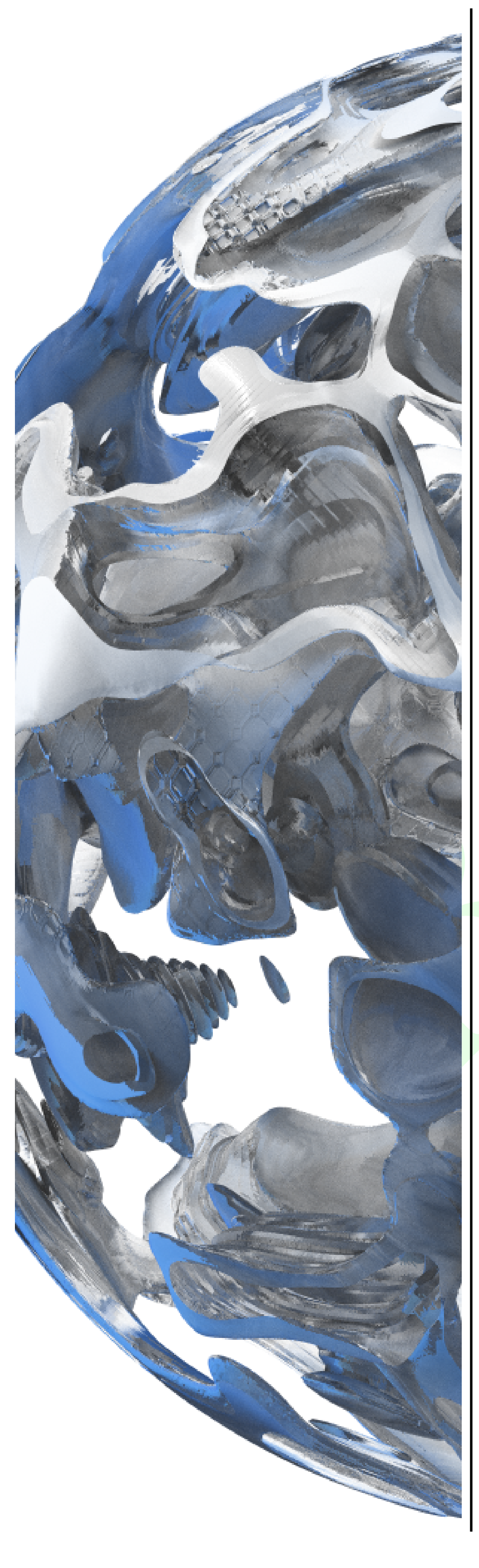


The VTK-m User's Guide

VTK-m version 1.0

Kenneth Moreland

May 2, 2016

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

DRAFT

Published by Kitware Inc. ©2016

All product names mentioned herein are the trademarks of their respective owners.
This document is available under a Creative Commons Attribution 4.0 International license available at [ADD URL]

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 multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Printed and produced in the United States of America.
ISBN number [FILL IN ISBN NUMBERS HERE]

DRAFT

CONTRIBUTORS

This book includes contributions from the VTK-m community including the VTK-m development team and the user community.

ABOUT THE COVER

Join the VTK-m Commuity at m.vtk.org

DRAFT

CONTENTS

I Getting Started 1

1 Introduction 3

1.1 How to Use This Guide . . . 3

1.2 Conventions Used in This Guide . . . 4

2 File I/O 7

2.1 Readers . . . 7

2.1.1 Legacy VTK File Reader . . . 7

2.2 Writers . . . 8

2.2.1 Legacy VTK File Writer . . . 8

3 Provided Filters 9

3.1 Field Filters . . . 9

3.2 Data Set Filters . . . 9

4 Rendering 11

II Using VTK-m 13

5 Basic Provisions 15

5.1 General Approach . . . 15

5.2 Package Structure . . . 16

5.3 Function and Method Exports . . . 17

5.4 Error Handling . . . 18

5.5 Core Data Types . . . 19

5.5.1 Single Number Types . . . 20

5.5.2 Vector Types . . . 20

DRAFT

5.5.3 Pair . . . 21
5.6 Traits . . . 21
5.6.1 Type Traits . . . 22
5.6.2 Vector Traits . . . 23
5.7 List Tags . . . 25
5.7.1 Building List Tags . . . 25
5.7.2 Type Lists . . . 26
5.7.3 Operating on Lists . . . 27

6 Array Handles **29**
6.1 Creating Array Handles . . . 29
6.2 Array Portals . . . 31
6.3 Allocating and Populating Array Handles . . . 33
6.4 Interface to Execution Environment . . . 34

7 Device Adapters **37**
7.1 Device Adapter Tag . . . 37
7.1.1 Default Device Adapter . . . 37
7.1.2 Specifying Device Adapter Tags . . . 39
7.2 Device Adapter Algorithms . . . 40
7.3 Implementing Device Adapters . . . 42
7.3.1 Tag . . . 43
7.3.2 Array Manager Execution . . . 43
7.3.3 Algorithms . . . 45
7.3.4 Timer Implementation . . . 49

8 Timers **51**

9 Fancy Array Storage **53**
9.1 Basic Storage . . . 54
9.2 Provided Fancy Arrays . . . 54
9.2.1 Constant Arrays . . . 55
9.2.2 Counting Arrays . . . 55
9.2.3 Cast Arrays . . . 56
9.2.4 Permuted Arrays . . . 57
9.2.5 Zipped Arrays . . . 58
9.2.6 Coordinate System Arrays . . . 59
9.2.7 Composite Vector Arrays . . . 60

DRAFT

9.2.8 Grouped Vector Arrays . . . 62
9.3 Implementing Fancy Arrays . . . 62
9.3.1 Implicit Array Handles . . . 62
9.3.2 Transformed Arrays . . . 64
9.3.3 Derived Storage . . . 66
9.4 Adapting Data Structures . . . 73

10 Dynamic Array Handles **79**
10.1 Querying and Casting . . . 79
10.2 Casting to Unknown Types . . . 81
10.3 Specifying Cast Lists . . . 82

11 Data Sets **85**
11.1 Building Data Sets . . . 85
11.1.1 Creating Uniform Grids . . . 85
11.1.2 Creating Rectilinear Grids . . . 86
11.1.3 Creating Explicit Meshes . . . 87
11.1.4 Add Fields . . . 89
11.2 Cell Sets . . . 90
11.2.1 Structured Cell Sets . . . 91
11.2.2 Explicit Cell Sets . . . 92
11.2.3 Cell Set Permutations . . . 93
11.2.4 Dynamic Cell Sets . . . 93
11.2.5 Blocks and Assemblies . . . 94
11.2.6 Zero Cell Sets . . . 94
11.3 Fields . . . 94
11.4 Coordinate Systems . . . 94

12 Filter Policies **97**

13 OpenGL Interoperability **99**

III Developing with VTK-m **101**

14 Worklets **103**
14.1 Worklet Types . . . 103
14.2 Dispatchers . . . 104
14.3 Provided Worklets . . . 104
14.4 Creating Worklets . . . 104

DRAFT

14.4.1 Control Signature . . . 105
Type List Tags . . . 106
14.4.2 Execution Signature . . . 107
14.4.3 Input Domain . . . 107
14.4.4 Worklet Operator . . . 107
14.5 Worklet Type Reference . . . 108
14.5.1 Field Map . . . 108
14.5.2 Topology Map . . . 111
Point to Cell Map . . . 111
Cell To Point Map . . . 114
General Topology Maps . . . 118
14.6 Whole Arrays . . . 120
14.7 Execution Objects . . . 123
14.8 Scatter . . . 124
14.9 Error Handling . . . 127

15 Creating Filters **129**

16 Math **131**
16.1 Basic Math . . . 131
16.2 Vector Analysis . . . 134
16.3 Matrices . . . 135
16.4 Newton's Method . . . 136

17 Working with Cells **139**
17.1 Cell Shape Tags and Ids . . . 139
17.1.1 Converting Between Tags and Identifiers . . . 139
17.1.2 Cell Traits . . . 141
17.2 Parametric and World Coordinates . . . 142
17.3 Interpolation . . . 143
17.4 Derivatives . . . 143

IV Advanced Development **145**

18 Advanced Worklet Customization **147**
18.1 Transferring Arguments from Control to Execution . . . 147
18.1.1 Type Checks . . . 147
18.1.2 Transport . . . 148

DRAFT

18.1.3 Fetch 149
18.2 Function Interface Objects 150
18.2.1 Declaring and Creating 150
18.2.2 Parameters 151
18.2.3 Invoking 152
18.2.4 Modifying Parameters 154
18.2.5 Transformations 155
18.2.6 For Each 158
18.3 Invocation Objects 159
18.4 Creating New `ControlSignature` Tags 159
18.5 Creating New `ExecutionSignature` Tags 159
18.6 Creating New Worklet Types 159
18.6.1 New Worklet Superclasses 159
18.6.2 Dispatch Workflow 159
18.6.3 New Dispatch Classes 159

