in DeviceAdapterAlgorithmGeneral are optimized for all types of devices. Thus, it is worthwhile to provide algorithms optimized for the specific device when possible. In particular, it is best to provide specializations for the sort, scan, and reduce algorithms.

It is standard practice to implement a specialization of DeviceAdapterAlgorithm by having it inherit from vtkm::cont::internal::DeviceAdapterAlgorithmGeneral and specializing those methods that are optimized for a particular system. DeviceAdapterAlgorithmGeneral is a templated class that takes as its single template parameter the type of the subclass. For example, a device adapter algorithm structure named DeviceAdapterAlgorithm<DeviceAdapterTagFoo> will subclass DeviceAdapterAlgorithmGeneral<DeviceAdapterAlgorithm<DeviceAdapterTagFoo> >.

Did you know?

The convention of having a subclass be templated on the derived class' type is known as the Curiously Recurring Template Pattern (CRTP). In the case of DeviceAdapterAlgorithmGeneral*, VTK-m uses this CRTP behavior to allow the general implementation of these algorithms to run* Schedule *and other specialized algorithms in the subclass.*

One point to note when implementing the Schedule methods is to make sure that errors handled in the execution environment are handled correctly. As described in Chapter 23, errors are signaled in the execution environment by calling RaiseError on a functor or worklet object. This is handled internally by the vtkm::exec::internal::ErrorMessageBuffer class. ErrorMessageBuffer really just holds a small string buffer, which must be provided by the device adapter's Schedule method.

So, before Schedule executes the functor it is given, it should allocate a small string array in the execution environment, initialize it to the empty string, encapsulate the array in an ErrorMessageBuffer object, and set this buffer object in the functor. When the execution completes, Schedule should check to see if an error exists in this buffer and throw a vtkm::cont::ErrorExecution if an error has been reported.

Common Errors

Exceptions are generally not supposed to be thrown in the execution environment, but it could happen on devices that support them. Nevertheless, few thread schedulers work well when an exception is thrown in them. Thus, when implementing adapters for devices that do support exceptions, it is good practice to catch them within the thread and report them through the ErrorMessageBuffer*.*

The following example is a minimal implementation of device adapter algorithms using C++11's std::thread class. Note that no attempt at providing optimizations has been attempted (and many are possible). By convention this code would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h header file.

Example 38.14: Minimal specialization of DeviceAdapterAlgorithm.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/DeviceAdapterAlgorithm.h>
#include <vtkm/cont/ErrorExecution.h>
#include <vtkm/cont/internal/DeviceAdapterAlgorithmGeneral.h>

#include <thread>

namespace vtkm
{
namespace cont
{

template<>
struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
  : vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
      DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>,
      vtkm::cont::DeviceAdapterTagCxx11Thread>
{
private:
  template<typename FunctorType>
  struct ScheduleKernel1D
  {
    VTKM_CONT
    ScheduleKernel1D(const FunctorType& functor)
      : Functor(functor)
    {
    }

    VTKM_EXEC
    void operator()() const
    {
      try
      {
        for (vtkm::Id threadId = this->BeginId; threadId < this->EndId; threadId++)
        {
          this->Functor(threadId);
          // If an error is raised, abort execution.
          if (this->ErrorMessage.IsErrorRaised())
          {
            return;
          }
        }
      }
      catch (const vtkm::cont::Error& error)
      {
        this->ErrorMessage.RaiseError(error.GetMessage().c_str());
      }
      catch (const std::exception& error)
      {
        this->ErrorMessage.RaiseError(error.what());
      }
      catch (...)
      {
        this->ErrorMessage.RaiseError("Unknown exception raised.");
      }
    }

    FunctorType Functor;
    vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
    vtkm::Id BeginId;
    vtkm::Id EndId;
  };

```

```
template<typename FunctorType>
struct ScheduleKernel3D
{
  VTKM_CONT
  ScheduleKernel3D(const FunctorType& functor, vtkm::Id3 maxRange)
    : Functor(functor)
    , MaxRange(maxRange)
  {
  }

  VTKM_EXEC
  void operator()() const
  {
    vtkm::Id3 threadId3D(this->BeginId % this->MaxRange[0],
                         (this->BeginId / this->MaxRange[0]) % this->MaxRange[1],
                         this->BeginId / (this->MaxRange[0] * this->MaxRange[1]));

    try
    {
      for (vtkm::Id threadId = this->BeginId; threadId < this->EndId; threadId++)
      {
        this->Functor(threadId3D);
        // If an error is raised, abort execution.
        if (this->ErrorMessage.IsErrorRaised())
        {
          return;
        }

        threadId3D[0]++;
        if (threadId3D[0] >= MaxRange[0])
        {
          threadId3D[0] = 0;
          threadId3D[1]++;
          if (threadId3D[1] >= MaxRange[1])
          {
            threadId3D[1] = 0;
            threadId3D[2]++;
          }
        }
      }
    }
    catch (const vtkm::cont::Error& error)
    {
      this->ErrorMessage.RaiseError(error.GetMessage().c_str());
    }
    catch (const std::exception& error)
    {
      this->ErrorMessage.RaiseError(error.what());
    }
    catch (...)
    {
      this->ErrorMessage.RaiseError("Unknown exception raised.");
    }
  }

  FunctorType Functor;
  vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
  vtkm::Id BeginId;
  vtkm::Id EndId;
  vtkm::Id3 MaxRange;
};

template<typename KernelType>
VTKM_CONT static void DoSchedule(KernelType kernel, vtkm::Id numInstances)
```

```
  {
    if (numInstances < 1)
    {
      return;
    }

    const vtkm::Id MESSAGE_SIZE = 1024;
    char errorString[MESSAGE_SIZE];
    errorString[0] = '\0';
    vtkm::exec::internal::ErrorMessageBuffer errorMessage(errorString, MESSAGE_SIZE);
    kernel.Functor.SetErrorMessageBuffer(errorMessage);
    kernel.ErrorMessage = errorMessage;

    vtkm::Id numThreads = static_cast<vtkm::Id>(std::thread::hardware_concurrency());
    if (numThreads > numInstances)
    {
      numThreads = numInstances;
    }
    vtkm::Id numInstancesPerThread = (numInstances + numThreads - 1) / numThreads;

    std::thread* threadPool = new std::thread[numThreads];
    vtkm::Id beginId = 0;
    for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
    {
      vtkm::Id endId = std::min(beginId + numInstancesPerThread, numInstances);
      KernelType threadKernel = kernel;
      threadKernel.BeginId = beginId;
      threadKernel.EndId = endId;
      std::thread newThread(threadKernel);
      threadPool[threadIndex].swap(newThread);
      beginId = endId;
    }

    for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
    {
      threadPool[threadIndex].join();
    }

    delete[] threadPool;

    if (errorMessage.IsErrorRaised())
    {
      throw vtkm::cont::ErrorExecution(errorString);
    }
  }

public:
  template<typename FunctorType>
  VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id numInstances)
  {
    DoSchedule(ScheduleKernel1D<FunctorType>(functor), numInstances);
  }

  template<typename FunctorType>
  VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id3 maxRange)
  {
    vtkm::Id numInstances = maxRange[0] * maxRange[1] * maxRange[2];
    DoSchedule(ScheduleKernel3D<FunctorType>(functor, maxRange), numInstances);
  }

  VTKM_CONT
  static void Synchronize()
  {
    // Nothing to do. This device schedules all of its operations using a
    // split/join paradigm. This means that the if the control threaad is
    // calling this method, then nothing should be running in the execution
    // environment.
  }
};

} // namespace cont
} // namespace vtkm
```

38.7 Timer Implementation

The VTK-m timer, described in Chapter 13, delegates to an internal class named `vtkm::cont::DeviceAdapterTimerImplementation`. The interface for this class is the same as that for `vtkm::cont::Timer`. A default implementation of this templated class uses the system timer and the `Synchronize` method in the device adapter algorithms.

However, some devices might provide alternate or better methods for implementing timers. For example, the TBB and CUDA libraries come with high resolution timers that have better accuracy than the standard system timers. Thus, the device adapter can optionally provide a specialization of `DeviceAdapterTimerImplementation`, which is typically placed in the same header file as the device adapter algorithms.

Continuing our example of a custom device adapter using C++11's `std::thread` class, we could use the default timer and it would work fine. But C++11 also comes with a `std::chrono` package that contains some portable time functions. The following code demonstrates creating a custom timer for our device adapter using this package. By convention, `DeviceAdapterTimerImplementation` is placed in the same header file as `DeviceAdapterAlgorithm`.

Example 38.15: Specialization of `DeviceAdapterTimerImplementation`.

```
#include <chrono>

namespace vtkm
{
namespace cont
{

template<>
class DeviceAdapterTimerImplementation<vtkm::cont::DeviceAdapterTagCxx11Thread>
{
public:
  VTKM_CONT
  DeviceAdapterTimerImplementation() { this->Reset(); }

  VTKM_CONT
  void Reset()
  {
    vtkm::cont::DeviceAdapterAlgorithm<
      vtkm::cont::DeviceAdapterTagCxx11Thread>::Synchronize();
    this->StartTime = std::chrono::high_resolution_clock::now();
  }

  VTKM_CONT
  vtkm::Float64 GetElapsedTime()
  {
    vtkm::cont::DeviceAdapterAlgorithm<
      vtkm::cont::DeviceAdapterTagCxx11Thread>::Synchronize();
    std::chrono::high_resolution_clock::time_point endTime =
      std::chrono::high_resolution_clock::now();

    std::chrono::high_resolution_clock::duration elapsedTicks =
      endTime - this->StartTime;

    std::chrono::duration<vtkm::Float64> elapsedSeconds(elapsedTicks);

    return elapsedSeconds.count();
  }

private:
  std::chrono::high_resolution_clock::time_point StartTime;
};

} // namespace cont
} // namespace vtkm
```

CHAPTER

THIRTYNINE

FUNCTION INTERFACE OBJECTS

For flexibility's sake a worklet is free to declare a ControlSignature with whatever number of arguments are sensible for its operation. The Invoker is expected to support arguments that match these arguments, and part of the invocation operation may require these arguments to be augmented before the worklet is scheduled. This leaves the invoker with the tricky task of managing some collection of arguments of unknown size and unknown types.

To simplify this management, VTK-m has the vtkm::internal::FunctionInterface class. FunctionInterface is a templated class that manages a generic set of arguments and return value from a function. An instance of FunctionInterface holds an instance of each argument. You can apply the arguments in a FunctionInterface object to a functor of a compatible prototype, and the resulting value of the function call is saved in the FunctionInterface.

39.1 Declaring and Creating

vtkm::internal::FunctionInterface is a templated class with a single parameter. The parameter is the *signature* of the function. A signature is a function type. The syntax in C++ is the return type followed by the argument types encased in parentheses.

Example 39.1: Declaring vtkm::internal::FunctionInterface.

```
// FunctionInterfaces matching some common POSIX functions.
vtkm::internal::FunctionInterface<size_t(const char*)> strlenInterface;

vtkm::internal::FunctionInterface<char*(char*, const char* s2, size_t)>
  strncpyInterface;
```

The vtkm::internal::make_FunctionInterface function provies an easy way to create a FunctionInterface and initialize the state of all the parameters. make_FunctionInterface takes a variable number of arguments, one for each parameter. Since the return type is not specified as an argument, you must always specify it as a template parameter.

Example 39.2: Using vtkm::internal::make_FunctionInterface.

```
const char* s = "Hello World";
static const size_t BUFFER_SIZE = 100;
char* buffer = (char*)malloc(BUFFER_SIZE);

strlenInterface = vtkm::internal::make_FunctionInterface<size_t>(s);

strncpyInterface =
  vtkm::internal::make_FunctionInterface<char*>(buffer, s, BUFFER_SIZE);
```

39.2 Parameters

One created, FunctionInterface contains methods to query and manage the parameters and objects associated with them. The number of parameters can be retrieved either with the constant field ARITY or with the GetArity method.

Example 39.3: Getting the arity of a FunctionInterface.

```
VTKM_STATIC_ASSERT(vtkm::internal::FunctionInterface<size_t(const char*)>::ARITY ==
                   1);

vtkm::IdComponent arity = strncpyInterface.GetArity(); // arity = 3
```

To get a particular parameter, FunctionInterface has the templated method GetParameter. The template parameter is the index of the parameter. Note that the parameters in FunctionInterface start at index 1. Although this is uncommon in C++, it is customary to number function arguments starting at 1.

There are two ways to specify the index for GetParameter. The first is to directly specify the template parameter (e.g. GetParameter<1>()). However, note that in a templated function or method where the type is not fully resolved the compiler will not register GetParameter as a templated method and will fail to parse the template argument without a template keyword. The second way to specify the index is to provide a vtkm::internal::IndexTag object as an argument to GetParameter. Although this syntax is more verbose, it works the same whether the FunctionInterface is fully resolved or not. The following example shows both methods in action.

Example 39.4: Using FunctionInterface::GetParameter().