V Appendix 161

A Coding Conventions 163

Index 167

DRAFT

LIST OF FIGURES

1.1 Comparison of Marching Cubes implementations. 4

5.1 Diagram of the VTK-m framework. 16
5.2 VTK-m package hierarchy. 17

9.1 Array handles, storage objects, and the underlying data source. 53

11.1 An example explicit mesh. 87
11.2 The arrangement of points and cells in a 3D structured grid. 91
11.3 Example of cells in a CellSetExplict and the arrays that define them. 92

14.1 Annotated example of a worklet declaration. 105

17.1 Basic Cell Shapes 140

DRAFT

DRAFT

LIST OF EXAMPLES

2.1 Reading a legacy VTK file. 7
2.2 Writing a legacy VTK file. 8
5.1 Usage of export macro. 18
5.2 Simple error reporting. 18
5.3 Using VTKM_ASSERT. 19
5.4 Creating vector types. 20
5.5 A Longer Vector. 21
5.6 Vector operations. 21
5.7 Repurposing a vtkm::Vec. 21
5.8 Definition of vtkm::TypeTraits<vtkm::Float32>. 22
5.9 Using TypeTraits for a generic remainder. 22
5.10 Definition of vtkm::VecTraits<vtkm::Id3>. 23
5.11 Using VecTraits for less functors. 24
5.12 Creating list tags. 25
5.13 Defining new type lists. 27
5.14 Converting dynamic types to static types with ListForEach. 27
6.1 Declaration of the vtkm::cont::ArrayHandle templated class. 29
6.2 Creating an ArrayHandle for output data. 29
6.3 Creating an ArrayHandle that points to a provided C array. 30
6.4 Creating an ArrayHandle that points to a provided std::vector. 30
6.5 Invalidating an ArrayHandle by letting the source std::vector leave scope. 30
6.6 A simple array portal implementation. 31
6.7 Using ArrayPortalToIterators. 32
6.8 Using ArrayPortalToIteratorBegin and ArrayPortalToIteratorEnd. 32
6.9 Using portals from an ArrayHandle. 33
6.10 Allocating an ArrayHandle. 34

DRAFT

6.11 Populating a newly allocated ArrayHandle. . . . 34
6.12 Using an execution array portal from an ArrayHandle. . . . 35
7.1 Macros to port VTK-m code among different devices 38
7.2 Specifying a device using a device adapter tag. . . . 39
7.3 Specifying a default device for template parameters. . . . 39
7.4 Prototype for vtkm::cont::DeviceAdapterAlgorithm. . . . 40
7.5 Contents of the base header for a device adapter. . . . 42
7.6 Implementation of a device adapter tag. . . . 43
7.7 Prototype for vtkm::cont::internal::ArrayManagerExecution. . . . 43
7.8 Specialization of ArrayManagerExecution. . . . 44
7.9 Minimal specialization of DeviceAdapterAlgorithm. . . . 46
7.10 Specialization of DeviceAdapterTimerImplementation. . . . 49
8.1 Using vtkm::cont::Timer. . . . 51
9.1 Declaration of the vtkm::cont::ArrayHandle templated class (again). . . . 54
9.2 Specifying the storage type for an ArrayHandle. . . . 54
9.3 Using ArrayHandleConstant. . . . 55
9.4 Using make_ArrayHandleConstant. . . . 55
9.5 Using ArrayHandleIndex. . . . 55
9.6 Using ArrayHandleCounting. . . . 55
9.7 Using make_ArrayHandleCounting. . . . 56
9.8 Counting backwards with ArrayHandleCounting. . . . 56
9.9 Using ArrayHandleCounting with vtkm::Vec objects. . . . 56
9.10 Using ArrayHandleCast. . . . 56
9.11 Using make_ArrayHandleCast. . . . 57
9.12 Using ArrayHandlePermutation. . . . 57
9.13 Using make_ArrayHandlePermutation. . . . 57
9.14 Using ArrayHandleZip. . . . 58
9.15 Using make_ArrayHandleZip. . . . 59
9.16 Using ArrayHandleUniformPointCoordinates. . . . 59
9.17 Using a ArrayHandleCartesianProduct. . . . 59
9.18 Using make_ArrayHandleCartesianProduct. . . . 60
9.19 Using ArrayHandleCompositeVector. . . . 61
9.20 Using make_ArrayHandleCompositeVector. . . . 61
9.21 Combining vector components with ArrayHandleCompositeVector. . . . 61
9.22 Using ArrayHandleGroupVec. . . . 62
9.23 Using make_ArrayHandleGroupVec. . . . 62
9.24 Functor that doubles an index. . . . 63

DRAFT

9.25 Declaring a ArrayHandleImplicit. 63
9.26 Using make_ArrayHandleImplicit. 63
9.27 Custom implicit array handle for even numbers. 63
9.28 Functor to scale and bias a value. 64
9.29 Using make_ArrayHandleTransform. 65
9.30 Custom transform array handle for scale and bias. 65
9.31 Derived array portal for concatenated arrays. 66
9.32 Storage for derived container of concatenated arrays. 67
9.33 Prototype for vtkm::cont::internal::ArrayTransfer. 69
9.34 Prototype for ArrayTransfer constructor. 70
9.35 ArrayTransfer for derived storage of concatenated arrays. 71
9.36 ArrayHandle for derived storage of concatenated arrays. 72
9.37 Fictitious field storage used in custom array storage examples. 73
9.38 Array portal to adapt a third-party container to VTK-m. 74
9.39 Prototype for vtkm::cont::internal::Storage. 75
9.40 Storage to adapt a third-party container to VTK-m. 76
9.41 Array handle to adapt a third-party container to VTK-m. 76
9.42 Using an ArrayHandle with custom container. 77
9.43 Redefining the default array handle storage. 78
10.1 Creating a DynamicArrayHandle. 79
10.2 Non type-specific queries on DynamicArrayHandle. 80
10.3 Using DynamicArrayHandle::NewInstance(). 80
10.4 Querying the component and storage types of a DynamicArrayHandle. 80
10.5 Casting a DynamicArrayHandle to a concrete ArrayHandle. 81
10.6 Operating on DynamicArrayHandle with CastAndCall. 81
10.7 Trying all component types in a DynamicArrayHandle. 83
10.8 Specifying a single component type in a DynamicArrayHandle. 83
10.9 Specifying different storage types in a DynamicArrayHandle. 83
10.10 Specifying both component and storage types in a DynamicArrayHandle. 83
10.11 Using DynamicArrayHandleBase to accept generic dynamic array handles. 84
11.1 Creating a uniform grid. 86
11.2 Creating a uniform grid with custom origin and spacing. 86
11.3 Creating a rectilinear grid. 86
11.4 Creating an explicit mesh with DataSetBuilderExplicit. 87
11.5 Creating an explicit mesh with DataSetBuilderExplicitIterative. 88
11.6 Adding fields to a DataSet. 89
11.7 Subsampling a data set with CellSetPermutation. 93

DRAFT

14.1 A ControlSignature. 105
14.2 An ExecutionSignature. 107
14.3 An InputDomain declaration. 107
14.4 An overloaded parenthesis operator of a worklet. 107
14.5 Implementation and use of a field map worklet. 109
14.6 Leveraging field maps and field maps for general processing. 110
14.7 Implementation and use of a map point to cell worklet. 113
14.8 Implementation and use of a map cell to point worklet. 116
14.9 Using WholeArrayIn to access a lookup table in a worklet. 120
14.10 Using ExecObject to access a lookup table in a worklet. 123
14.11 Declaration of a scatter type in a worklet. 125
14.12 Using ScatterUniform. 125
14.13 Using ScatterCounting. 126
14.14 Raising an error in the execution environment. 128
16.1 Creating a Matrix. 135
16.2 Using NewtonsMethod to solve a small system of nonlinear equations. 136
17.1 Using CellShapeIdToTag. 140
17.2 Using CellTraits to implement a polygon normal estimator. 141
17.3 Interpolating field values to a cell's center. 143
17.4 Computing the derivative of the field at cell centers. 143
18.1 Behavior of vtkm::cont::arg::TypeCheck. 148
18.2 Behavior of vtkm::cont::arg::Transport. 149
18.3 Declaring vtkm::internal::FunctionInterface. 150
18.4 Using vtkm::internal::make_FunctionInterface. 151
18.5 Getting the arity of a FunctionInterface. 151
18.6 Using FunctionInterface::GetParameter(). 151
18.7 Using FunctionInterface::SetParameter(). 152
18.8 Invoking a FunctionInterface. 152
18.9 Invoking a FunctionInterface with a transform. 152
18.10 Getting return value from FunctionInterface safely. 153
18.11 Appending parameters to a FunctionInterface. 154
18.12 Replacing parameters in a FunctionInterface. 154
18.13 Chaining Replace and Append with a FunctionInterface. 155
18.14 Using a static transform of function interface class. 156
18.15 Using a dynamic transform of a function interface. 156
18.16 Using DynamicTransform to cast dynamic arrays in a function interface. 157
18.17 Using the ForEach feature of FunctionInterface. 158

DRAFT

Part I

Getting Started

DRAFT

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 Dax 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 implementations for the Marching Cubes algorithm in VTK-m with the equivalent algorithms implemented in the CUDA software development kit reference implementation and the PISTON visualization library. 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 the other libraries that make code written in VTK-m more versatile.

Did you know?

VTK-m is written in C++ and makes extensive use of templates. The toolkit is implemented as a header library, meaning that all the code is implemented in header files (with extension `.h`*) and completely included in any code that uses it. This allows the compiler to inline and specialize code for better performance.*

1.1 How to Use This Guide

This user's guide is organized into three parts to help guide novice to advanced users and to provide a convenient reference. Part I, Getting Started, provides everything needed to get up and running with VTK-m. In this part we learn the basics of reading and writing data files, using filters to process data, and perform basic rendering to view the results.

Part II, Using VTK-m, dives deeper into the VTK-m library and provides all the information needed to customize VTK-m's data structures and support multiple devices.

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

Part III, Developing with VTK-m, documents how to use VTK-m's framework to develop new or custom visualization algorithms. This part describes how worklets are used to implement and execute algorithms and how to use worklets to implement new filters. Part III also describes the facilities available in the execution environment that help write visualization algorithms.

Part IV, Advanced Development, exposes the inner workings of VTK-m and allows you to design new algorithmic structures not already available. [This might be removed in the first version of the book.]

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`.

DRAFT

- Macros and namespaces from VTK-m are printed in `red`.
- Identifiers from VTK-m are printed in `blue`.
- Signatures, described in Chapter 14, 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.*

DRAFT

CHAPTER

TWO

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 Part II we explore efficient ways to define VTK-m data structures. In particular, Section 11.1 describes how to build VTK-m data set objects and Section 9.4 documents how to adapt data structures defined in other libraries to be used directly in VTK-m.

2.1 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. Section **??** provides much more details about the contents of a data set object, but for now we treat `DataSet` as an opaque object that can be passed around readers, writers, filters, and rendering units.

2.1.1 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 2.1: Reading a legacy VTK file.

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

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

DRAFT

```
2
3 vtkm::cont::DataSet OpenDataFromVTKFile()
4 {
5   vtkm::io::reader::VTKDataSetReader reader("data.vtk");
6
7   return reader.ReadDataSet();
8 }
```

2.2 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.

2.2.1 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 2.2: Writing a legacy VTK file.

```
1 #include <vtkm/io/writer/VTKDataSetWriter.h>
2
3 void SaveDataAsVTKFile(vtkm::cont::DataSet data)
4 {
5   vtkm::io::writer::VTKDataSetWriter writer("data.vtk");
6
7   writer.WriteDataSet(data);
8 }
```

DRAFT

CHAPTER

THREE

PROVIDED 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 introduced with the file I/O operations in Chapter **??** and are described in more detail in Chapter 11. For now we treat `DataSet` mostly as an opaque object that can be passed around readers, writers, filters, and rendering units.

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. We group filters based on the type of operation that they do and the shared interfaces that they have. Later Part **??** describes the necessary steps in creating new filters in VTK-m.

3.1 Field Filters

Every `vtkm::cont::DataSet` object contains a list of *fields*. A field describes some numerical value associated with different parts of the data set in space. Fields often represent physical properties such as temperature, pressure, or velocity. HEREHEREHERE.

3.2 Data Set Filters

DRAFT

DRAFT

CHAPTER

FOUR

RENDERING

[Write this once the rendering module is implemented.]

DRAFT

Part II

Using VTK-m

DRAFT

CHAPTER

FIVE

BASIC PROVISIONS

This section describes the core facilities provided by VTK-m. These include macros, types, and classes that define the environment in which code is run, the core types of data stored, and template introspection. We also start with a description of package structure used by VTK-m.

5.1 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 small functor or kernel designed to operate on a small element of data. (The name "worklet" means a small amount of 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.

DRAFT

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.

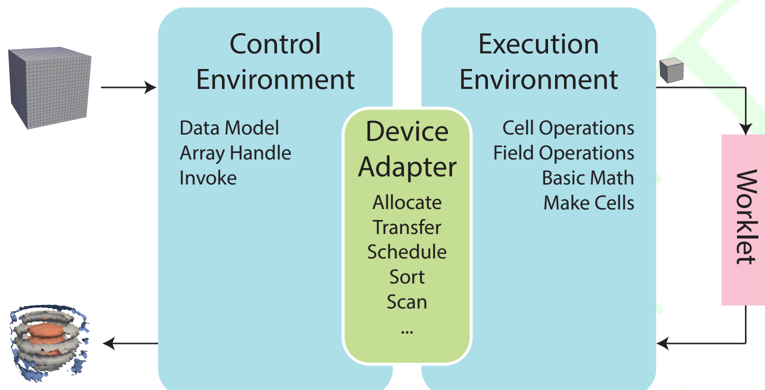


Figure 5.1: Diagram of the VTK-m framework.

Figure 5.1 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.

DRAFT

5.2 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 in Section 5.1, 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 (see the coding conventions in Chapter A), 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 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, described in Chapter 14, 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 I.

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 5.2 provides a diagram of the VTK-m package hierarchy.

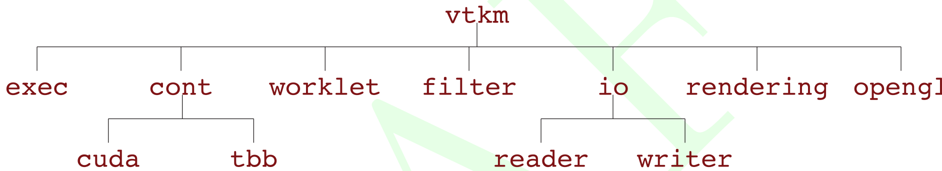


Figure 5.2: VTK-m package hierarchy.

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::ArrayHandle class is found in the vtkm/cont/ArrayHandle.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.

5.3 Function and Method Exports

Any function or method defined by VTK-m must come with an export modifier that determines in which environments the function may be run. These export 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 export macros, VTKM_CONT_EXPORT, VTKM_EXEC_EXPORT, and VTKM_EXEC_CONT_EXPORT, 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 export 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 exported to both places.

Example 5.1: Usage of export macro.

```
1 template<typename ValueType>
2 VTKM_EXEC_CONT_EXPORT
3 ValueType Square(const ValueType &inValue)
4 {
5   return inValue * inValue;
6 }
```

The primary function of the export macros is to inject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control exports have `__host__` in them and execution exports have `__device__` in them.

There is one additional export macro that is not used for functions but rather used when declaring a constant data object that is used in the execution environment. This macro is named `VTKM_EXEC_CONSTANT_EXPORT` and is used to declare a constant lookup table used when executing a worklet. Its primary reason for existing is to add a `__constant__` keyword when compiling with CUDA. This export currently has no effect on any other compiler.

Finally, it is sometimes the case that a function declared as `VTKM_EXEC_CONT_EXPORT` has to call a method declared as `VTKM_EXEC_EXPORT` or `VTKM_CONT_EXPORT`. Generally functions should not call other functions with incompatible control/execution exports, but sometimes a generic `VTKM_EXEC_CONT_EXPORT` function calls another function determined by the template parameters, and the export 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 export macro.

5.4 Error Handling

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 `GetMessage` method.

Example 5.2: Simple error reporting.

```
1 int main(int argc, char **argv)
2 {
3   try
4   {
5     // Do something cool with VTK-m
6     // ...
7   }
8   catch (vtkm::cont::Error error)
9   {
10    std::cout << error.GetMessage() << std::endl;
11    return 1;
12  }
13  return 0;
14 }
```

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::ErrorControlBadAllocation` 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::ErrorControlBadType` Thrown when VTK-m attempts to perform an operation on an object that is of an incompatible type.

`vtkm::cont::ErrorControlBadValue` 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::ErrorControlInternal` 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.