```
void GetFirstParameterResolved(
  const vtkm::internal::FunctionInterface<void(std::string)>& interface)
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>()) << std::endl;
}

template<typename FunctionSignature>
void GetFirstParameterTemplated(
  const vtkm::internal::FunctionInterface<FunctionSignature>& interface)
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.template GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>()) << std::endl;
}
```

Likewise, there is a SetParmeter method for changing parameters. The same rules for indexing and template specification apply.

Example 39.5: Using FunctionInterface::SetParameter().

```
void SetFirstParameterResolved(
  vtkm::internal::FunctionInterface<void(std::string)>& interface,
  const std::string& newFirstParameter)
{
  // The following two uses of SetParameter are equivalent
  interface.SetParameter<1>(newFirstParameter);
  interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
}

template<typename FunctionSignature, typename T>
void SetFirstParameterTemplated(
  vtkm::internal::FunctionInterface<FunctionSignature>& interface,
  T newFirstParameter)
{
  // The following two uses of SetParameter are equivalent
  interface.template SetParameter<1>(newFirstParameter);
  interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
}
```

39.3 Invoking

FunctionInterface can invoke a functor of a matching signature using the parameters stored within. If the functor returns a value, that return value will be stored in the FunctionInterface object for later retrieval. There are several versions of the invoke method. There are always seperate versions of invoke methods for the control and execution environments so that functors for either environment can be executed. The basic version of invoke passes the parameters directly to the function and directly stores the result.

Example 39.6: Invoking a FunctionInterface.

```
  vtkm::internal::FunctionInterface<size_t(const char*)> strlenInterface;
  strlenInterface.SetParameter<1>("Hello world");

  strlenInterface.InvokeCont(strlen);

  size_t length = strlenInterface.GetReturnValue(); // length = 11
```

Another form of the invoke methods takes a second transform functor that is applied to each argument before passed to the main function. If the main function returns a value, the transform is applied to that as well before being stored back in the FunctionInterface.

Example 39.7: Invoking a FunctionInterface with a transform.

```
// Our transform converts C strings to integers, leaves everything else alone.
struct TransformFunctor
{
  template<typename T>
  VTKM_CONT const T& operator()(const T& x) const
  {
    return x;
  }

  VTKM_CONT
  vtkm::Int32 operator()(const char* x) const { return atoi(x); }
};

// The function we are invoking simply compares two numbers.
struct IsSameFunctor
{
  template<typename T1, typename T2>
  VTKM_CONT bool operator()(const T1& x, const T2& y) const
  {
    return x == y;
  }
};

void TryTransformedInvoke()
{
  vtkm::internal::FunctionInterface<bool(const char*, vtkm::Int32)>
    functionInterface = vtkm::internal::make_FunctionInterface<bool>(
      (const char*)"42", (vtkm::Int32)42);

  functionInterface.InvokeCont(IsSameFunctor(), TransformFunctor());

  bool isSame = functionInterface.GetReturnValue(); // isSame = true
}
```

As demonstrated in the previous examples, FunctionInterface has a method named GetReturnValue that returns the value from the last invoke. Care should be taken to only use GetReturnValue when the function specification has a return value. If the function signature has a void return type, using GetReturnValue will cause a compile error.

FunctionInterface has an alternate method named GetReturnValueSafe that returns the value wrapped in a templated structure named vtkm::internal::FunctionInterfaceReturnContainer. This structure always has a static constant Boolean named VALID that is false if there is no return type and true otherwise. If the container is valid, it also has an entry named Value containing the result.

Example 39.8: Getting return value from FunctionInterface safely.

```
template<typename ResultType, bool Valid>
struct PrintReturnFunctor;

template<typename ResultType>
struct PrintReturnFunctor<ResultType, true>
{
  VTKM_CONT
  void operator()(
    const vtkm::internal::FunctionInterfaceReturnContainer<ResultType>& x) const
  {
    std::cout << x.Value << std::endl;
  }
};

template<typename ResultType>
struct PrintReturnFunctor<ResultType, false>
{
  VTKM_CONT
  void operator()(
    const vtkm::internal::FunctionInterfaceReturnContainer<ResultType>&) const
  {
    std::cout << "No return type." << std::endl;
  }
};

template<typename FunctionInterfaceType>
void PrintReturn(const FunctionInterfaceType& functionInterface)
{
  using ResultType = typename FunctionInterfaceType::ResultType;
  using ReturnContainerType =
    vtkm::internal::FunctionInterfaceReturnContainer<ResultType>;

  PrintReturnFunctor<ResultType, ReturnContainerType::VALID> printReturn;
  printReturn(functionInterface.GetReturnValueSafe());
}
```

39.4 Modifying Parameters

In addition to storing and querying parameters and invoking functions, FunctionInterface also contains multiple ways to modify the parameters to augment the function calls. This can be used in the same use case as a chain of function calls that generally pass their parameters but also augment the data along the way.

The Append method returns a new FunctionInterface object with the same parameters plus a new parameter (the argument to Append) to the end of the parameters. There is also a matching AppendType templated structure that can return the type of an augmented FunctionInterface with a new type appended.

Example 39.9: Appending parameters to a FunctionInterface.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

using InitialFunctionInterfaceType =
  FunctionInterface<void(std::string, vtkm::Id)>;
InitialFunctionInterfaceType initialFunctionInterface =
  make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

using AppendedFunctionInterfaceType1 =
  FunctionInterface<void(std::string, vtkm::Id, std::string)>;
AppendedFunctionInterfaceType1 appendedFunctionInterface1 =
  initialFunctionInterface.Append(std::string("foobar"));
// appendedFunctionInterface1 has parameters ("Hello World", 42, "foobar")

using AppendedFunctionInterfaceType2 =
  InitialFunctionInterfaceType::AppendType<vtkm::Float32>::type;
AppendedFunctionInterfaceType2 appendedFunctionInterface2 =
  initialFunctionInterface.Append(vtkm::Float32(3.141));
// appendedFunctionInterface2 has parameters ("Hello World", 42, 3.141)
```

`Replace` is a similar method that returns a new `FunctionInterface` object with the same paraemters except with a specified parameter replaced with a new parameter (the argument to `Replace`). There is also a matching `ReplaceType` templated structure that can return the type of an augmented `FunctionInterface` with one of the parameters replaced.

Example 39.10: Replacing parameters in a `FunctionInterface`.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

using InitialFunctionInterfaceType =
  FunctionInterface<void(std::string, vtkm::Id)>;
InitialFunctionInterfaceType initialFunctionInterface =
  make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

using ReplacedFunctionInterfaceType1 =
  FunctionInterface<void(vtkm::Float32, vtkm::Id)>;
ReplacedFunctionInterfaceType1 replacedFunctionInterface1 =
  initialFunctionInterface.Replace<1>(vtkm::Float32(3.141));
// replacedFunctionInterface1 has parameters (3.141, 42)

using ReplacedFunctionInterfaceType2 =
  InitialFunctionInterfaceType::ReplaceType<2, std::string>::type;
ReplacedFunctionInterfaceType2 replacedFunctionInterface2 =
  initialFunctionInterface.Replace<2>(std::string("foobar"));
// replacedFunctionInterface2 has parameters ("Hello World", "foobar")
```

It is sometimes desirable to make multiple modifications at a time. This can be achieved by chaining modifications by calling `Append` or `Replace` on the result of a previous call.

Example 39.11: Chaining `Replace` and `Append` with a `FunctionInterface`.

```
template<typename FunctionInterfaceType>
void FunctionCallChain(const FunctionInterfaceType& parameters, vtkm::Id arraySize)
{
  // In this hypothetical function call chain, this function replaces the
  // first parameter with an array of that type and appends the array size
  // to the end of the parameters.

  using ArrayValueType =
    typename FunctionInterfaceType::template ParameterType<1>::type;

  // Allocate and initialize array.
  ArrayValueType value = parameters.template GetParameter<1>();
  ArrayValueType* array = new ArrayValueType[arraySize];
  for (vtkm::Id index = 0; index < arraySize; index++)
  {
    array[index] = value;
  }

  // Call next function with modified parameters.
  NextFunctionChainCall(parameters.template Replace<1>(array).Append(arraySize));

  // Clean up.
  delete[] array;
}
```

39.5 Transformations

Rather than replace a single item in a FunctionInterface, it is sometimes desirable to change them all in a similar way. FunctionInterface supports two basic transform operations on its parameters: a static transform and a dynamic transform. The static transform determines its types at compile-time whereas the dynamic transform happens at run-time.

The static transform methods (named StaticTransformCont and StaticTransformExec) operate by accepting a functor that defines a function with two arguments. The first argument is the FunctionInterface parameter to transform. The second argument is an instance of the vtkm::internal::IndexTag templated class that statically identifies the parameter index being transformed. An IndexTag object has no state, but the class contains a static integer named INDEX. The function returns the transformed argument.

The functor must also contain a templated class named ReturnType with an internal type named type that defines the return type of the transform for a given parameter type. ReturnType must have two template parameters. The first template parameter is the type of the FunctionInterface parameter to transform. It is the same type as passed to the operator. The second template parameter is a vtkm::IdComponent specifying the index.

The transformation is only applied to the parameters of the function. The return argument is unaffected.

The return type can be determined with the StaticTransformType template in the FunctionInterface class. StaticTransformType has a single parameter that is the transform functor and contains a type named type that is the transformed FunctionInterface.

In the following example, a static transform is used to convert a FunctionInterface to a new object that has the pointers to the parameters rather than the values themselves. The parameter index is always ignored as all parameters are uniformly transformed.

Example 39.12: Using a static transform of function interface class.

```
struct ParametersToPointersFunctor
{
  template<typename T, vtkm::IdComponent Index>
  struct ReturnType
  {
    using type = const T*;
  };

  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT const T* operator()(const T& x, vtkm::internal::IndexTag<Index>) const
  {
    return &x;
  }
};
```

```

template<typename FunctionInterfaceType>
VTKM_CONT typename FunctionInterfaceType::template StaticTransformType<
  ParametersToPointersFunctor>::type
ParametersToPointers(FunctionInterfaceType& functionInterface)
{
  return functionInterface.StaticTransformCont(ParametersToPointersFunctor());
}
```

There are cases where one set of parameters must be transformed to another set, but the types of the new set are not known until run-time. That is, the transformed type depends on the contents of the data. The `FunctionInterface::DynamicTransformCont` method achieves this using a templated callback that gets called with the correct type at run-time.

The dynamic transform works with two functors provided by the user code (as opposed to the one functor in static transform). These functors are called the transform functor and the finish functor. The transform functor accepts three arguments. The first argument is a parameter to transform. The second argument is a continue function. Rather than return the transformed value, the transform functor calls the continue function, passing the transformed value as an argument. The third argument is a `vtkm::internal::IndexTag` for the index of the argument being transformed.

Unlike its static counterpart, the dynamic transform method does not return the transformed `FunctionInterface`. Instead, it passes the transformed `FunctionInterface` to the finish functor passed into `DynamicTransformCont`.

In the following contrived but illustrative example, a dynamic transform is used to convert strings containing numbers into number arguments. Strings that do not have numbers and all other arguments are passed through. Note that because the types for strings are not determined till run-time, this transform cannot be determined at compile time with meta-template programming. The index argument is ignored because all arguments are transformed the same way.

Example 39.13: Using a dynamic transform of a function interface.

```
struct UnpackNumbersTransformFunctor
{
  template<typename InputType, typename ContinueFunctor, vtkm::IdComponent Index>
  VTKM_CONT void operator()(const InputType& input,
                            const ContinueFunctor& continueFunction,
                            vtkm::internal::IndexTag<Index>) const
  {
    continueFunction(input);
  }

  template<typename ContinueFunctor, vtkm::IdComponent Index>
  VTKM_CONT void operator()(const std::string& input,
                            const ContinueFunctor& continueFunction,
                            vtkm::internal::IndexTag<Index>) const
  {
    if ((input[0] >= '0') && (input[0] <= '9'))
    {
      std::stringstream stream(input);
      vtkm::FloatDefault value;
      stream >> value;
      continueFunction(value);
    }
    else
    {
      continueFunction(input);
    }
  }
};
```

```

struct UnpackNumbersFinishFunctor
{
  template<typename FunctionInterfaceType>
  VTKM_CONT void operator()(FunctionInterfaceType& functionInterface) const
  {
    // Do something
  }
};

template<typename FunctionInterfaceType>
void DoUnpackNumbers(const FunctionInterfaceType& functionInterface)
{
  functionInterface.DynamicTransformCont(UnpackNumbersTransformFunctor(),
                                         UnpackNumbersFinishFunctor());
}
```

One common use for the FunctionInterface dynamic transform is to convert parameters of virtual polymorphic type like vtkm::cont::VariantArrayHandle and vtkm::cont::DynamicPointCoordinates. This use case is handled with a functor named vtkm::cont::internal::DynamicTransform. When used as the dynamic transform functor, it will convert all of these dynamic types to their static counterparts.

Example 39.14: Using DynamicTransform to cast dynamic arrays in a function interface.