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 5.3: Using `VTKM_ASSERT`.

```
1 template<typename T>
2 VTKM_CONT_EXPORT
3 T GetArrayValue(vtkm::cont::ArrayHandle<T> arrayHandle, vtkm::Id index)
4 {
5   VTKM_ASSERT(index >= 0);
6   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.

5.5 Core Data Types

Except in rare circumstances where precision is not a concern, VTK-m does not directly use the core C types like `int`, `float`, and `double`. Instead, VTK-m provides its own core types, which are declared in vtkm/Types.h.

5.5.1 Single Number Types

To ensure portability across different compilers and architectures, VTK-m provides `typedef`s for the following basic types with explicit precision: `vtkm::Float32`, `vtkm::Float64`, `vtkm::Int8`, `vtkm::Int16`, `vtkm::Int32`, `vtkm::Int64`, `vtkm::UInt8`, `vtkm::UInt16`, `vtkm::UInt32`, and `vtkm::UInt64`. Under most circumstances when using VTK-m (and performing visualization in general) the type of data is determined by the source of the data or resolved through templates. In the case where a specific type of data is required, these VTK-m–defined types should be preferred over basic C types like `int` or `float`.

Many of the structures in VTK-m require indices to identify elements like points and cells. All indices for arrays and other lists use the type `vtkm::Id`. By default this type is a 32-bit wide integer but can be easily changed by compile options. The CMake configuration option VTKM_USE_64BIT_IDS can be used to change `vtkm::Id` to be 64 bits wide. This configuration can be overridden by defining the C macro `VTKM_USE_64BIT_IDS` or `VTKM_NO_64BIT_IDS` to force `vtkm::Id` to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

There is also a secondary index type named `vtkm::IdComponent` that is used to index components of short vectors (discussed in Section 5.5.2). This type is an integer that might be a shorter width than `vtkm::Id`.

There is also the rare circumstance in which an algorithm in VTK-m computes data values for which there is no indication what the precision should be. For these circumstances, the type `vtkm::FloatDefault` is provided. By default this type is a 32-bit wide floating point number but can be easily changed by compile options. The CMake configuration option VTKM_USE_DOUBLE_PRECISION can be used to change `vtkm::FloatDefault` to be 64 bits wide. This configuration can be overridden by defining the C macro `VTKM_USE_DOUBLE_PRECISION` or `VTKM_NO_DOUBLE_PRECISION` to force `vtkm::FloatDefault` to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

For convenience, you can include either vtkm/internal/ConfigureFor32.h or vtkm/internal/ConfigureFor64.h to force both `vtkm::Id` and `vtkm::FloatDefault` to be 32 or 64 bits.

5.5.2 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 with a size of 4 or less is specialized to also have a constructor that allows you to set the individual components. Likewise, there is a `vtkm::make_Vec` function that builds initialized vector types of up to 4 components. Once created, you can use the bracket operator to get and set component values with the same syntax as an array.

Example 5.4: Creating vector types.

```
1  vtkm::Vec<vtkm::Float32,3> A(1);                        // A is (1, 1, 1)
2  A[1] = 2;                                               // A is now (1, 2, 1)
3  vtkm::Vec<vtkm::Float32,3> B(1, 2, 3);                  // B is (1, 2, 3)
4  vtkm::Vec<vtkm::Float32,3> C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
```

The types `vtkm::Id2` and `vtkm::Id3` are `typedef`s of `vtkm::Vec<vtkm::Id,2>` and `vtkm::Vec<vtkm::Id,2>`. These are used to index arrays of 2 and 3 dimensions, which is common.

Vectors longer than 4 are also supported, but independent component values must be set after construction. The `vtkm::Vec` class contains a constant named `NUM_COMPONENTS` to specify how many components are in the vector.

Example 5.5: A Longer Vector.

```
vtkm::Vec<vtkm::Float32, 5> A(2); // A is (2, 2, 2, 2, 2)
for (vtkm::IdComponent index = 1; index < A.NUM_COMPONENTS; index++)
{
  A[index] = A[index-1] * 1.5;
}
// A is now (2, 3, 4.5, 6.75, 10.125)
```

`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 5.6: Vector operations.

```
vtkm::Vec<vtkm::Float32,3> A(1, 2, 3);
vtkm::Vec<vtkm::Float32,3> B(4, 5, 6.5);
vtkm::Vec<vtkm::Float32,3> C = A + B;          // C is (5, 7, 9.5)
vtkm::Vec<vtkm::Float32,3> 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
```

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 5.7: Repurposing a `vtkm::Vec`.

```
vtkm::Vec<vtkm::Vec<vtkm::Float32,2>, 3> equilateralTriangle(
                                              vtkm::make_Vec(0.0, 0.0),
                                              vtkm::make_Vec(1.0, 0.0),
                                              vtkm::make_Vec(0.5, 0.866));
```

5.5.3 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.

5.6 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 typedefs 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.

DRAFT

5.6.1 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 Section 5.5. 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.

The definition of vtkm::TypeTraits for vtkm::Float32 could like something like this.

Example 5.8: Definition of vtkm::TypeTraits<vtkm::Float32>.

```
1  namespace vtkm {
2
3  template<>
4  struct TypeTraits<vtkm::Float32>
5  {
6    typedef vtkm::TypeTraitsRealTag NumericTag;
7    typedef vtkm::TypeTraitsScalarTag DimensionalityTag;
8  };
9
10 }
```

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 5.9: Using TypeTraits for a generic remainder.

```
1  #include <vtkm/TypeTraits.h>
2
3  #include <vtkm/Math.h>
4
5  template<typename T>
6  T Remainder(const T &numerator, const T &denominator);
7
8  namespace detail {
9
10 template<typename T>
11 T RemainderImpl(const T &numerator,
12                 const T &denominator,
13                 vtkm::TypeTraitsIntegerTag,
14                 vtkm::TypeTraitsScalarTag)
15 {
16   return numerator % denominator;
17 }
18
19 template<typename T>
20 T RemainderImpl(const T &numerator,
21                 const T &denominator,
22                 vtkm::TypeTraitsRealTag,
23                 vtkm::TypeTraitsScalarTag)
24 {
25   // The VTK-m math library contains a Remainder function that operates on
26   // floating point numbers.
27   return vtkm::Remainder(numerator, denominator);
28 }
29
30 template<typename T, typename NumericTag>
```

```
31  T RemainderImpl(const T &numerator,
32                  const T &denominator,
33                  NumericTag,
34                  vtkm::TypeTraitsVectorTag)
35  {
36    T result;
37    for (int componentIndex = 0;
38         componentIndex < T::NUM_COMPONENTS;
39         componentIndex++)
40    {
41      result[componentIndex] =
42          Remainder(numerator[componentIndex], denominator[componentIndex]);
43    }
44    return result;
45  }
46
47  } // namespace detail
48
49  template<typename T>
50  T Remainder(const T &numerator, const T &denominator)
51  {
52    return detail::RemainderImpl(numerator,
53                                 denominator,
54                                 typename vtkm::TypeTraits<T>::NumericTag(),
55                                 typename vtkm::TypeTraits<T>::DimensionalityTag());
56  }
```

5.6.2 Vector Traits

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`.

`NUM_COMPONENTS` An integer specifying how many components are contained in the vector.

`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.

`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.

`ToVec` A static method that converts a vector of the given type to a `vtkm::Vec`.

The definition of `vtkm::VecTraits` for `vtkm::Id3` could like something like this.

Example 5.10: Definition of `vtkm::VecTraits<vtkm::Id3>`.

```
1  namespace vtkm {
2
3  template<>
4  struct VecTraits<vtkm::Id3>
5  {
6    typedef vtkm::Id ComponentType;
7    static const int NUM_COMPONENTS = 3;
8    typedef VecTraitsTagMultipleComponents HasMultipleComponents;
```

```
9
10  VTKM_EXEC_CONT_EXPORT
11  static vtkm::Id &GetComponent(vtkm::Id3 &vector, int component) {
12    return vector[component];
13  }
14
15  VTKM_EXEC_CONT_EXPORT
16  static void SetComponent(vtkm::Id3 &vector, int component, vtkm::Id value) {
17    vector[component] = value;
18  }
19
20  VTKM_EXEC_CONT_EXPORT
21  static vtkm::Vec<vtkm::Id,3> ToTuple(const vtkm::Id3 &vector) {
22    return vector;
23  }
24 };
25
26 } // 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 5.11: Using VecTraits for less functors.

```
1  #include <vtkm/VecTraits.h>
2
3  // This functor provides a total ordering of vectors. Every compared vector
4  // will be either less, greater, or equal (assuming all the vector components
5  // also have a total ordering).
6  template<typename T>
7  struct LessTotalOrder
8  {
9    VTKM_EXEC_CONT_EXPORT
10   bool operator()(const T &left, const T &right)
11   {
12     for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
13     {
14       typedef typename vtkm::VecTraits<T>::ComponentType ComponentType;
15       const ComponentType &leftValue =
16           vtkm::VecTraits<T>::GetComponent(left, index);
17       const ComponentType &rightValue =
18           vtkm::VecTraits<T>::GetComponent(right, index);
19       if (leftValue < rightValue) { return true; }
20       if (rightValue < leftValue) { return false; }
21     }
22     // If we are here, the vectors are equal (or at least equivalent).
23     return false;
24   }
25 };
26
27 // This functor provides a partial ordering of vectors. It returns true if and
28 // only if all components satisfy the less operation. It is possible for
29 // vectors to be neither less, greater, nor equal, but the transitive closure
30 // is still valid.
31 template<typename T>
32 struct LessPartialOrder
33 {
34   VTKM_EXEC_CONT_EXPORT
35   bool operator()(const T &left, const T &right)
36   {
37     for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
38     {
```

```
39        typedef typename vtkm::VecTraits<T>::ComponentType ComponentType;
40        const ComponentType &leftValue =
41            vtkm::VecTraits<T>::GetComponent(left, index);
42        const ComponentType &rightValue =
43            vtkm::VecTraits<T>::GetComponent(right, index);
44        if (!(leftValue < rightValue)) { return false; }
45      }
46      // If we are here, all components satisfy less than relation.
47      return true;
48    }
49  };
```

5.7 List Tags

VTK-m internally uses template metaprogramming, which utilizes the C++ template 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 5.7.2. 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.)