```
template<typename Device>
struct ArrayCopyFunctor
{
  template<typename Signature>
  VTKM_CONT void operator()(
    vtkm::internal::FunctionInterface<Signature> functionInterface) const
  {
    functionInterface.InvokeCont(*this);
  }

  template<typename T, typename SIn, typename SOut>
  VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<T, SIn>& input,
                            vtkm::cont::ArrayHandle<T, SOut>& output) const
  {
    vtkm::cont::Algorithm::Copy(input, output);
  }

  template<typename TIn, typename SIn, typename TOut, typename SOut>
  VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<TIn, SIn>&,
                            vtkm::cont::ArrayHandle<TOut, SOut>&) const
  {
    throw vtkm::cont::ErrorBadType("Arrays to copy must be the same type.");
  }
};

template<typename Device>
void CopyVariantArrays(vtkm::cont::VariantArrayHandle input,
                       vtkm::cont::VariantArrayHandle output,
                       Device)
{
  vtkm::internal::FunctionInterface<void(vtkm::cont::VariantArrayHandle,
                                         vtkm::cont::VariantArrayHandle)>
    functionInterface = vtkm::internal::make_FunctionInterface<void>(input, output);

  functionInterface.DynamicTransformCont(vtkm::cont::internal::DynamicTransform(),
                                         ArrayCopyFunctor<Device>());
}
```

39.6 For Each

The invoke methods (principally) make a single function call passing all of the parameters to this function. The transform methods call a function on each parameter to convert it to some other data type. It is also sometimes helpful to be able to call a unary function on each parameter that is not expected to return a value. Typically the use case is for the function to have some sort of side effect. For example, the function might print out some value (such as in the following example) or perform some check on the data and throw an exception on failure.

This feature is implemented in the for each methods of `FunctionInterface`. As with all the `FunctionInterface` methods that take functors, there are separate implementations for the control environment and the execution environment. There are also separate implementations taking `const` and non-`const` references to functors to simplify making functors with side effects.

Example 39.15: Using the `ForEach` feature of `FunctionInterface`.

```
struct PrintArgumentFunctor
{
  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT void operator()(const T& argument, vtkm::internal::IndexTag<Index>) const
  {
    std::cout << Index << ":" << argument << " ";
  }
};

template<typename FunctionInterfaceType>
VTKM_CONT void PrintArguments(const FunctionInterfaceType& functionInterface)
{
  std::cout << "( ";
  functionInterface.ForEachCont(PrintArgumentFunctor());
  std::cout << ")" << std::endl;
}
```

CHAPTER

FORTY

WORKLET ARGUMENTS

From the ControlSignature and ExecutionSignature defined in worklets, VTK-m uses template metaprogramming to build the code required to manage data from the control to the execution environment. These signatures contain tags that define the meaning of each argument and control how the argument data are transferred from the control to execution environments and broken up for each worklet instance.

Chapter 21 documents the many ControlSignature and ExecutionSignature tags that come with the worklet types. This chapter discusses the internals of these tags and how they control data management. Defining new worklet argument types can allow you to define new data structures in VTK-m. New worklet arguments are also usually a critical components for making new worklet types, as described in Chapter 41.

The management of data in worklet arguments is handled by three classes that provide type checking, transportation, and fetching, respectively. This chapter will first describe these type checking, transportation, and fetching classes and then describe how ControlSignature and ExecutionSignature tags specify these classes.

Throughout this chapter we demonstrate the definition of worklet arguments using an example of a worklet argument that represents line segments in 2D. The input for such an argument expects an ArrayHandle containing floating point vtkm::Vec s of size 2 to represent coordinates in the plane. The values in the array are paired up to define the two endpoints of each segment, and the worklet instance will receive a Vec-2 of Vec-2's representing the two endpoints. In practice, it is generally easier to use a vtkm::cont::ArrayHandleGroupVec (see Section 26.12), but this is a simple example for demonstration purposes. Plus, we will use this special worklet argument for our example of a custom worklet type in Chapter 41.

40.1 Type Checks

Before attempting to move data from the control to the execution environment, the VTK-m invokers check the input types to ensure that they are compatible with the associated ControlSignature concept. This is done with the vtkm::cont::arg::TypeCheck struct.

The TypeCheck struct is templated with two parameters. The first parameter is a tag that identifies which check to perform. The second parameter is the type of the control argument (after any dynamic casts). The TypeCheck class contains a static constant Boolean named value that is true if the type in the second parameter is compatible with the tag in the first or false otherwise.

Type checks are implemented with a defined type check tag (which, by convention, is defined in the vtkm::cont::arg namespace and starts with TypeCheckTag) and a partial specialization of the vtkm::cont::arg::TypeCheck structure. The following type checks (identified by their tags) are provided in VTK-m.

vtkm::cont::arg::TypeCheckTagExecObject True if the type is an execution object. All execution objects must derive from vtkm::cont::ExecutionObjectBase and follow the conventions of that class.

vtkm::cont::arg::TypeCheckTagArray True if the type is a vtkm::cont::ArrayHandle.

vtkm::cont::arg::TypeCheckTagAtomicArray Similar to TypeCheckTagArray except it only returns true for array types with values that are supported for atomic arrays.

vtkm::cont::arg::TypeCheckTagCellSet True if and only if the object is a vtkm::cont::CellSet or one of its subclasses.

vtkm::cont::arg::TypeCheckTagKeys True if and only if the object is a vtkm::worklet::Keys class.

Here are some trivial examples of using TypeCheck. Typically these checks are done internally in the base VTK-m invoker code, so these examples are for demonstration only.

Example 40.1: Behavior of vtkm::cont::arg::TypeCheck.

```
struct MyExecObject : vtkm::cont::ExecutionObjectBase
{
  vtkm::Id Value;
};

void DoTypeChecks()
{
  using vtkm::cont::arg::TypeCheck;
  using vtkm::cont::arg::TypeCheckTagArray;
  using vtkm::cont::arg::TypeCheckTagExecObject;

  bool check1 = TypeCheck<TypeCheckTagExecObject, MyExecObject>::value; // true
  bool check2 = TypeCheck<TypeCheckTagExecObject, vtkm::Id>::value;     // false

  using ArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;

  bool check3 = TypeCheck<TypeCheckTagArray, ArrayType>::value;      // true
  bool check4 = TypeCheck<TypeCheckTagExecObject, ArrayType>::value; // false
}
```

A type check is created by first defining a type check tag object, which by convention is placed in the vtkm::cont::arg namespace and whose name starts with TypeCheckTag. Then, create a specialization of the vtkm::cont::arg::TypeCheck template class with the first template argument matching the aforementioned tag. As stated previously, the TypeCheck class must contain a value static constant Boolean representing whether the type is acceptable for the corresponding Invoker argument.

This example of a TypeCheck returns true for control objects that are ArrayHandles with a value type that is a floating point vtkm::Vec of size 2.

Example 40.2: Defining a custom TypeCheck.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

struct TypeCheckTag2DCoordinates
{
};

template<typename ArrayType>
struct TypeCheck<TypeCheckTag2DCoordinates, ArrayType>
{
  static constexpr bool value = false;
};

template<typename T, typename Storage>
struct TypeCheck<TypeCheckTag2DCoordinates, vtkm::cont::ArrayHandle<T, Storage>>
{
  static constexpr bool value =
    vtkm::ListContains<vtkm::TypeListTagFieldVec2, T>::value;
};

} // namespace arg
} // namespace cont
} // namespace vtkm
```

Did you know?

The type check defined in Example 40.2 could actually be replaced by the more general TypeCheckTagArray *that already comes with VTK-m. This example is mostly provided for demonstrative purposes. In practice, it is often useful to use* std::is_same *or* std::is_base_of*, which are provided by the standard template library starting with C++11, to determine* value *in a* TypeCheck*.*

40.2 Transport

After all the argument types are checked, the VTK-m dispatch mechanism must load the data into the execution environment before scheduling a job to run there. This is done with the vtkm::cont::arg::Transport struct.

The Transport struct is templated with three parameters. The first parameter is a tag that identifies which transport to perform. The second parameter is the type of the control parameter (after any dynamic casts). The third parameter is a device adapter tag for the device on which the data will be loaded.

A Transport contains a type named ExecObjectType that is the type used after data is moved to the execution environment. A Transport also has a const parenthesis operator that takes 4 arguments: the control-side object that is to be transported to the execution environment, the control-side object that represents the input domain, the size of the input domain, and the size of the output domain and returns an execution-side object. This operator is called in the control environment, and the operator returns an object that is ready to be used in the execution environment.

Transports are implemented with a defined transport tag (which, by convention, is defined in the vtkm::cont::arg namespace and starts with TransportTag) and a partial specialization of the vtkm::cont::arg::Transport structure. The following transports (identified by their tags) are provided in VTK-m.

vtkm::cont::arg::TransportTagExecObject Simply returns the given execution object, which should be ready to load onto the device.

vtkm::cont::arg::TransportTagArrayIn Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInput method. The size of the array must be the same as the input domain. The returned execution object is an array portal.

vtkm::cont::arg::TransportTagArrayOut Allocates data onto the specified device for a vtkm::cont::ArrayHandle using the array handle's PrepareForOutput method. The array is allocated to the size of the output domain. The returned execution object is an array portal.

vtkm::cont::arg::TransportTagArrayInOut Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInPlace method. The size of the array must be the same size as the output domain (which is not necessarily the same size as the input domain). The returned execution object is an array portal.

vtkm::cont::arg::TransportTagWholeArrayIn Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInput method. This transport is designed to be used with random access whole arrays, so unlike TransportTagArrayIn the array size can be unassociated with the input domain. The returned execution object is an array portal.

vtkm::cont::arg::TransportTagWholeArrayOut Readies data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForOutput method. This transport is designed to be used with random access whole arrays, so unlike TransportTagArrayOut the array size can be unassociated with the input domain. Thus, the array must be pre-allocated and its size is not changed. The returned execution object is an array portal.

vtkm::cont::arg::TransportTagWholeArrayInOut Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInPlace method. This transport is designed to be used with random access whole arrays, so unlike TransportTagArrayInOut the array size can be unassociated with the input domain. The returned execution object is an array portal.

vtkm::cont::arg::TransportTagAtomicArray Loads data from a vtkm::cont::ArrayHandle and creates a vtkm::exec::AtomicArray.

vtkm::cont::arg::TransportTagCellSetIn Loads data from a vtkm::cont::CellSet object. The TransportTagCellSetIn it a templated class with two parameters: the "from" topology and the "to" topology. (See Section 21.2.3 for a description of "from" and "to" topologies.) The returned execution object is a connectivity object (as described in Section 28.3).

vtkm::cont::arg::TransportTagTopologyFieldIn Similar to TransportTagArrayIn except that the size is checked against the "from" topology of a cell set for the input domain. The input domain object is assumed to be a vtkm::cont::CellSet.

vtkm::cont::arg::TransportTagKeysIn Loads data from a vtkm::worklet::Keys object. This transport is intended to be used for the input domain of a vtkm::worklet::WorkletReduceByKey. The returned execution object is of type vtkm::exec::internal::ReduceByKeyLookup.

vtkm::cont::arg::TransportTagKeyedValuesIn Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInput method. This transport uses the input domain object, which is expected to be a vtkm::worklet::Keys object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

vtkm::cont::arg::TransportTagKeyedValuesOut Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForOutput method. This transport uses the input domain object, which is expected to be a vtkm::worklet::Keys object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

vtkm::cont::arg::TransportTagKeyedValuesInOut Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInPlace method. This transport uses the input domain object, which is expected to be a vtkm::worklet::Keys object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

Here are some trivial examples of using Transport. Typically this movement is done internally in the VTK-m dispatching code, so these examples are for demonstration only.

Example 40.3: Behavior of vtkm::cont::arg::Transport.

```
template<typename ArrayType, typename Device>
void DoTransport(ArrayType inArray, ArrayType outArray, Device)
{
  VTKM_IS_ARRAY_HANDLE(ArrayType);
  VTKM_IS_DEVICE_ADAPTER_TAG(Device);

  using vtkm::cont::arg::Transport;
  using vtkm::cont::arg::TransportTagArrayIn;
  using vtkm::cont::arg::TransportTagArrayOut;
  using vtkm::cont::arg::TransportTagWholeArrayInOut;

  // The array in transport returns a read-only array portal.
  using ArrayInTransport = Transport<TransportTagArrayIn, ArrayType, Device>;
  typename ArrayInTransport::ExecObjectType inPortal =
    ArrayInTransport()(inArray, inArray, 10, 10);

  // The array out transport returns an allocated array portal.
  using ArrayOutTransport = Transport<TransportTagArrayOut, ArrayType, Device>;
  typename ArrayOutTransport::ExecObjectType outPortal =
    ArrayOutTransport()(outArray, inArray, 10, 10);

  // The whole array in transport returns a read-only array portal wrapped in
  // a vtkm::exec::ExecutionWholeArrayConst.
  using WholeArrayTransport =
    Transport<TransportTagWholeArrayInOut, ArrayType, Device>;
  typename WholeArrayTransport::ExecObjectType wholeArray =
    WholeArrayTransport()(inArray, inArray, 10, 10);
}
```

A transport is created by first defining a transport tag object, which by convention is placed in the vtkm::cont::arg namespace and whose name starts with TransportTag. Then, create a specialization of the vtkm::cont::arg::Transport template class with the first template argument matching the aforementioned tag. As stated previously, the Transport class must contain an ExecObjectType type and a parenthesis operator turning the associated control argument into an execution environment object.

This example internally uses a vtkm::cont::ArrayHandleGroupVec to take values from an input ArrayHandle and pair them up to represent line segments. The resulting execution object is an array portal containing Vec-2 values of Vec-2's.

Example 40.4: Defining a custom Transport.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

struct TransportTag2DLineSegmentsIn
{
};

template<typename ContObjectType, typename Device>
struct Transport<vtkm::cont::arg::TransportTag2DLineSegmentsIn,
                 ContObjectType,
                 Device>
{
  VTKM_IS_ARRAY_HANDLE(ContObjectType);

  using GroupedArrayType = vtkm::cont::ArrayHandleGroupVec<ContObjectType, 2>;

  using ExecObjectType =
22      typename GroupedArrayType::template ExecutionTypes<Device>::PortalConst;
23
24    template<typename InputDomainType>
25    VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
26                                        const InputDomainType&,
27                                        vtkm::Id inputRange,
28                                        vtkm::Id) const
29    {
30      if (object.GetNumberOfValues() != inputRange * 2)
31      {
32        throw vtkm::cont::ErrorBadValue(
33          "2D line segment array size does not agree with input size.");
34      }
35
36      GroupedArrayType groupedArray(object);
37      return groupedArray.PrepareForInput(Device());
38    }
39  };
40
41  } // namespace arg
42  } // namespace cont
43  } // namespace vtkm
```

Common Errors

It is fair to assume that the Transport*'s control object type matches whatever the associated* TypeCheck *allows. However, it is good practice to provide a secondary compile-time check in the* Transport *class, like the one on line 17 in Example 40.4, for debugging purposes in case there is a problem with the* TypeCheck *or this* Transport *is used with an unexpected* TypeCheck*.*

40.3 Fetch

Before the function of a worklet is invoked, the VTK-m internals pull the appropriate data out of the execution object and pass it to the worklet function. A class named vtkm::exec::arg::Fetch is responsible for pulling this data out and putting computed data in to the execution objects.

The Fetch struct is templated with four parameters. The first parameter is a tag that identifies which type of fetch to perform. The second parameter is a different tag that identifies the aspect of the data to fetch.

The third template parameter to a Fetch struct is a type of thread indices object, which manages the indices and other metadata associated with the thread for which the Fetch operator gets called. The specific type of the thread indices object depends on the type of worklet begin invoked, but all thread indices classes implement methods named GetInputIndex, GetOutputIndex, and GetVisitIndex to get those respective indices. The thread indices object may also contain other methods to get information pertinent to the associated worklet's execution. For example a thread indices object associated with a topology map has methods to get the shape identifier and incident from indices of the current input object. Thread indices objects are discussed in more detail in Section 41.2.

The fourth template parameter to a Fetch struct is the type of the execution object that is created by the Transport (as described in Section 40.2). This is generally where the data are fetched from.

A Fetch contains a type named ValueType that is the type of data that is passed to and from the worklet function. A Fetch also has a pair of methods named Load and Store that get data from and add data to the execution object at a given domain or thread index.

Fetches are specified with a pair of fetch and aspect tags. Fetch tags are by convention defined in the vtkm::exec::arg namespace and start with FetchTag. Likewise, aspect tags are also defined in the vtkm::exec::arg namespace and start with AspectTag. The Fetch class is partially specialized on these two tags.

The most common aspect tag is vtkm::exec::arg::AspectTagDefault, and all fetch tags should have a specialization of vtkm::exec::arg::Fetch with this tag. The following list of fetch tags describes the execution objects they work with and the data they pull for each aspect tag they support.

vtkm::exec::arg::FetchTagExecObject Simply returns an execution object. This fetch only supports the AspectTagDefault aspect. The Load returns the executive object in the associated parameter. The Store does nothing.

vtkm::exec::arg::FetchTagWholeCellSetIn Loads data from a cell set. The Load simply returns the execution object created with a TransportTagCellSetIn and the Store does nothing.

vtkm::exec::arg::FetchTagArrayDirectIn Loads data from an array portal. This fetch only supports the AspectTagDefault aspect. The Load gets data directly from the domain (thread) index. The Store does nothing.

vtkm::exec::arg::FetchTagArrayDirectOut Stores data to an array portal. This fetch only supports the AspectTagDefault aspect. The Store sets data directly to the domain (thread) index. The Load does nothing.

vtkm::exec::arg::FetchTagCellSetIn Load data from a cell set. This fetch is used with the worklet topology maps to pull topology information from a cell set. The Load simply returns the cell shape of the given input cells and the Store method does nothing. This tag is typically used with the input domain object, and aspects like vtkm::exec::arg::AspectTagIncidentElementCount and vtkm::exec::arg::AspectTagIncidentElementIndices are used to get more detailed information.

vtkm::exec::arg::FetchTagArrayTopologyMapIn Loads data from the "from" topology in a topology map. For example, in a point to cell topology map, this fetch will get the field values for all points attached to the cell being visited. The Load returns a Vec-like object containing all the incident field values whereas the Store method does nothing. This fetch is designed for use in topology maps and expects the input domain to be a cell set.

A fetch is created by first defining a fetch tag object, which by convention is placed in the vtkm::exec::arg namespace and whose name starts with FetchTag. Then, create a specialization of the vtkm::exec::arg::Fetch template class with the first template argument matching the aforementioned tag. As stated previously, the Fetch class must contain a ValueType type and a pair of Load and Store methods that get a value out of the data and store a value in the data, respectively.

Example 40.5: Defining a custom Fetch.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct FetchTag2DLineSegmentsIn
{
};

template<typename ThreadIndicesType, typename ExecObjectType>
struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
             vtkm::exec::arg::AspectTagDefault,
             ThreadIndicesType,
             ExecObjectType>
{
  using ValueType = typename ExecObjectType::ValueType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC
  ValueType Load(const ThreadIndicesType& indices,
                 const ExecObjectType& arrayPortal) const
  {
    return arrayPortal.Get(indices.GetInputIndex());
  }

  VTKM_EXEC
  void Store(const ThreadIndicesType&, const ExecObjectType&, const ValueType&) const
  {
    // Store is a no-op for this fetch.
  }
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

Did you know?

The fetch defined in Example 40.5 could actually be replaced by the more general `FetchTagArrayDirectIn` *that already comes with VTK-m. This example is mostly provided for demonstrative purposes.*

In addition to the aforementioned aspect tags that are explicitly paired with fetch tags, VTK-m also provides some aspect tags that either modify the behavior of a general fetch or simply ignore the type of fetch.

`vtkm::exec::arg::AspectTagDefault` Performs the "default" fetch. Every fetch tag should have an implementation of `vtkm::exec::arg::Fetch` with that tag and `AspectTagDefault`.

`vtkm::exec::arg::AspectTagWorkIndex` Simply returns the domain (or thread) index ignoring any associated data. This aspect is used to implement the `WorkIndex` execution signature tag.

`vtkm::exec::arg::AspectTagInputIndex` Returns the index of the element being used from the input domain. This is often the same as the work index but can be different if a scatter is being used. (See Section 31.1 for information on scatters in worklets.)

`vtkm::exec::arg::AspectTagOutputIndex` Returns the index of the element being written to the output. This is generally the same as the work index.

`vtkm::exec::arg::AspectTagVisitIndex` Returns the visit index corresponding to the current input. Together the pair of input index and visit index are unique.

`vtkm::exec::arg::AspectTagCellShape` Returns the cell shape from the input domain. This aspect is designed to be used with topology maps.

`vtkm::exec::arg::AspectTagIncidentElementCount` Returns the number of elements associated with the "from" topology that are incident to the input element of the "to" topology. This aspect is designed to be used with topology maps.

vtkm::exec::arg::AspectTagIncidentElementIndices Returns a Vec-like object containing the indices to the elements associated with the “from” topology that are incident to the input element of the “to” topology. This aspect is designed to be used with topology maps.

vtkm::exec::arg::AspectTagValueCount Returns the number of times the key associated with the current input. This aspect is designed to be used with reduce by key maps.

An aspect is created by first defining an aspect tag object, which by convention is placed in the vtkm::exec::arg namespace and whose name starts with AspectTag. Then, create specializations of the vtkm::exec::arg::Fetch template class where appropriate with the second template argument matching the aforementioned tag.

This example creates a specialization of a Fetch to retrieve the first point of a line segment.

Example 40.6: Defining a custom Aspect.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct AspectTagFirstPoint
{
};

template<typename ThreadIndicesType, typename ExecObjectType>
struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
             vtkm::exec::arg::AspectTagFirstPoint,
             ThreadIndicesType,
             ExecObjectType>
{
  using ValueType = typename ExecObjectType::ValueType::ComponentType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC
  ValueType Load(const ThreadIndicesType& indices,
                 const ExecObjectType& arrayPortal) const
  {
    return arrayPortal.Get(indices.GetInputIndex())[0];
  }

  VTKM_EXEC
  void Store(const ThreadIndicesType&, const ExecObjectType&, const ValueType&) const
  {
    // Store is a no-op for this fetch.
  }
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

40.4 Creating New ControlSignature Tags

The type checks, transports, and fetches defined in the previous sections of this chapter conspire to interpret the arguments given to a Invoker and provide data to an instance of a worklet. What remains to be defined are the tags used in the ControlSignature and ExecutionSignature that bring these three items together. These two types of tags are defined differently. In this section we discuss the ControlSignature tags.

A ControlSignature tag is defined by a struct (or equivocally a class). This struct is typically defined inside a worklet (or, more typically, a worklet superclass) so that it can be used without qualifying its namespace. VTK-m has requirements for every defined ControlSignature tag.

The first requirement of a ControlSignature tag is that it must inherit from vtkm::cont::arg::ControlSignatureTagBase. You will get a compile error if you attempt to use a type that is not a subclass of ControlSignatureTagBase in a ControlSignature.

The second requirement of a ControlSignature tag is that it must contain the following three types: TypeCheckTag, TransportTag, and FetchTag. As the names would imply, these specify tags for TypeCheck, Transport, and Fetch classes, respectively, which were discussed earlier in this chapter.

The following example defines a ControlSignature tag for an array that represents 2D line segments using the classes defined in previous examples.

Example 40.7: Defining a new ControlSignature tag.

```
struct LineSegment2DCoordinatesIn : vtkm::cont::arg::ControlSignatureTagBase
{
  using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
  using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
  using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
};
```

Once defined, this tag can be used like any other ControlSignature tag.

Example 40.8: Using a custom ControlSignature tag.

```
using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
                              FieldOut vecOut,
                              FieldIn index);
```

40.5 Creating New ExecutionSignature Tags

An ExecutionSignature tag is defined by a struct (or equivocally a class). This struct is typically defined inside a worklet (or, more typically, a worklet superclass) so that it can be used without qualifying its namespace. VTK-m has requirements for every defined ExecutionSignature tag.

The first requirement of an ExecutionSignature tag is that it must inherit from vtkm::exec::arg::ExecutionSignatureTagBase. You will get a compile error if you attempt to use a type that is not a subclass of ExecutionSignatureTagBase in an ExecutionSignature.

The second requirement of an ExecutionSignature tag is that it must contain a type named AspectTag, which is set to an aspect tag. As discussed in Section 40.3, the aspect tag is passed as a template argument to the vtkm::exec::arg::Fetch class to modify the data it loads and stores. The numerical ExecutionSignature tags (i.e. _1, _2, etc.) operate by setting the AspectTag to vtkm::exec::arg::AspectTagDefault, effectively engaging the default fetch.

The third requirement of an ExecutionSignature tag is that it contains an INDEX member that is a static const vtkm::IdComponent. The number that INDEX is set to refers to the ControlSignature argument from which that data come from (indexed starting at 1). The numerical ExecutionSignature tags (i.e. _1, _2, etc.) operate by setting their INDEX values to the corresponding number (i.e. 1, 2, etc.). An ExecutionSignature tag might take another tag as a template argument and copy the INDEX from one to another. This allows you to use a tag to modify the aspect of another tag. Most often this is used to apply a particular aspect to a numerical ExecutionSignature tag (i.e. _1, _2, etc.). Still other ExecutionSignature tags might not need direct access to any ControlSignature arguments (such as those that pull information from thread indices). If the INDEX does not matter (because the execution object parameter to the Fetch Load and Store is ignored). In this case, the ExecutionSignature tag can set the INDEX to 1, because there is guaranteed to be at least one control argument.

The following example defines an ExecutionSignature tag to get the coordinates for only the first point in a 2D line segment. The defined tag takes as an argument another tag (generally one of the numeric tags), which is expected to point to a ControlSignature argument with a LineSegment2DCoordinatesIn (as defined in Example 40.7).

Example 40.9: Defining a new ExecutionSignature tag.

```
template<typename ArgTag>
struct FirstPoint : vtkm::exec::arg::ExecutionSignatureTagBase
{
  static const vtkm::IdComponent INDEX = ArgTag::INDEX;
  using AspectTag = vtkm::exec::arg::AspectTagFirstPoint;
};
```

Once defined, this tag can be used like any other ExecutionSignature tag.

Example 40.10: Using a custom ExecutionSignature tag.