5.7.1 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 `typedef`. Although `typedef` will work, it will lead to much uglier type names defined by the compiler.

Example 5.12: Creating list tags.

```
1  #include <vtkm/ListTag.h>
2
3  // Placeholder classes representing things that might be in a template
4  // metaprogram list.
5  class Foo;
6  class Bar;
```

```
7  class Baz;
8  class Qux;
9  class Xyzzy;
10
11 // The names of the following tags are indicative of the lists they contain.
12
13 struct FooList : vtkm::ListTagBase<Foo> { };
14
15 struct FooBarList : vtkm::ListTagBase<Foo,Bar> { };
16
17 struct BazQuxXyzzyList : vtkm::ListTagBase<Baz,Qux,Xyzzy> { };
18
19 struct QuxBazBarFooList : vtkm::ListTagBase<Qux,Baz,Bar,Foo> { };
20
21 struct FooBarBazQuxXyzzyList
22     : vtkm::ListTagJoin<FooBarList, BazQuxXyzzyList> { };
```

5.7.2 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::TypeListTagFieldVec3` A list containing types for values of fields with 3 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::Vecs 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 dynamic arrays (described in Section ??).

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 5.7.1 as demonstrated in the following example.

Example 5.13: 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::Vec<vtkm::Float32,2>,
                        vtkm::Vec<vtkm::Float64,2> > {  };

// 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::DynamicArrayHandle (Section ??). 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 5.13 also contains an example of overriding the VTKM_DEFAULT_TYPE_LIST_TAG macro.

5.7.3 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::DynamicArrayHandle (which is documented in Section ??).

Example 5.14: Converting dynamic types to static types with ListForEach.

```
struct MyArrayBase {
  // A virtual destructor makes sure C++ RTTI will be generated. It also helps
  // ensure subclass destructors are called.
  virtual ~MyArrayBase() {  }
};

```

```
 7  template<typename T>
 8  struct MyArrayImpl : public MyArrayBase {
 9    std::vector<T> Array;
10  };
11
12  template<typename T>
13  void PrefixSum(std::vector<T> &array)
14  {
15    T sum(typename vtkm::VecTraits<T>::ComponentType(0));
16    for (typename std::vector<T>::iterator iter = array.begin();
17         iter != array.end();
18         iter++)
19    {
20      sum = sum + *iter;
21      *iter = sum;
22    }
23  }
24
25  struct PrefixSumFunctor {
26    MyArrayBase *ArrayPointer;
27
28    PrefixSumFunctor(MyArrayBase *arrayPointer) : ArrayPointer(arrayPointer) {  }
29
30    template<typename T>
31    void operator()(T) {
32      typedef MyArrayImpl<T> ConcreteArrayType;
33      ConcreteArrayType *concreteArray =
34          dynamic_cast<ConcreteArrayType *>(this->ArrayPointer);
35      if (concreteArray != NULL)
36      {
37        PrefixSum(concreteArray->Array);
38      }
39    }
40  };
41
42  void DoPrefixSum(MyArrayBase *array)
43  {
44    PrefixSumFunctor functor = PrefixSumFunctor(array);
45    vtkm::ListForEach(functor, vtkm::TypeListTagCommon());
46  }
```

DRAFT

CHAPTER

SIX

ARRAY HANDLES

An *array handle*, 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 allocate and 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.

6.1 Creating Array Handles

vtkm::cont::ArrayHandle is a templated class with two template parameters. The first template parameter is the only one required and specifies the base type of the entries in the array. The second template parameter specifies the storage used when storing data in the control environment. Storage objects are discussed later in Chapter 9, and for now we will use the default value.

Example 6.1: Declaration of the vtkm::cont::ArrayHandle templated class.

```
1 template<
2     typename T,
3     typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
4 class ArrayHandle;
```

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 6.2: 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 6.3: 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 6.4: 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` 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 6.4 where to be placed within a callable function or method, it could cause the `vtkm::cont::ArrayHandle` to become invalid.

Common Errors

Because ArrayHandle does not manage data provided by make_ArrayHandle, you should only use these as temporary objects. Example 6.5 demonstrates a method of copying one of these temporary arrays into safe managed memory, and Section 6.3 describes how to put data directly into an ArrayHandle object.

Example 6.5: Invalidating an ArrayHandle by letting the source `std::vector` leave scope.

```
VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
{
  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);

  return inputArray;
  // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
  // memory. However, inputArray has a pointer to that memory, which becomes an
  // invalid pointer in the returned object. Bad things will happen when the
  // ArrayHandle is used.
}

VTKM_CONT_EXPORT
vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad()
{
  std::vector<vtkm::Float32> dataBuffer;
```

```
21   // Populate dataBuffer with meaningful data. Perhaps read data from a file.
22
23   vtkm::cont::ArrayHandle<vtkm::Float32> tmpArray =
24       vtkm::cont::make_ArrayHandle(dataBuffer);
25
26   // This copies the data from one ArrayHandle to another (in the execution
27   // environment). Although it is an extraneous copy, it is usually pretty fast
28   // on a parallel device. Another option is to make sure that the buffer in
29   // the std::vector never goes out of scope before all the ArrayHandle
30   // references, but this extra step allows the ArrayHandle to manage its own
31   // memory and ensure everything is valid.
32   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray;
33   vtkm::cont::DeviceAdapterAlgorithm<VTKM_DEFAULT_DEVICE_ADAPTER_TAG>::Copy(
34       tmpArray, inputArray);
35
36   return inputArray;
37   // This is safe.
38 }
```

6.2 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` A `typedef` of 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 6.6: A simple array portal implementation.

```
1  template<typename T>
2  class SimpleScalarArrayPortal
3  {
4  public:
5    typedef T ValueType;
6
7    // There is no specification for creating array portals, but they generally
8    // need a constructor like this to be practical.
9    VTKM_EXEC_CONT_EXPORT
10   SimpleScalarArrayPortal(ValueType *array, vtkm::Id numberOfValues)
11     : Array(array), NumberOfValues(numberOfValues) {  }
12
13   VTKM_EXEC_CONT_EXPORT
14   SimpleScalarArrayPortal() : Array(NULL), NumberOfValues(0) {  }
```

```
  VTKM_EXEC_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }

  VTKM_EXEC_CONT_EXPORT
  ValueType Get(vtkm::Id index) const { return this->Array[index]; }

  VTKM_EXEC_CONT_EXPORT
  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` A `typedef` 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 6.7: Using `ArrayPortalToIterators`.

```
template<typename PortalType>
VTKM_CONT_EXPORT
std::vector<typename PortalType::ValueType>
CopyArrayPortalToVector(const PortalType &portal)
{
  typedef typename PortalType::ValueType ValueType;
  std::vector<ValueType> result(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 `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd` that each take an array portal as an argument and return a begin and end iterator, respectively.

Example 6.8: Using `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd`.

```
  std::vector<vtkm::Float32> myContainer(portal.GetNumberOfValues());

  std::copy(vtkm::cont::ArrayPortalToIteratorBegin(portal),
            vtkm::cont::ArrayPortalToIteratorEnd(portal),
            myContainer.begin());
```

ArrayHandle contains two typedefs 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 typedefs for array portals in the execution environment. Because these types are dependent on the device adapter used for execution, these typedefs are embedded in a templated class named ExecutionTypes. Within ExecutionTypes are the typedefs 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, so this can be a way of synchronizing the data. Be aware that when an ArrayHandle is created with a pointer or std::vector, it is put in a read-only mode, and GetPortalControl can fail (although GetPortalConstControl will still work). Also 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 6.9: Using portals from an ArrayHandle.

```
1  template<typename T>
2  void SortCheckArrayHandle(vtkm::cont::ArrayHandle<T> arrayHandle)
3  {
4    typedef typename vtkm::cont::ArrayHandle<T>::PortalControl
5        PortalType;
6    typedef typename vtkm::cont::ArrayHandle<T>::PortalConstControl
7        PortalConstType;
8
9    PortalType readwritePortal = arrayHandle.GetPortalControl();
10   // This is actually pretty dumb. Sorting would be generally faster in
11   // parallel in the execution environment using the device adapter algorithms.
12   std::sort(vtkm::cont::ArrayPortalToIteratorBegin(readwritePortal),
13             vtkm::cont::ArrayPortalToIteratorEnd(readwritePortal));
14
15   PortalConstType readPortal = arrayHandle.GetPortalConstControl();
16   for (vtkm::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
17   {
18     if (readPortal.Get(index-1) > readPortal.Get(index))
19     {
20       std::cout << "Sorting is wrong!" << std::endl;
21       break;
22     }
23   }
24 }
```

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.

6.3 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 Allocate method. The Allocate method takes a single argument, which is the number of elements to make the array.

Example 6.10: 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 `GetPortalControl`, as described in Section 6.2. This is roughly the method used by the readers in the I/O package (Chapter 2).