```
using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
                              FieldOut vecOut,
                              FieldIn index);
using ExecutionSignature = void(FirstPoint<_1>, SecondPoint<_1>, _2);
```

CHAPTER

FORTYONE

NEW WORKLET TYPES

The basic building block for an algorithm in VTK-m is the worklet. Chapter 21 describes the different types of worklet types provided by VTK-m and how to use them to create algorithms. However, it is entirely possible that this set of worklet types does not directly cover what is needed to implement a particular algorithm. One way around this problem is to use some of the numerous back doors provided by VTK-m to provide less restricted access in the execution environment such as using whole arrays for random access.

However, it make come to pass that you encounter a particular pattern of execution that you find useful for implementing several algorithms. If such is the case, it can be worthwhile to create a new worklet type that directly supports such a pattern. Creating a new worklet type can provide two key advantages. First, it makes implementing algorithms of this nature easier, which saves developer time. Second, it can make the implementation of such algorithms safer. By encapsulating the management of structures and regulating the data access, users of the worklet type can be more assured of correct behavior.

This chapter documents the process for creating new worklet types. The operation of a worklet requires the coordination of several different object types such as dispatchers, argument handlers, and thread indices. This chapter will provide examples of all these required components. To tie all these features together, we start this chapter with a motivating example for an implementation of a custom worklet type. The chapter then discusses the individual components of the worklet, which in the end come together for the worklet type that is then demonstrated.

41.1 Motivating Example

For our motivation to create a new worklet type, let us consider the use case of building fractals. Fractals are generally not a primary concern of visualization libraries like VTK-m, but building a fractal (or approximations of fractals) has similarities the the computational geometry problems in scientific visualization. In particular, we consider the class of fractals that is generated by replacing each line in a shape with some collection of lines. These types of fractals are interesting because, in addition to other reasons, the right parameters result in a shape that has infinite length confined to a finite area.

A simple but well known example of a line fractal is the Koch Snowflake. The Koch Snowflake starts as a line or triangle that gets replaced with the curve shown in Figure 41.1.

The fractal is formed by iteratively replacing the curve's lines with this basic shape. Figure 41.2 shows the second iteration and then several subsequent iterations that create a “fuzzy” curve. The curve is confined to a limited area regardless of how many iterations are performed, but the length of the curve approaches infinity as the number of iterations approaches infinity.

In our finite world we want to estimate the curve of the Koch Snowflake by performing a finite amount of

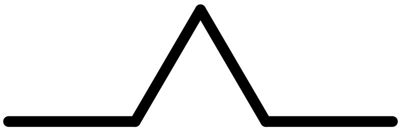


Figure 41.1: Basic shape for the Koch Snowflake.

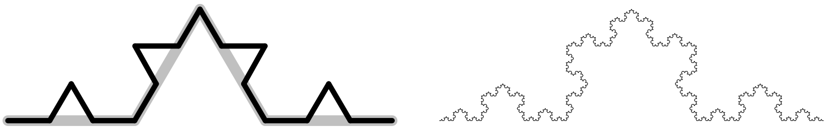


Figure 41.2: The Koch Snowflake after the second iteration (left image) and after several more iterations (right image).

iterations. This is similar to a Lindenmayer system but with less formality. The size of the curve grows quickly and in practice it takes few iterations to make close approximations.

Did you know?

The Koch Snowflake is just one example of many line fractals we can make with this recursive line substitution, which is why it is fruitful to create a worklet type to implement such fractals. We use the Koch Snowflake to set up the example here. Section 41.6 provides several more examples.

To implement line fractals of this nature, we want to be able to define the lines of the base shape in terms of parametric coordinates and then transform the coordinates to align with a line segment. For example, the Koch Snowflake base shape could be defined with parametric coordinates shown in Figure 41.3.

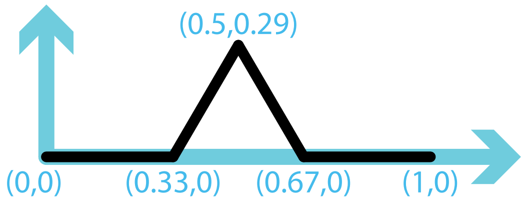


Figure 41.3: Parametric coordinates for the Koch Snowflake shape.

Given these parametric coordinates, for each line we define an axis with the main axis along the line segment and the secondary axis perpendicular to that. Given this definition, we can perform each fractal iteration by applying this transform for each line segment as shown in Figure 41.4.

To implement the application of the line fractal demonstrated in Figure 41.4, let us define a class named `LineFractalTransform` that takes as its constructor the coordinates of two ends of the original line. As its operator, `LineFractalTransform` takes a point in parametric space and returns the coordinates in world space in respect to the original line segment. We define this class in the `vtkm::exec` namespace because the intended use case is by worklets of the type we are making. A definition of `LineFractalTransform` is given in Example 41.1

Example 41.1: A support class for a line fractal worklet.

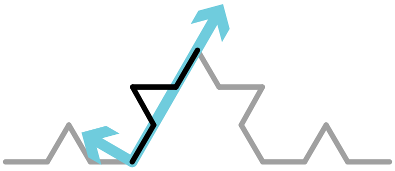


Figure 41.4: Applying the line fractal transform for the Koch Snowflake.

```
namespace vtkm
{
namespace exec
{

class LineFractalTransform
{
public:
  template<typename T>
  VTKM_EXEC LineFractalTransform(const vtkm::Vec<T, 2>& point0,
                                 const vtkm::Vec<T, 2>& point1)
  {
    this->Offset = point0;
    this->UAxis = point1 - point0;
    this->VAxis = vtkm::make_Vec(-this->UAxis[1], this->UAxis[0]);
  }

  template<typename T>
  VTKM_EXEC vtkm::Vec<T, 2> operator()(const vtkm::Vec<T, 2>& ppoint) const
  {
    vtkm::Vec2f ppointCast(ppoint);
    vtkm::Vec2f transform =
      ppointCast[0] * this->UAxis + ppointCast[1] * this->VAxis + this->Offset;
    return vtkm::Vec<T, 2>(transform);
  }

  template<typename T>
  VTKM_EXEC vtkm::Vec<T, 2> operator()(T x, T y) const
  {
    return (*this)(vtkm::Vec<T, 2>(x, y));
  }

private:
  vtkm::Vec2f Offset;
  vtkm::Vec2f UAxis;
  vtkm::Vec2f VAxis;
};

} // namespace exec
} // namespace vtkm
```

Did you know?

The definition of `LineFractalTransform` *(or something like it) is not strictly necessary for implementing a worklet type. However, it is common to implement such supporting classes that operate in the execution environment in support of the operations typically applied by the worklet type.*

The remainder of this chapter is dedicated to defining a `WorkletLineFractal` class and supporting objects that allow you to easily make line fractals. Example 41.2 demonstrates how we intend to use this worklet type.

Example 41.2: Demonstration of how we want to use the line fractal worklet.

```cpp
struct KochSnowflake
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<4>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.00f, 0.00f);
      segmentsOutVec[0][1] = transform(0.33f, 0.00f);

      segmentsOutVec[1][0] = transform(0.33f, 0.00f);
      segmentsOutVec[1][1] = transform(0.50f, 0.29f);

      segmentsOutVec[2][0] = transform(0.50f, 0.29f);
      segmentsOutVec[2][1] = transform(0.67f, 0.00f);

      segmentsOutVec[3][0] = transform(0.67f, 0.00f);
      segmentsOutVec[3][1] = transform(1.00f, 0.00f);
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.GetPortalControl().Set(1, vtkm::Vec2f(1.0f, 0.0f));

    vtkm::cont::Invoker invoke;
    KochSnowflake::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      invoke(worklet, points, outPoints);
      points = outPoints;
    }

    return points;
  }
};
```

41.2 Thread Indices

The first internal support class for implementing a worklet type is a class that manages indices for a thread. As the name would imply, the thread indices class holds a reference to an index identifying work to be done by the current thread. This includes indices to the current input element and the current output element. The thread indices object can also hold other information (that may not strictly be index data) about the input and output data. For example, the thread indices object for topology maps (named vtkm::exec::arg::ThreadIndicesTopologyMap) maintains cell shape and connection indices for the current input object.

As is discussed briefly in Section 40.3, a thread indices object is given to the vtkm::exec::arg::Fetch class to retrieve data from the execution object. The thread indices object serves two important functions for the Fetch. The first function is to cache information about the current thread that is likely to be used by multiple objects retrieving information. For example, in a point to cell topology map data from point fields must be retrieved by looking up indices in the topology connections. It is more efficient to retrieve the topology connections once and store them in the thread indices than it is to look them up independently for each field.

The second function of thread indices is to make it easier to find information about the input domain when fetching data. Once again, getting point data in a point to cell topology map requires looking up connectivity information in the input domain. However, the Fetch object for the point field does not have direct access to the data for the input domain. Instead, it gets this information from the thread indices.

All worklet classes have a method named GetThreadIndices that constructs a thread indices object for a given thread. GetThreadIndices is called with 6 parameters: a unique index for the thread (i.e. worklet instance), an array portal that maps output indices to input indices (which might not be one-to-one if a scatter is being used), an array portal that gives the visit index for each output index, an array portal that maps each unique thread index to output index for that thread, the execution object for the input domain, and an offset of the current index of the local invoke to a global indexing (used for streaming).

The base worklet implementation provides an implementation of GetThreadIndices that creates a vtkm::exec::arg::ThreadIndicesBasic object. This provides the minimum information required in a thread indices object, but non-trivial worklet types are likely to need to provide their own thread indices type. This following example shows the implementation of GetThreadIndices we will use in our worklet type superclass (discussed in more detail in Section 41.4).

Example 41.3: Implementation of GetThreadIndices in a worklet superclass.

```
VTKM_SUPPRESS_EXEC_WARNINGS
template<typename OutToInPortalType,
         typename VisitPortalType,
         typename ThreadToOutType,
         typename InputDomainType>
VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
  vtkm::Id threadIndex,
  const OutToInPortalType& outToIn,
  const VisitPortalType& visit,
  const ThreadToOutType& threadToOut,
  const InputDomainType& inputPoints,
  vtkm::Id globalThreadIndexOffset) const
{
  vtkm::Id outputIndex = threadToOut.Get(threadIndex);
  vtkm::Id inputIndex = outToIn.Get(outputIndex);
  vtkm::IdComponent visitIndex = visit.Get(outputIndex);
  return vtkm::exec::arg::ThreadIndicesLineFractal(threadIndex,
                                                   inputIndex,
                                                   visitIndex,
                                                   outputIndex,
                                                   inputPoints,
                                                   globalThreadIndexOffset);
}
```

As we can see in Example 41.3, our new worklet type needs a custom thread indices class. Specifically, we want the thread indices class to manage the coordinate information of the input line segment.

Did you know?

The implementation of a thread indices object we demonstrate here stores point coordinate information in addition to actual indices. It is acceptable for a thread indices object to store data that are not strictly indices. That said, the thread indices object should only load data (index or not) that is almost certain to be used by any worklet implementation. The thread indices object is created before any time that the worklet operator is called. If the thread indices object loads data that is never used by a worklet, that is a waste.

An implementation of a thread indices object usually derives from `vtkm::exec::arg::ThreadIndicesBasic` (or some other existing thread indices class) and adds to it information specific to a particular worklet type.

Example 41.4: Implementation of a thread indices class.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

class ThreadIndicesLineFractal : public vtkm::exec::arg::ThreadIndicesBasic
{
  using Superclass = vtkm::exec::arg::ThreadIndicesBasic;

public:
  using CoordinateType = vtkm::Vec2f;

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename InputPointPortal>
  VTKM_EXEC ThreadIndicesLineFractal(vtkm::Id threadIndex,
                                     vtkm::Id inputIndex,
                                     vtkm::IdComponent visitIndex,
                                     vtkm::Id outputIndex,
                                     const InputPointPortal& inputPoints,
                                     vtkm::Id globalThreadIndexOffset = 0)
    : Superclass(threadIndex,
                 inputIndex,
                 visitIndex,
                 outputIndex,
                 globalThreadIndexOffset)
  {
    this->Point0 = inputPoints.Get(this->GetInputIndex())[0];
    this->Point1 = inputPoints.Get(this->GetInputIndex())[1];
  }

  VTKM_EXEC
  const CoordinateType& GetPoint0() const { return this->Point0; }

  VTKM_EXEC
  const CoordinateType& GetPoint1() const { return this->Point1; }

private:
  CoordinateType Point0;
  CoordinateType Point1;
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

41.3 Signature Tags

It is common that when defining a new worklet type, the new worklet type is associated with new types of data. Thus, it is common that implementing new worklet types involves defining custom tags for ControlSignatures and ExecutionSignatures. This in turn typically requires creating custom TypeCheck, Transport, and Fetch classes.

Chapter 40 describes in detail the process of defining new worklet types and the associated code to manage data from a vtkm::cont::Invoker argument to the data that are passed to the worklet operator. Rather than repeat the discussion, readers should review Chapter 40 for details on how custom arguments are defined for a new worklet type. In particular, we use the code from Examples 40.2 (page 340), 40.4 (page 343), and 40.5 (page 345) to implement an argument representing 2D line segments (which is our input domain). All these examples culminate in the definition of a ControlSignature tag in our worklet superclass.

Example 41.5: Custom ControlSignature tag for the input domain of our example worklet type.

```
  struct SegmentsIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
    using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
  };
```

As you have worked with different existing worklet types, you have likely noticed that different worklet types have special ExecutionSignature tags to point to information in the input domain. For example, a point to cell topology map has special ExecutionSignature tags for getting the input cell shape and the indices to all points incident on the current input cell. We described in the beginning of the chapter that we wanted our worklet type to provide worklet implementations an object named LineFractalTransform (Example 41.1), so it makes sense to define our own custom ExecutionSignature tag to provide this object.

Chapter 40 gives an example of a custom ExecutionSignature tag that modifies what information is fetched from an argument (Examples 40.6 and 40.9). However, ExecutionSignature tags that only pull data from input domain behave a little differently because they only get information from the thread indices object and ignore the associated data object. This is done by providing a partial specialization of vtkm::exec::arg::Fetch that specializes on the aspect tag but not on the fetch tag.

Example 41.6: A Fetch for an aspect that does not depend on any control argument.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct AspectTagLineFractalTransform
{
};

template<typename FetchTag, typename ExecObjectType>
struct Fetch<FetchTag,
             vtkm::exec::arg::AspectTagLineFractalTransform,
             vtkm::exec::arg::ThreadIndicesLineFractal,
             ExecObjectType>
{
  using ValueType = LineFractalTransform;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC
  ValueType Load(const vtkm::exec::arg::ThreadIndicesLineFractal& indices,
                 const ExecObjectType&) const
  {
    return ValueType(indices.GetPoint0(), indices.GetPoint1());
  }

  VTKM_EXEC
  void Store(const vtkm::exec::arg::ThreadIndicesLineFractal&,
             const ExecObjectType&,
             const ValueType&) const
  {
    // Store is a no-op for this fetch.
  }
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

The definition of an associated ExecutionSignature tag simply has to use the define aspect as its AspectTag. The tag also has to define a INDEX member (which is required of all ExecutionSignature tags). This is problematic as this execution argument does not depend on any particular control argument. Thus, it is customary to simply set the INDEX to 1. There is guaranteed to be at least one ControlSignature argument for any worklet implementation. Thus, the first argument is sure to exist and can then be ignored.

Example 41.7: Custom ExecutionSignature tag that only relies on input domain information in the thread indices.

```
  struct Transform : vtkm::exec::arg::ExecutionSignatureTagBase
  {
    static const vtkm::IdComponent INDEX = 1;
    using AspectTag = vtkm::exec::arg::AspectTagLineFractalTransform;
  };
```

So far we have discussed how to get input line segments into our worklet. We also need a ControlSignature tag to represent the output line segments created by instances of our worklet. The motivating example has each worklet outputting a fixed number (greater than 1) of line segments for each input line segment. To manage this, we will define another ControlSignature tag that outputs these line segments (as two Vec-2 coordinates). This is defined as a Vec of Vec-2's. The tag takes the number of line segments as a template argument.

Example 41.8: Output ControlSignature tag for our motivating example.

```
  template<vtkm::IdComponent NumSegments>
  struct SegmentsOut : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumSegments>;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
  };
```

You can see that the tag in Example 41.8 relies on a custom transport named TransportTag2DLineSegmentsOut. There is nothing particularly special about this transport, but we provide the implementation here for completeness.

Example 41.9: Implementation of Transport for the output in our motivating example.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

```

```
template<vtkm::IdComponent NumOutputPerInput>
struct TransportTag2DLineSegmentsOut
{
};

template<vtkm::IdComponent NumOutputPerInput,
         typename ContObjectType,
         typename Device>
struct Transport<vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumOutputPerInput>,
                 ContObjectType,
                 Device>
{
  VTKM_IS_ARRAY_HANDLE(ContObjectType);

  using GroupedArrayType = vtkm::cont::ArrayHandleGroupVec<
    vtkm::cont::ArrayHandleGroupVec<ContObjectType, 2>,
    NumOutputPerInput>;

  using ExecObjectType =
    typename GroupedArrayType::template ExecutionTypes<Device>::Portal;

  template<typename InputDomainType>
  VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
                                      const InputDomainType&,
                                      vtkm::Id,
                                      vtkm::Id outputRange) const
  {
    GroupedArrayType groupedArray(vtkm::cont::make_ArrayHandleGroupVec<2>(object));
    return groupedArray.PrepareForOutput(outputRange, Device());
  }
};

} // namespace arg
} // namespace cont
} // namespace vtkm
```

In addition to these special ControlSignature tags that are specific to the nature of our worklet type, it is common to need to replicate some more common or general ControlSignature tags. One such tag, which is appropriate for our worklet type, is a “field” type that takes an array with exactly one value associated with each input or output element. We can build these field tags using existing type checks, transports, and fetches. The following example defines a FieldIn tag for our fractal worklet type. A FieldOut tag can be made in a similar manner.

Example 41.10: Implementing a FieldIn tag.

```
  struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray;
    using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
  };
```

41.4 Worklet Superclass

The penultimate step in defining a new worklet type is to define a class that will serve as the superclass of all implementations of worklets of this type. This class itself must inherit from vtkm::worklet::internal::WorkletBase. By convention the worklet superclass is placed in the vtkm::worklet namespace and its name starts with Worklet.

Within the worklet superclass we define the signature tags (as discussed in Section 41.3) and the GetThreadIndices method (as discussed in Section 41.2. The worklet superclass can also override other default behavior of the WorkletBase (such as special scatter). And the worklet superclass can provide other items that might be particularly useful to its subclasses (such as commonly used tags). Also, the worklet superclass must declare a Dispatcher template that points to a *dispatcher* object used to invoke the worklet. The dispatcher is created in Section 41.5.

Example 41.11: Superclass for a new type of worklet.

```
namespace vtkm
{
namespace worklet
{

template<typename WorkletType>
class DispatcherLineFractal;

class WorkletLineFractal : public vtkm::worklet::internal::WorkletBase
{
public:
  /// The dispatcher used to invoke worklets of this type.
  ///
  template<typename Worklet>
  using Dispatcher = vtkm::worklet::DispatcherLineFractal<Worklet>;

  /// Control signature tag for line segments in the plane. Used as the input
  /// domain.
  ///
  struct SegmentsIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
    using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
  };

  /// Control signature tag for a group of output line segments. The template
  /// argument specifies how many line segments are outputted for each input.
  /// The type is a Vec-like (of size NumSegments) of Vec-2's.
  ///
  template<vtkm::IdComponent NumSegments>
  struct SegmentsOut : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumSegments>;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
  };

  /// Control signature tag for input fields. There is one entry per input line
  /// segment. This tag takes a template argument that is a type list tag that
  /// limits the possible value types in the array.
  ///
  struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray;
    using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
  };

  /// Control signature tag for input fields. There is one entry per input line
  /// segment. This tag takes a template argument that is a type list tag that
  /// limits the possible value types in the array.
  ///
  struct FieldOut : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray;
    using TransportTag = vtkm::cont::arg::TransportTagArrayOut;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
  };

  /// Execution signature tag for a LineFractalTransform from the input.
  ///
  struct Transform : vtkm::exec::arg::ExecutionSignatureTagBase
  {
    static const vtkm::IdComponent INDEX = 1;
    using AspectTag = vtkm::exec::arg::AspectTagLineFractalTransform;
  };

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OutToInPortalType,
           typename VisitPortalType,
           typename ThreadToOutType,
           typename InputDomainType>
  VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
    vtkm::Id threadIndex,
    const OutToInPortalType& outToIn,
    const VisitPortalType& visit,
    const ThreadToOutType& threadToOut,
    const InputDomainType& inputPoints,
    vtkm::Id globalThreadIndexOffset) const
  {
    vtkm::Id outputIndex = threadToOut.Get(threadIndex);
    vtkm::Id inputIndex = outToIn.Get(outputIndex);
    vtkm::IdComponent visitIndex = visit.Get(outputIndex);
    return vtkm::exec::arg::ThreadIndicesLineFractal(threadIndex,
                                                     inputIndex,
                                                     visitIndex,
                                                     outputIndex,
                                                     inputPoints,
                                                     globalThreadIndexOffset);
  }
};

} // namespace worklet
} // namespace vtkm
```

Common Errors

Be wary of creating worklet superclasses that are templated. The C++ compiler rules for superclass templates that are only partially specialized are non-intuitive. If a subclass does not fully resolve the template, features of the superclass such as signature tags will have to be qualified with `typename` *keywords, which reduces the usability of the class.*

41.5 Dispatcher

Worklets are instantiated in the control environment and run in the execution environment. This means that the control environment must have a means to *invoke* worklets that start running in the execution environment. This is ostensibly done by the `vtkm::cont::Invoker` object, but the `Invoker` does this through a *dispatcher* object.

A dispatcher object is an object in the control environment that has an instance of a worklet and can invoke that worklet with a set of arguments. There are multiple types of dispatcher objects, each corresponding to a type of worklet object. All dispatcher objects have at least one template parameter: the worklet class being invoked, which is always the first argument. All dispatcher objects must be constructed with an instance of the worklet they are to invoke.

All dispatcher classes have a method named `Invoke` that launches the worklet in the execution environment. The arguments to `Invoke` must match those expected by the worklet, which is specified by something called a *control signature*.

The following is a list of the dispatchers defined in VTK-m. The dispatcher classes correspond to the list of worklet types specified in Chapter 21.

`vtkm::worklet::DispatcherMapField` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapField`. The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherMapTopology` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapTopology` or one of its sibling classes (such as `vtkm::worklet::WorkletVisitCellsWithPoints`). The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherPointNeighborhood` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletPointNeighborhood`. The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherReduceByKey` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletReduceByKey`. The dispatcher class has one template argument: the worklet type.

The final element required for a new worklet type is an associated dispatcher class for invoking the worklet.

Example 41.12: Standard template arguments for a dispatcher class.

```
template<typename WorkletType>
class DispatcherLineFractal
```

A dispatcher implementation inherits from `vtkm::worklet::internal::DispatcherBase`. `DispatcherBase` is itself a templated class with the following three templated arguments.

1. The dispatcher class that is subclassing `DispatcherBase`. All template arguments must be given.
2. The type of the worklet being dispatched (which by convention is the first argument of the dispatcher's template).
3. The expected superclass of the worklet, which is associated with the dispatcher implementation. `DispatcherBase` will check that the worklet has the appropriate superclass and provide a compile error if there is a mismatch.

Did you know?

The convention of having a subclass be templated on the derived class' type is known as the Curiously Recurring Template Pattern (CRTP). In the case of `DispatcherBase`*, VTK-m uses this CRTP behavior to allow the general implementation of* `Invoke` *to run* `DoInvoke` *in the subclass, which as we see in a moment is itself templated.*

Example 41.13: Subclassing DispatcherBase.

```
template<typename WorkletType>
class DispatcherLineFractal
  : public vtkm::worklet::internal::DispatcherBase<
      DispatcherLineFractal<WorkletType>,
      WorkletType,
      vtkm::worklet::WorkletLineFractal>
```

The dispatcher should have two constructors. The first constructor takes a worklet and a dispatcher. Both arguments should have a default value that is a new object created with its default constructor. It is good practice to put a warning on this constructor letting users know if they get a compile error there it is probably because the worklet or dispatcher does not have a default constructor and they need to provide one. The second constructor just takes a dispatcher.

Example 41.14: Typical constructor for a dispatcher.

```
  // If you get a compile error here about there being no appropriate constructor
  // for ScatterType, then that probably means that the worklet you are trying to
  // execute has defined a custom ScatterType and that you need to create one
  // (because there is no default way to construct the scatter). By convention,
  // worklets that define a custom scatter type usually provide a static method
  // named MakeScatter that constructs a scatter object.
  VTKM_CONT
  DispatcherLineFractal(const WorkletType& worklet = WorkletType(),
                        const ScatterType& scatter = ScatterType())
    : Superclass(worklet, scatter)
  {
  }

  VTKM_CONT
  DispatcherLineFractal(const ScatterType& scatter)
    : Superclass(WorkletType(), scatter)
  {
  }
```

Finally, the dispatcher must implement a const method named DoInvoke. The DoInvoke method should take a single argument. The argument will be an object of type vtkm::internal::Invocation although it is usually more convenient to just express the argument type as a single template parameter. The Invocation could contain several data items, so it is best to pass this argument as a constant reference.

Example 41.15: Declaration of DoInvoke of a dispatcher.

```
  template<typename Invocation>
  VTKM_CONT void DoInvoke(Invocation& invocation) const
```

Invocation is an object that encapsulates the state and data relevant to the invoke. Invocation contains multiple types and data items. For brevity only the ones most likely to be used in a DoInvoke method are documented here. We discuss these briefly before getting back to the implementation of DoInvoke.

vtkm::internal::Invocation contains a data member named Parameters that contains the data passed to the Invoke method of the dispatcher (with some possible transformations applied). Parameters is stored in a vtkm::internal::FunctionInterface template object. (FunctionInterface is described in Chapter 39.) The specific type of Parameters is defined as type ParameterInterface in the Invoke object.