Example 6.11: Populating a newly allocated ArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;

const vtkm::Id ARRAY_SIZE = 50;
arrayHandle.Allocate(ARRAY_SIZE);

typedef vtkm::cont::ArrayHandle<vtkm::Float32>::PortalControl PortalType;
PortalType portal = arrayHandle.GetPortalControl();

for (vtkm::Id index = 0; index < ARRAY_SIZE; index++)
{
  portal.Set(index, GetValueForArray(index));
}
```

6.4 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.

`PrepareForInput` Copies data from the control to the execution environment, if necessary, and readies the data for read-only access.

`PrepareForInPlace` Copies the data from the control to the execution environment, if necessary, and readies the data for both reading and writing.

`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 7.1 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. Typically this is done with a call to Schedule in vtkm::cont::DeviceAdapterAlgorithm. This and other device adapter algorithms are described in detail in Section 7.2, but here is a quick example of using these execution array portals in a simple functor.

Example 6.12: Using an execution array portal from an ArrayHandle.

```
template<typename T, typename Device>
struct DoubleFunctor : public vtkm::exec::FunctorBase
{
  typedef typename vtkm::cont::ArrayHandle<T>::
      template ExecutionTypes<Device>::PortalConst InputPortalType;
  typedef typename vtkm::cont::ArrayHandle<T>::
      template ExecutionTypes<Device>::Portal OutputPortalType;

  VTKM_CONT_EXPORT
  DoubleFunctor(InputPortalType inputPortal, OutputPortalType outputPortal)
    : InputPortal(inputPortal), OutputPortal(outputPortal) {  }

  VTKM_EXEC_EXPORT
  void operator()(vtkm::Id index) const {
    this->OutputPortal.Set(index, 2*this->InputPortal.Get(index));
  }

  InputPortalType InputPortal;
  OutputPortalType OutputPortal;
};

template<typename T, typename Device>
void DoubleArray(vtkm::cont::ArrayHandle<T> inputArray,
                 vtkm::cont::ArrayHandle<T> outputArray,
                 Device)
{
  vtkm::Id numValues = inputArray.GetNumberOfValues();

  DoubleFunctor<T, Device> functor(
        inputArray.PrepareForInput(Device()),
        outputArray.PrepareForOutput(numValues, Device()));

  vtkm::cont::DeviceAdapterAlgorithm<Device>::Schedule(functor, numValues);
}
```

It should be noted that the array handle will expect any use of the execution array portal to finish before the next call to any ArrayHandle method. Since these Prepare methods are typically used in the process of scheduling an algorithm in the execution environment, this is seldom an issue.

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` *each time the data are accessed in the execution environment.*

DRAFT

CHAPTER

SEVEN

DEVICE ADAPTERS

DRAFT

As multiple vendors vie to provide accelerator-type processors, a great variance in the computer architecture exists. Likewise, there exist multiple compiler environments and libraries for these devices such as CUDA, OpenMP, and various threading libraries. These compiler technologies also vary from system to system.

To make porting among these systems at all feasible, we require a base language support, and the language we use is C++. The majority of the code in VTK-m is constrained to the standard C++ language constructs to minimize the specialization from one system to the next.

Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a *device adapter*. Thus, porting VTK-m to a new architecture can be done by adding only a device adapter.

The device adapter is shown diagrammatically as the connection between the control and execution environments in Figure 5.1 on page 16. The functionality of the device adapter comprises two main parts: a collection of parallel algorithms run in the execution environment and a module to transfer data between the control and execution environments.

This chapter describes how tags are used to specify which devices to use for operations within VTK-m. The chapter also outlines the features provided by a device adapter that allow you to directly control a device. Finally, we document how to create a new device adapter to port or specialize VTK-m for a different device or system.

7.1 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.

There are two ways to select a device adapter. The first is to make a global selection of a default device adapter. The second is to specify a specific device adapter as a template parameter.

7.1.1 Default Device Adapter

A default device adapter tag is specified in `vtkm/cont/DeviceAdapter.h` (although it can also by specified in many other VTK-m headers via header dependencies). If no other information is given, VTK-m attempts to choose a default device adapter that is a best fit for the system it is compiled on. VTK-m currently select the default device adapter with the following sequence of conditions.

- If the source code is being compiled by CUDA, the CUDA device is used.

- If the CUDA compiler is not being used and the current compiler supports OpenMP, then the OpenMP device is used. [TECHNICALLY, OPENMP IS NOT YET SUPPORTED IN VTK-M, SO THIS WILL NEVER ACTUALLY BE PICKED. BUT ONCE IT IS IMPLEMENTED, THIS WILL BE THE CHAIN.]
- If the compiler supports neither CUDA nor OpenMP and VTK-m was configured to use Intel Threading Building Blocks, then that device is used.
- If no parallel device adapters are found, then VTK-m falls back to a serial device.

You can also set the default device adapter specifically by setting the `VTKM_DEVICE_ADAPTER` macro. This macro must be set *before* including any VTK-m files. You can set `VTKM_DEVICE_ADAPTER` to any one of the following.

`VTKM_DEVICE_ADAPTER_SERIAL` Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available.

`VTKM_DEVICE_ADAPTER_CUDA` 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.

`VTKM_DEVICE_ADAPTER_OPENMP` 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. [NOT CURRENTLY IMPLEMENTED.]

`VTKM_DEVICE_ADAPTER_TBB` 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.

These macros provide a useful mechanism for quickly reconfiguring code to run on different devices. The following example shows a typical block of code at the top of a source file that could be used for quick reconfigurations.

Example 7.1: Macros to port VTK-m code among different devices

```
1 // Uncomment one (and only one) of the following to reconfigure the Dax
2 // code to use a particular device. Comment them all to automatically pick a
3 // device.
4 #define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_SERIAL
5 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_CUDA
6 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_OPENMP
7 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_TBB
8
9 #include <vtkm/cont/DeviceAdapter.h>
```

Did you know?

Filters do not actually use the default device adapter tag. They have a more sophisticated device selection mechanism that determines the devices available at runtime and will attempt running on multiple devices.

The default device adapter can always be overridden by specifying a device adapter tag, as described in the next section. There is actually one more internal default device adapter named `VTKM_DEVICE_ADAPTER_ERROR` that will cause a compile error if any component attempts to use the default device adapter. This feature is most often used in testing code to check when device adapters should be specified.

7.1.2 Specifying Device Adapter Tags

In addition to setting a global default device adapter, it is possible to explicitly set which device adapter to use in any feature provided by VTK-m. This is done by providing a device adapter tag as a template argument to VTK-m templated objects. 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/openmp/cont/DeviceAdapterOpenMP.h. [Not currently implemented.]

`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 prepare an output array for that device. In this case, the device adapter tag is passed as a parameter for the `PrepareForOutput` method of `vtkm::cont::ArrayHandle`.

Example 7.2: Specifying a device using a device adapter tag.

```
1   arrayHandle.PrepareForOutput(50, vtkm::cont::DeviceAdapterTagTBB());
```

When structuring your code to always specify a particular device adapter, consider setting the default device adapter (with the `VTKM_DEVICE_ADAPTER` macro) to `VTKM_DEVICE_ADAPTER_ERROR`. This will cause the compiler to produce an error if any object is instantiated with the default device adapter, which checks to make sure the code properly specifies every device adapter parameter.

VTK-m also defines a macro named `VTKM_DEFAULT_DEVICE_ADAPTER_TAG`, which can be used in place of an explicit device adapter tag to use the default tag. This macro is used to create new templates that have template parameters for device adapters that can use the default. The following example defines a functor to be used with the `Schedule` operation (to be described later) that is templated on the device it uses.

Example 7.3: Specifying a default device for template parameters.

```
1  template<typename Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG>
2  struct SetPortalFunctor : vtkm::exec::FunctorBase
3  {
4    VTKM_IS_DEVICE_ADAPTER_TAG(Device);
5
6    typedef typename vtkm::cont::ArrayHandle<vtkm::Id>::
7        ExecutionTypes<Device>::Portal ExecPortalType;
8    ExecPortalType Portal;
9
10   VTKM_CONT_EXPORT
11   SetPortalFunctor(vtkm::cont::ArrayHandle<vtkm::Id> array, vtkm::Id size)
12     : Portal(array.PrepareForOutput(size, Device()))
13   {  }
14
15   VTKM_EXEC_EXPORT
```

DRAFT

```
16    void operator()(vtkm::Id index) const
17    {
18      typedef typename ExecPortalType::ValueType ValueType;
19      this->Portal.Set(index, TestValue(index, ValueType()));
20    }
21  };
```

Common Errors

A device adapter tag is a class just like every other type in C++. Thus it is possible to accidently use a type that is not a device adapter tag when one is expected. This leads to unexpected errors in strange parts of the code. To help identify these errors, it is good practice to use the `VTKM_IS_DEVICE_ADAPTER_TAG` *macro to verify the type is a valid device adapter tag. Example 7.3 uses this macro on line 4.*

7.2 Device Adapter Algorithms

VTK-m comes with the templated class `vtkm::cont::DeviceAdapterAlgorithm` that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. The template has a single argument that specifies the device adapter tag.

Example 7.4: Prototype for `vtkm::cont::DeviceAdapterAlgorithm`.