The Invoke object also contains the types ControlInterface and ExecutionInterface that are FunctionInterface classes built from the ControlSignature and ExecutionSignature of the worklet. These FunctionInterface classes provide a simple mechanism for introspecting the arguments of the worklet's signatures.

All worklets must also define an input domain index, which points to one of the ControlSignature/Invoke arguments. This number is also captured in the vtkm::internal::Invocation object in a field named InputDomainIndex. For convenience, Invocation also has the type InputDomainTag set to be the same as the

ControlSignature argument corresponding to the input domain. Likewise, Invocation has the type InputDomainType set to be the same type as the (transformed) input domain argument to Invoke. Invocation also has a method name GetInputDomain that returns the invocation object passed to Invoke.

Getting back to the implementation of a dispatcher, the DoInvoke should first verify that the ControlSignature argument associated with the input domain is of the expected type. This can be done by comparing the Invocation::InputDomainTag with the expected signature tag using a tool like std::is_same. This step is not strictly necessary, but is invaluable to users diagnosing issues with using the dispatcher. It does not hurt to also check that the Invoke argument for the input domain is also the same as expected (by checking Invocation::InputDomainType). It is additionally helpful to have a descriptive comment near these checks.

Example 41.16: Checking the input domain tag and type.

```
    // Get the control signature tag for the input domain.
    using InputDomainTag = typename Invocation::InputDomainTag;

    // If you get a compile error on this line, then you have set the input
    // domain to something that is not a SegmentsIn parameter, which is not
    // valid.
    VTKM_STATIC_ASSERT(
      (std::is_same<InputDomainTag,
                    vtkm::worklet::WorkletLineFractal::SegmentsIn>::value));

    // This is the type for the input domain
    using InputDomainType = typename Invocation::InputDomainType;

    // If you get a compile error on this line, then you have tried to use
    // something that is not a vtkm::cont::ArrayHandle as the input domain to a
    // topology operation (that operates on a cell set connection domain).
    VTKM_IS_ARRAY_HANDLE(InputDomainType);
```

Next, DoInvoke must determine the size in number of elements of the input domain. When the default identity scatter is used, the input domain size corresponds to the number of instances the worklet is executed. (Other scatters will transform the input domain size to an output domain size, and that output domain size will determine the number of instances.) The input domain size is generally determined by using Invocation::::GetInputDomain and querying the input domain argument. In our motivating example, the input domain is an ArrayHandle and the input domain size is half the size of the array (since array entries are paired up into line segments).

The final thing DoInvoke does is call BasicInvoke on its DispatcherBase superclass. BasicInvoke does the complicated work of transferring arguments, scheduling the parallel job, and calling the worklet's operator. BasicInvoke takes three arguments: the Invocation object, the size of the input domain, and the device adapter tag to run on.

Example 41.17: Calling BasicInvoke from a dispatcher's DoInvoke.

```
    // We can pull the input domain parameter (the data specifying the input
    // domain) from the invocation object.
    const InputDomainType& inputDomain = invocation.GetInputDomain();

    // Now that we have the input domain, we can extract the range of the
    // scheduling and call BasicInvoke.
    this->BasicInvoke(invocation, inputDomain.GetNumberOfValues() / 2);
```

Putting this all together, the following example demonstrates the full implementation of the dispatcher for our motivating example.

Example 41.18: Implementation of a dispatcher for a new type of worklet.

```
namespace vtkm
{
namespace worklet
{
```

```

template<typename WorkletType>
class DispatcherLineFractal
  : public vtkm::worklet::internal::DispatcherBase<
      DispatcherLineFractal<WorkletType>,
      WorkletType,
      vtkm::worklet::WorkletLineFractal>
{
  using Superclass =
    vtkm::worklet::internal::DispatcherBase<DispatcherLineFractal<WorkletType>,
                                            WorkletType,
                                            vtkm::worklet::WorkletLineFractal>;
  using ScatterType = typename Superclass::ScatterType;

public:
  // If you get a compile error here about there being no appropriate constructor
  // for ScatterType, then that probably means that the worklet you are trying to
  // execute has defined a custom ScatterType and that you need to create one
  // (because there is no default way to construct the scatter). By convention,
  // worklets that define a custom scatter type usually provide a static method
  // named MakeScatter that constructs a scatter object.
  VTKM_CONT
  DispatcherLineFractal(const WorkletType& worklet = WorkletType(),
                        const ScatterType& scatter = ScatterType())
    : Superclass(worklet, scatter)
  {
  }

  VTKM_CONT
  DispatcherLineFractal(const ScatterType& scatter)
    : Superclass(WorkletType(), scatter)
  {
  }

  template<typename Invocation>
  VTKM_CONT void DoInvoke(Invocation& invocation) const
  {
    // Get the control signature tag for the input domain.
    using InputDomainTag = typename Invocation::InputDomainTag;

    // If you get a compile error on this line, then you have set the input
    // domain to something that is not a SegmentsIn parameter, which is not
    // valid.
    VTKM_STATIC_ASSERT(
      (std::is_same<InputDomainTag,
                    vtkm::worklet::WorkletLineFractal::SegmentsIn>::value));

    // This is the type for the input domain
    using InputDomainType = typename Invocation::InputDomainType;

    // If you get a compile error on this line, then you have tried to use
    // something that is not a vtkm::cont::ArrayHandle as the input domain to a
    // topology operation (that operates on a cell set connection domain).
    VTKM_IS_ARRAY_HANDLE(InputDomainType);

    // We can pull the input domain parameter (the data specifying the input
    // domain) from the invocation object.
    const InputDomainType& inputDomain = invocation.GetInputDomain();

    // Now that we have the input domain, we can extract the range of the
    // scheduling and call BasicInvoke.
    this->BasicInvoke(invocation, inputDomain.GetNumberOfValues() / 2);
  }
};
```

```
 
} // namespace worklet
} // namespace vtkm
```

41.6 Using the Worklet

Now that we have our full implementation of a worklet type that generates line fractals, let us have some fun with it. The beginning of this chapter shows an implementation of the Koch Snowflake. The remainder of this chapter demonstrates other fractals that are easily implemented with our worklet type.

41.6.1 Quadratic Type 2 Curve

There are multiple variants of the Koch Snowflake. One simple but interesting version is the quadratic type 1 curve. This fractal has a shape similar to what we used for Koch but has right angles and goes both up and down as shown in Figure 41.5.

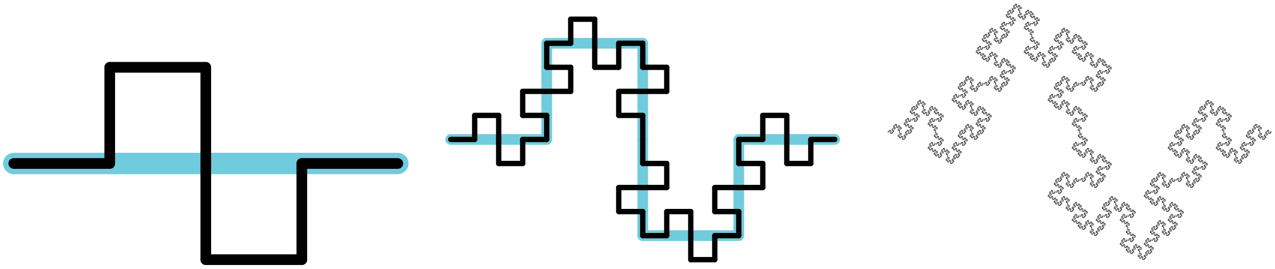


Figure 41.5: The quadratic type 2 curve fractal. The left image gives the first iteration. The middle image gives the second iteration. The right image gives the result after a few iterations.

The quadratic type 2 curve is implemented exactly like the Koch Snowflake except we output 8 lines to every input instead of 4, and, of course, the positions of the lines we generate are different.

Example 41.19: A worklet to generate a quadratic type 2 curve fractal.

```
struct QuadraticType2
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<8>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.00f, 0.00f);
      segmentsOutVec[0][1] = transform(0.25f, 0.00f);

      segmentsOutVec[1][0] = transform(0.25f, 0.00f);
      segmentsOutVec[1][1] = transform(0.25f, 0.25f);

      segmentsOutVec[2][0] = transform(0.25f, 0.25f);
      segmentsOutVec[2][1] = transform(0.50f, 0.25f);

      segmentsOutVec[3][0] = transform(0.50f, 0.25f);
      segmentsOutVec[3][1] = transform(0.50f, 0.00f);

      segmentsOutVec[4][0] = transform(0.50f, 0.00f);
      segmentsOutVec[4][1] = transform(0.50f, -0.25f);

      segmentsOutVec[5][0] = transform(0.50f, -0.25f);
      segmentsOutVec[5][1] = transform(0.75f, -0.25f);

      segmentsOutVec[6][0] = transform(0.75f, -0.25f);
      segmentsOutVec[6][1] = transform(0.75f, 0.00f);

      segmentsOutVec[7][0] = transform(0.75f, 0.00f);
      segmentsOutVec[7][1] = transform(1.00f, 0.00f);
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.GetPortalControl().Set(1, vtkm::Vec2f(1.0f, 0.0f));

    vtkm::cont::Invoker invoke;
    QuadraticType2::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      invoke(worklet, points, outPoints);
      points = outPoints;
    }

    return points;
  }
};
```

41.6.2 Tree Fractal

Another type of fractal we can make is a tree fractal. We will make a fractal similar to a Pythagoras tree except using lines instead of squares. Our fractal will start with a vertical line that will be replaced with the off-center “Y” shape shown in Figure 41.6. Iterative replacing using this “Y” shape produces a bushy tree shape.

One complication of implementing this tree fractal is that we really only want to apply the “Y” shape to the “leaves” of the tree. For example, once we apply the “Y” to the trunk, we do not want to apply it to the trunk again. If we were to apply it to the trunk again, we would create duplicates of the first layer of branches.

We can implement this feature in our worklet by using a count scatter. (Worklet scatters are described in Section 31.1.) Instead of directing the fractal worklet to generate 3 output line segments for every input line segment, we tell the fractal worklet to generate just 1 output line segment. We then use a scatter counting to generate 3 line segments for the leaves and 1 line segment for all other line segments. The count array for the initial iteration is initialized to a single 3. Each iteration then creates the count array for the next iteration by writing a 1 for the base line segment and a 3 from the other two line segments.

Example 41.20: A worklet to generate a tree fractal.

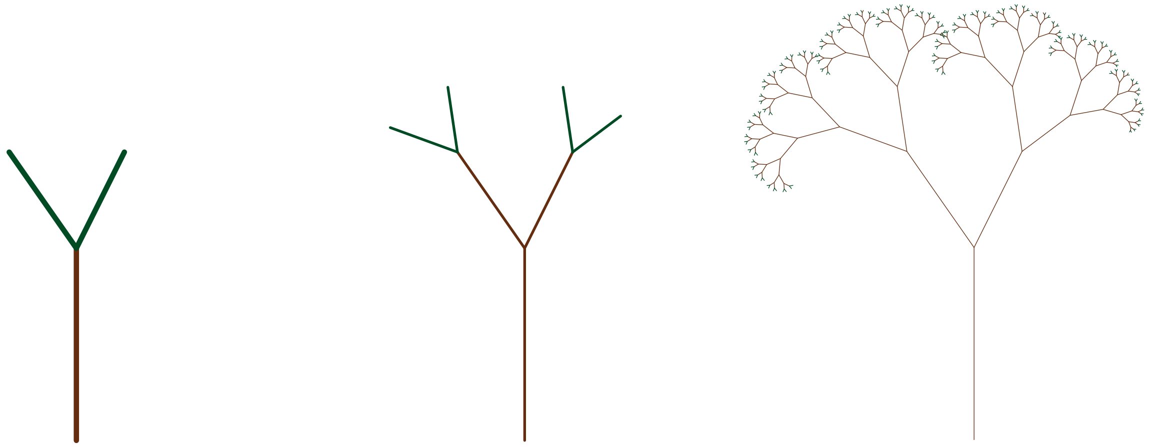


Figure 41.6: The tree fractal replaces each line with the “Y” shape shown at left. An iteration grows branches at the end (middle). After several iterations the tree branches out to the bushy shape at right.

```
struct TreeFractal
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn,
                                  SegmentsOut<1>,
                                  FieldOut countNextIteration);
    using ExecutionSignature = void(Transform, VisitIndex, _2, _3);
    using InputDomain = _1;

    using ScatterType = vtkm::worklet::ScatterCounting;

    template<typename Storage>
    VTKM_CONT static ScatterType MakeScatter(
      const vtkm::cont::ArrayHandle<vtkm::IdComponent, Storage>& count)
    {
      return ScatterType(count);
    }

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    vtkm::IdComponent visitIndex,
                    SegmentsOutVecType& segmentsOutVec,
                    vtkm::IdComponent& countNextIteration) const
    {
      switch (visitIndex)
      {
        case 0:
          segmentsOutVec[0][0] = transform(0.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.0f, 0.0f);
          countNextIteration = 1;
          break;
        case 1:
          segmentsOutVec[0][0] = transform(1.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.5f, -0.25f);
          countNextIteration = 3;
          break;
        case 2:
          segmentsOutVec[0][0] = transform(1.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.5f, 0.35f);
          countNextIteration = 3;
          break;
        default:
          this->RaiseError("Unexpected visit index.");
      }
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.GetPortalControl().Set(1, vtkm::Vec2f(0.0f, 1.0f));

    vtkm::cont::ArrayHandle<vtkm::IdComponent> count;

    // Initialize count array with 3 (meaning iterate)
    count.Allocate(1);
    count.GetPortalControl().Set(0, 3);

    vtkm::cont::Invoker invoke;
    TreeFractal::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      auto scatter = TreeFractal::FractalWorklet::MakeScatter(count);
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      invoke(worklet, scatter, points, outPoints, count);
      points = outPoints;
    }

    return points;
  }
};
```

41.6.3 Dragon Fractal

The next fractal we will implement is known as the dragon fractal. The dragon fractal is also sometimes known as the Heighway dragon or the Harter-Heighway dragon after creators John Heighway, Bruce Banks, and William Harter. It is also sometimes colloquially referred to as the Jurassic Park dragon as the fractal was prominently featured in the *Jurassic Park* novel by Michael Crichton.

The basic building block is simple. Each line segment is replaced by two line segments bent at 90 degrees and attached to the original segments endpoints as shown in Figure 41.7. As you can see by the fourth iteration a more complicated pattern starts to emerge. Figure 41.8 shows the twelfth iteration a demonstrates a repeating spiral.

What makes the dragon fractal different than the Koch Snowflake and similar fractals like the the quadratic curves implementation-wise is that the direction shape flips from one side to another. Note in the second image of Figure 41.7 the first bend is under the its associated line segment whereas the second is above its line segment. The easiest way for us to control the bend is to alternate the direction of the line segments. In Figure 41.7 each line segment has an arrowhead indicating the orientation of the first and second point with the arrowhead at the second point. Note that the shape is defined such that the first point of both line segments meet at the right angle. With the shape defined this way, each iteration is applied to put the bend to the left of the segment with respect to an observer at the first point looking at the second point.

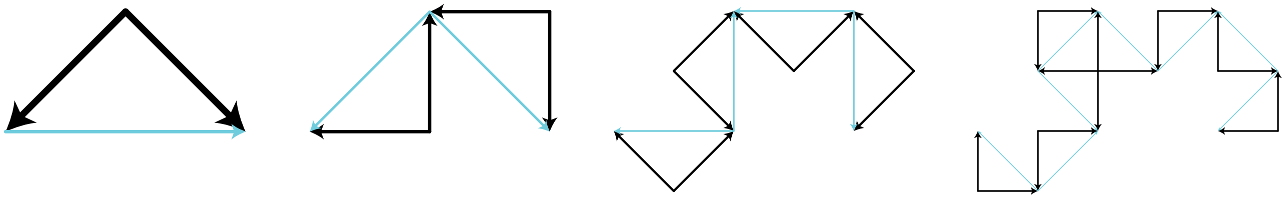


Figure 41.7: The first four iterations of the dragon fractal. The cyan lines give the previous iteration for reference.

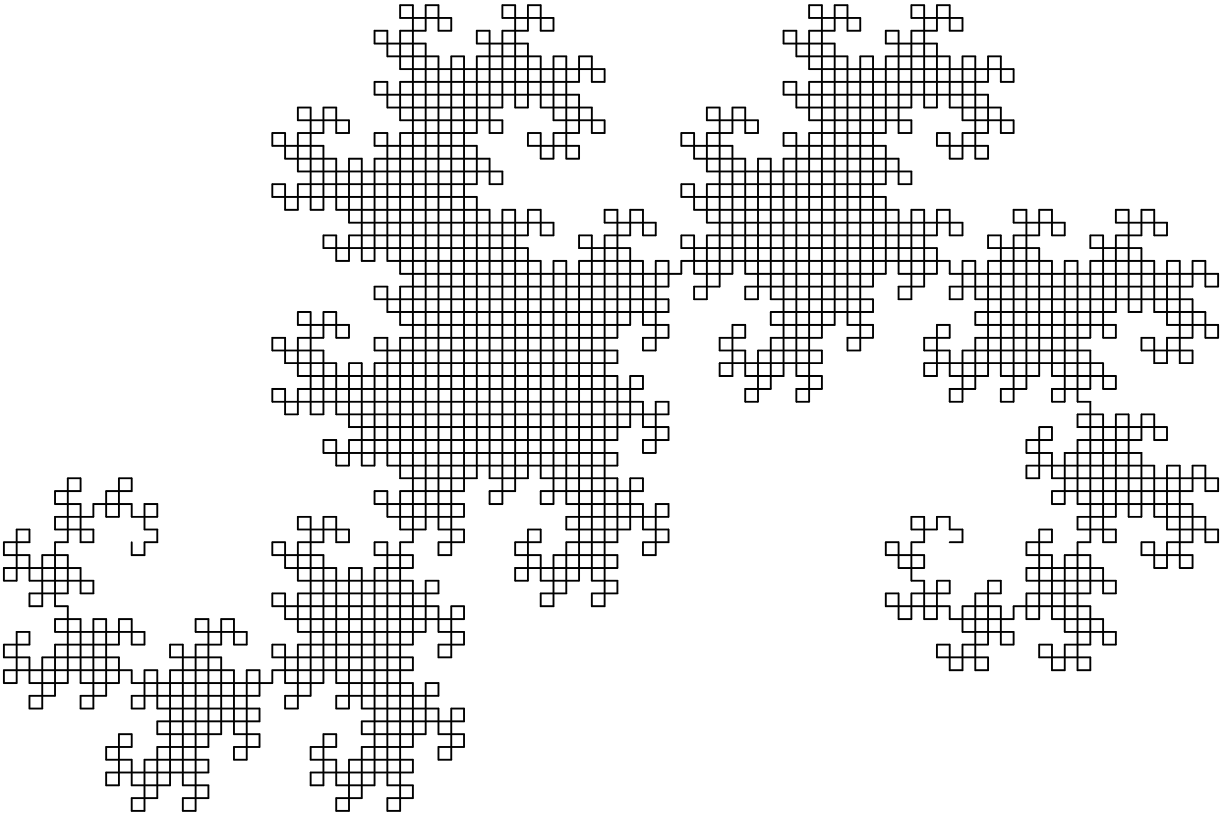


Figure 41.8: The dragon fractal after 12 iterations.

Other than reversing the direction of half the line segments, the implementation of the dragon fractal is nearly identical to the Koch Snowflake.

Example 41.21: A worklet to generate the dragon fractal.

```
struct DragonFractal
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<2>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.5f, 0.5f);
      segmentsOutVec[0][1] = transform(0.0f, 0.0f);

      segmentsOutVec[1][0] = transform(0.5f, 0.5f);
      segmentsOutVec[1][1] = transform(1.0f, 0.0f);
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.GetPortalControl().Set(1, vtkm::Vec2f(1.0f, 0.0f));

    vtkm::cont::Invoker invoke;
    DragonFractal::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      invoke(worklet, points, outPoints);
      points = outPoints;
    }

    return points;
  }
};
```

41.6.4 Hilbert Curve

For our final example we will look into using our fractal worklet to construct a space-filling curve. A space-filling curve is a type of fractal that defines a curve that, when iterated to its infinite length, completely fills a space. Space-filling curves have several practical uses by allowing you to order points in a 2 dimensional or higher space in a 1 dimensional array in such a way that points close in the higher dimensional space are usually close in the 1 dimensional ordering. For this fractal we will be generating the well-known Hilbert curve. (Specifically, we will be generating the 2D Hilbert curve.)

The 2D Hilbert curve fills in a rectangular region in space. (Our implementation will fill a unit square in the [0,1] range, but a simple scaling can generalize it to any rectangle.) Without loss of generality, we will say that the curve starts in the lower left corner of the region and ends in the lower right corner. The Hilbert curve starts by snaking around the lower-left corner then into the upper-left followed by the upper-right and then lower-right. The curve is typically generated by recursively dividing and orienting these quadrants.

To generate the Hilbert curve in our worklet system, we will define our line segments as the connection from the lower left of (entrance to) the region to the lower right of (exit from) the region. The fractal generation breaks this line to a 4 segment curve that moves up, then right, then back down. Figure 41.9 demonstrates the Hilbert curve. (Readers familiar with the Hilbert curve might notice the shape is a bit different than other representations. Where many derivations derive the Hilbert curve by connecting the center of oriented boxes, our derivation uses a line segment along one edge of these boxes. The result is a more asymmetrical shape in early iterations, but the two approaches are equivalent as the iterations approach infinity.)

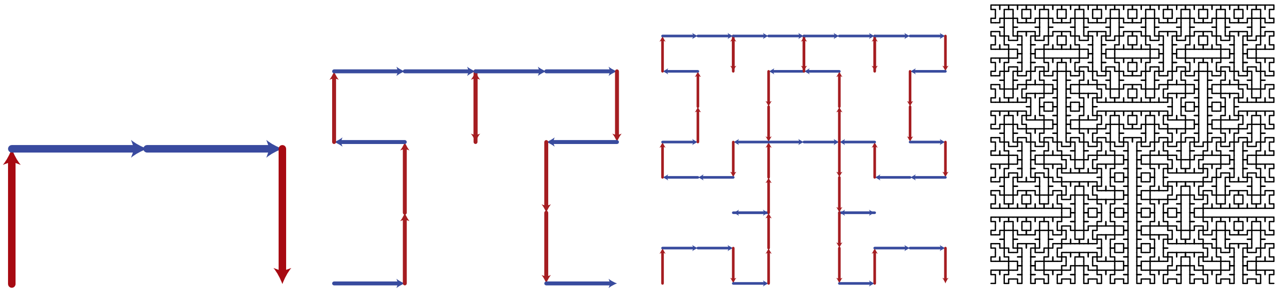


Figure 41.9: The first, second, third, and sixth iterations, respectively, of the Hilbert curve fractal.

Like the dragon fractal, the Hilbert curve needs to flip the shape in different directions. For example, the first iteration, shown at left in Figure 41.9, is drawn to the "left" of the initial line along the horizontal axis. The next iteration, the second image in Figure 41.9, is created by drawing the shape to the "right" of the vertical line segments but to the left of the horizontal segments.

Section 41.6.3 solved this problem for the dragon fractal by flipping the direction of some of the line segments. Such an approach would work for the Hilbert curve, but it results in line segments being listed out of order and with inconsistent directions with respect to the curve. For the dragon fractal, the order and orientation of line segments is of little consequence. But for many applications of a space-filling curve the distance along the curve is the whole point, so we want the order of the line segments to be consistent with the curve.

To support this flipped shape while preserving the line segment order, we will use a data field attached to the line segments. That is, each line segment will have a value to represent which way to draw the shape. If the field value is set to 1 (represented by the blue line segments in Figure 41.9), then the shape is drawn to the "left." If the field value is set to -1 (represented by the red line segments in Figure 41.9), then the shape is inverted and drawn to the "right." This field is passed in and out of the worklet using the `FieldIn` and `FieldOut` tags.

Example 41.22: A worklet to generate the Hilbert curve.

```cpp
struct HilbertCurve
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn,
                                  FieldIn directionIn,
                                  SegmentsOut<4>,
                                  FieldOut directionOut);
    using ExecutionSignature = void(Transform, _2, _3, _4);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    vtkm::Int8 directionIn,
                    SegmentsOutVecType& segmentsOutVec,
                    vtkm::Vec4i_8& directionOut) const
    {
      segmentsOutVec[0][0] = transform(0.0f, directionIn * 0.0f);
      segmentsOutVec[0][1] = transform(0.0f, directionIn * 0.5f);
      directionOut[0] = -directionIn;

      segmentsOutVec[1][0] = transform(0.0f, directionIn * 0.5f);
      segmentsOutVec[1][1] = transform(0.5f, directionIn * 0.5f);
      directionOut[1] = directionIn;

      segmentsOutVec[2][0] = transform(0.5f, directionIn * 0.5f);
      segmentsOutVec[2][1] = transform(1.0f, directionIn * 0.5f);
      directionOut[2] = directionIn;

      segmentsOutVec[3][0] = transform(1.0f, directionIn * 0.5f);
      segmentsOutVec[3][1] = transform(1.0f, directionIn * 0.0f);
      directionOut[3] = -directionIn;
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.GetPortalControl().Set(1, vtkm::Vec2f(1.0f, 0.0f));

    vtkm::cont::ArrayHandle<vtkm::Int8> directions;

    // Initialize direction with positive.
    directions.Allocate(1);
    directions.GetPortalControl().Set(0, 1);

    vtkm::cont::Invoker invoke;
    HilbertCurve::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      vtkm::cont::ArrayHandle<vtkm::Int8> outDirections;
      invoke(worklet,
             points,
             directions,
             outPoints,
             vtkm::cont::make_ArrayHandleGroupVec<4>(outDirections));
      points = outPoints;
      directions = outDirections;
    }

    return points;
  }
};
```

Part VI

Appendix

INDEX