```
1  namespace vtkm {
2  namespace cont {
3
4  template<typename DeviceAdapterTag>
5  struct DeviceAdapterAlgorithm;
6
7  }
8  } // namespace vtkm::cont
```

`DeviceAdapterAlgorithm` 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* `Copy` *method.*

`Copy` Copies data from an input array to an output array. The copy takes place in the execution environment.

`LowerBounds` The `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 `LowerBounds`. 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.

`Reduce` The `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.

`ReduceByKey` The `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.

`ScanInclusive` The `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.

`ScanExclusive` The `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`: one performs the sum using addition whereas the other accepts a custom binary function to use as the sum operator and a custom initial value to use in the running sum.

`Schedule` The `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.

`Sort` The `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.

`SortByKey` The `SortByKey` method works similarly to the `Sort` method except that it takes two `ArrayHandle`s: 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.

`StreamCompact` The `StreamCompact` method selectively removes values from an array. The first argument is an `ArrayHandle` to be compacted and the second argument 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

DRAFT

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.

There is also a second form of StreamCompact that only has the stencil and output as arguments. In this version, the output gets the corresponding index of where the input should be taken from.

Synchronize The Synchronize method waits for any asynchronous operations running on the device to complete and then returns.

Unique The 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.

UpperBounds The 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::Ids and the second is an ArrayHandle of vtkm::Ids 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.

7.3 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 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 and files associated with the Intel Threading Building Blocks (TBB) are located in vtkm/cont/tbb. The documentation here assumes that you are adding a device adapter to the VTK-m source code and following these file conventions. However, it is also possible to define a device adapter outside of the core VTK-m, in which case the file paths might be different.

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 3 header files with the names DeviceAdapterTag∗.h, ArrayManagerExecution∗.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 three 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 7.5: Contents of the base header for a device adapter.

```
1 #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2 #include <vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h>
3 #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.

7.3.1 Tag

The device adapter tag, as described in Section 7.1 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 7.6: Implementation of a device adapter tag.

```
1 #include <vtkm/cont/internal/DeviceAdapterTag.h>
2
3 // If this device adapter were to be contributed to VTK-m, then this macro
4 // declaration should be moved to DeviceAdapterTag.h and given a unique
5 // number.
6 #define VTKM_DEVICE_ADAPTER_CXX11_THREAD 101
7
8 VTKM_VALID_DEVICE_ADAPTER(Cxx11Thread, VTKM_DEVICE_ADAPTER_CXX11_THREAD);
```

7.3.2 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. The execution array manager is typically defined in an internal header file with a prefix of ArrayManagerExecution.

Example 7.7: Prototype for vtkm::cont::internal::ArrayManagerExecution.

```
1 namespace vtkm {
2 namespace cont {
3 namespace internal {
4
5 template<typename T, typename StorageTag, typename DeviceAdapterTag>
6 class ArrayManagerExecution;
7
8 }
9 }
10 } // namespace vtkm::cont::internal
```

A device adapter must provide a partial specialization of 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` A `typedef` of 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 7.8: Specialization of `ArrayManagerExecution`.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecution.h>
#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

```

DRAFT

```
6 namespace vtkm {
7 namespace cont {
8 namespace internal {
9
10 template<typename T, typename StorageTag>
11 class ArrayManagerExecution<
12       T, StorageTag, vtkm::cont::DeviceAdapterTagCxx11Thread>
13     : public vtkm::cont::internal::ArrayManagerExecutionShareWithControl<
14         T, StorageTag>
15 {
16   typedef vtkm::cont::internal::ArrayManagerExecutionShareWithControl
17       <T, StorageTag> Superclass;
18
19 public:
20   VTKM_CONT_EXPORT
21   ArrayManagerExecution(typename Superclass::StorageType *storage)
22     : Superclass(storage) {  }
23 };
24
25 }
26 }
27 } // namespace vtkm::cont::internal
```

7.3.3 Algorithms

A device adapter implementation must also provide a specialization of `vtkm::cont::DeviceAdapterAlgorithm`, which is documented in Section 7.2. 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.

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 Section 14.9, 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 7.9: Minimal specialization of `DeviceAdapterAlgorithm`.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/DeviceAdapterAlgorithm.h>
4  #include <vtkm/cont/internal/DeviceAdapterAlgorithmGeneral.h>
5
6  #include <thread>
7
8  namespace vtkm {
9  namespace cont {
10
11 template<>
12 struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
13     : vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
14           DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>,
15           vtkm::cont::DeviceAdapterTagCxx11Thread>
16 {
17 private:
18   template<typename FunctorType>
19   struct ScheduleKernel1D
20   {
21     VTKM_CONT_EXPORT
22     ScheduleKernel1D(const FunctorType &functor)
23       : Functor(functor)
24     {  }
25
26     VTKM_EXEC_EXPORT
27     void operator()() const
28     {
29       try
30       {
31         for (vtkm::Id threadId = this->BeginId;
32              threadId < this->EndId;
33              threadId++)
34         {
35           this->Functor(threadId);
36           // If an error is raised, abort execution.
37           if (this->ErrorMessage.IsErrorRaised()) { return; }
38         }
39       }
40       catch (vtkm::cont::Error error)
41       {
42         this->ErrorMessage.RaiseError(error.GetMessage().c_str());
43       }
44       catch (std::exception error)
45       {
46         this->ErrorMessage.RaiseError(error.what());
47       }
```

```
48      catch (...)
49      {
50        this->ErrorMessage.RaiseError("Unknown exception raised.");
51      }
52    }
53
54    FunctorType Functor;
55    vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
56    vtkm::Id BeginId;
57    vtkm::Id EndId;
58  };
59
60  template<typename FunctorType>
61  struct ScheduleKernel3D
62  {
63    VTKM_CONT_EXPORT
64    ScheduleKernel3D(const FunctorType &functor, vtkm::Id3 maxRange)
65      : Functor(functor), MaxRange(maxRange)
66    {  }
67
68    VTKM_EXEC_EXPORT
69    void operator()() const
70    {
71      vtkm::Id3 threadId3D(this->BeginId%this->MaxRange[0],
72                           (this->BeginId/this->MaxRange[0])%this->MaxRange[1],
73                           this->BeginId/(this->MaxRange[0]*this->MaxRange[1]));
74
75      try
76      {
77        for (vtkm::Id threadId = this->BeginId;
78             threadId < this->EndId;
79             threadId++)
80        {
81          this->Functor(threadId3D);
82          // If an error is raised, abort execution.
83          if (this->ErrorMessage.IsErrorRaised()) { return; }
84
85          threadId3D[0]++;
86          if (threadId3D[0] >= MaxRange[0])
87          {
88            threadId3D[0] = 0;
89            threadId3D[1]++;
90            if (threadId3D[1] >= MaxRange[1])
91            {
92              threadId3D[1] = 0;
93              threadId3D[2]++;
94            }
95          }
96        }
97      }
98      catch (vtkm::cont::Error error)
99      {
100       this->ErrorMessage.RaiseError(error.GetMessage().c_str());
101     }
102     catch (std::exception error)
103     {
104       this->ErrorMessage.RaiseError(error.what());
105     }
106     catch (...)
107     {
108       this->ErrorMessage.RaiseError("Unknown exception raised.");
109     }
110   }
111
```

```
112      FunctorType Functor;
113      vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
114      vtkm::Id BeginId;
115      vtkm::Id EndId;
116      vtkm::Id3 MaxRange;
117    };
118
119    template<typename KernelType>
120    VTKM_CONT_EXPORT
121    static void DoSchedule(KernelType kernel,
122                           vtkm::Id numInstances)
123    {
124      if (numInstances < 1) { return; }
125
126      const vtkm::Id MESSAGE_SIZE = 1024;
127      char errorString[MESSAGE_SIZE];
128      errorString[0] = '\0';
129      vtkm::exec::internal::ErrorMessageBuffer errorMessage(errorString,
130                                                            MESSAGE_SIZE);
131      kernel.Functor.SetErrorMessageBuffer(errorMessage);
132      kernel.ErrorMessage = errorMessage;
133
134      vtkm::Id numThreads =
135          static_cast<vtkm::Id>(std::thread::hardware_concurrency());
136      if (numThreads > numInstances)
137      {
138        numThreads = numInstances;
139      }
140      vtkm::Id numInstancesPerThread = (numInstances+numThreads-1)/numThreads;
141
142      std::thread *threadPool = new std::thread[numThreads];
143      vtkm::Id beginId = 0;
144      for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
145      {
146        vtkm::Id endId = std::min(beginId+numInstancesPerThread, numInstances);
147        KernelType threadKernel = kernel;
148        threadKernel.BeginId = beginId;
149        threadKernel.EndId = endId;
150        std::thread newThread(threadKernel);
151        threadPool[threadIndex].swap(newThread);
152        beginId = endId;
153      }
154
155      for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
156      {
157        threadPool[threadIndex].join();
158      }
159
160      delete[] threadPool;
161
162      if (errorMessage.IsErrorRaised())
163      {
164        throw vtkm::cont::ErrorExecution(errorString);
165      }
166    }
167
168  public:
169    template<typename FunctorType>
170    VTKM_CONT_EXPORT
171    static void Schedule(FunctorType functor, vtkm::Id numInstances)
172    {
173      DoSchedule(ScheduleKernel1D<FunctorType>(functor), numInstances);
174    }
175
```

```
176   template<typename FunctorType>
177   VTKM_CONT_EXPORT
178   static void Schedule(FunctorType functor, vtkm::Id3 maxRange)
179   {
180     vtkm::Id numInstances = maxRange[0]*maxRange[1]*maxRange[2];
181     DoSchedule(ScheduleKernel3D<FunctorType>(functor, maxRange), numInstances);
182   }
183
184   VTKM_CONT_EXPORT
185   static void Synchronize()
186   {
187     // Nothing to do. This device schedules all of its operations using a
188     // split/join paradigm. This means that the if the control threaad is
189     // calling this method, then nothing should be running in the execution
190     // environment.
191   }
192 };
193
194 }
195 } // namespace vtkm::cont
```

7.3.4 Timer Implementation

The VTK-m timer, described in Section ??, 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 7.10: Specialization of `DeviceAdapterTimerImplementation`.

```
1  #include <chrono>
2
3  namespace vtkm {
4  namespace cont {
5
6  template<>
7  class DeviceAdapterTimerImplementation<vtkm::cont::DeviceAdapterTagCxx11Thread>
8  {
9  public:
10   VTKM_CONT_EXPORT
11   DeviceAdapterTimerImplementation()
12   {
13     this->Reset();
14   }
15
16   VTKM_CONT_EXPORT
17   void Reset()
18   {
19     vtkm::cont::DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
```

```
        ::Synchronize();
    this->StartTime = std::chrono::high_resolution_clock::now();
  }

  VTKM_CONT_EXPORT
  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 vtkm::cont
```

CHAPTER

EIGHT

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. In the VTK-m control environment timing is simplified because the control environment operates on a single thread. However, operations invoked in the execution environment may run asynchronously to operations in the control environment.

To ensure that accurate timings can be made, VTK-m provides a `vtkm::cont::Timer` class that is templated on the device adapter to provide an accurate measurement of operations that happen on the device. If not template parameter is provided, the default device adapter is used.

The timing starts when the `Timer` is constructed. The time elapsed can be retrieved with a call to the `GetElapsedTime` method. This method will block until all operations in the execution environment complete so as to return an accurate time. The timer can be restarted with a call to the `Reset` method.

Example 8.1: Using `vtkm::cont::Timer`.

```
1  vtkm::filter::PointElevation elevationFilter;
2
3  vtkm::cont::Timer<> timer;
4
5  vtkm::filter::FieldResult fieldResult =
6      elevationFilter.Execute(dataSet, dataSet.GetCoordinateSystem());
7
8  // This code makes sure data is pulled back to the host in a host/device
9  // architecture.
10 vtkm::cont::ArrayHandle<vtkm::Float64> outArray;
11 fieldResult.FieldAs(outArray);
12 outArray.GetPortalConstControl();
13
14 vtkm::Float64 elapsedTime = timer.GetElapsedTime();
15
16 std::cout << "Time to run: " << elapsedTime << std::endl;
```

Common Errors

Make sure the `Timer` being used is match to the device adapter used for the computation. This will ensure that the parallel computation is synchronized.

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.

DRAFT

CHAPTER

NINE

FANCY ARRAY STORAGE

Chapter 6 introduces the `vtkm::cont::ArrayHandle` class. In it, 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 9.1.

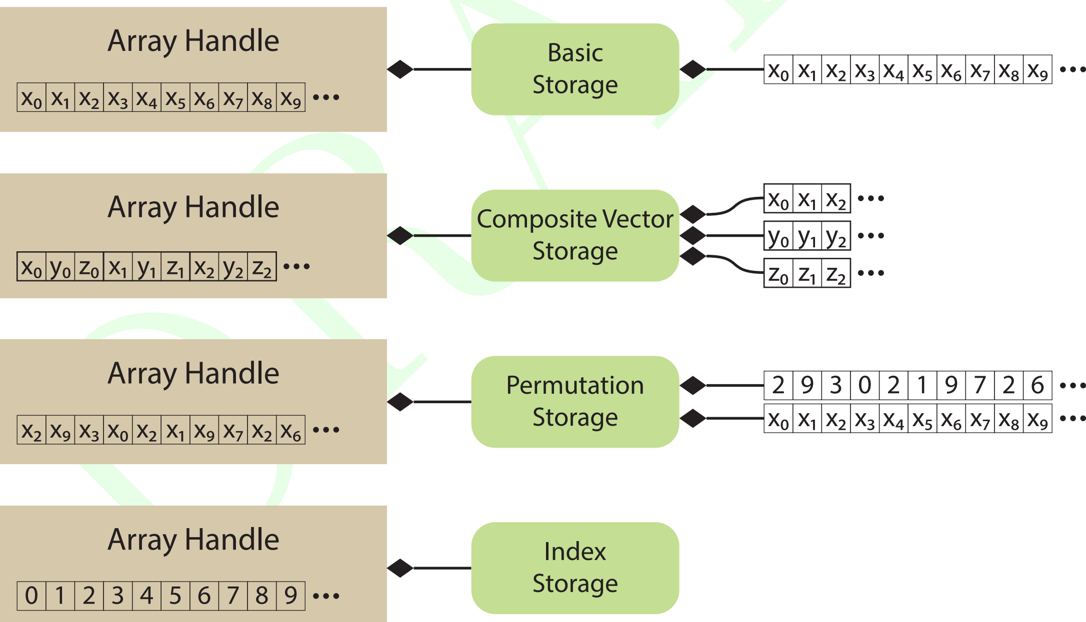


Figure 9.1: Array handles, storage objects, and the underlying data source.

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

DRAFT

the ArrayHandle storage.

Did you know?

VTK-m comes with many "fancy" array handles that can change the data in other arrays without modifying the memory or can generate data on the fly to behave like an array without actually using any memory. These fancy array handles are documented later in this chapter, and they can be very handy when developing with VTK-m.

9.1 Basic Storage

As previously discussed in Chapter 6, vtkm::cont::ArrayHandle takes two template arguments.

Example 9.1: 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.

In this and the following sections we describe this storage mechanism. A default storage is specified in much the same way as a default device adapter is defined (as described in Section 7.1.1. It is done by setting the VTKM_STORAGE macro. This macro must be set before including any VTK-m header files. Currently the only practical storage provided by VTK-m is the basic storage, which simply allocates a continuous section of memory of the given base type. This storage can be explicitly specified by setting VTKM_STORAGE to VTKM_STORAGE_BASIC although the basic storage will also be used as the default if no other storage is specified (which is typical).

The default storage can always be overridden by specifying an array storage tag. The tag for the basic storage is located in the vtkm/cont/StorageBasic.h header file and is named vtkm::cont::StorageTagBasic. Here is an example of specifying the storage type when declaring an array handle.

Example 9.2: Specifying the storage type for an ArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32,vtkm::cont::StorageTagBasic> arrayHandle;
```

VTK-m also defines a macro named VTKM_DEFAULT_STORAGE_TAG that can be used in place of an explicit storage tag to use the default tag. This macro is used to create new templates that have template parameters for storage that can use the default.

9.2 Provided Fancy Arrays

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 do many other operations. 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. This modified behavior for arrays is called "fancy" arrays.

VTK-m provides many of the fancy arrays, which we explore in this section. Later Section 9.3 describes many different ways in which new fancy arrays can be implemented.

9.2.1 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 9.3: Using `ArrayHandleConstant`.

```
1  // Create an array of 50 entries, all containing the number 3. This could be
2  // used, for example, to represent the sizes of all the polygons in a set
3  // where we know all the polygons are triangles.
4  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 9.4: Using `make_ArrayHandleConstant`.

```
1  // Create an array of 50 entries, all containing the number 3.
2  vtkm::cont::make_ArrayHandleConstant(3, 50)
```

9.2.2 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 9.5: Using `ArrayHandleIndex`.

```
1  // Create an array containing [0, 1, 2, 3, ..., 49].
2  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 9.6: Using `ArrayHandleCounting`.

```
1  // Create an array containing [-1.0, -0.9, -0.8, ..., 0.9, 1.0]
2  vtkm::cont::ArrayHandleCounting<vtkm::Float32> sampleArray(-1.0f, 0.1f, 21);
```

DRAFT

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 9.7: Using `make_ArrayHandleCounting`.

```
1  // Create an array of 50 entries, all containing the number 3.
2  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 9.8: Counting backwards with `ArrayHandleCounting`.

```
1  // Create an array containing [49, 48, 47, 46, ..., 0].
2  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 9.9: Using `ArrayHandleCounting` with `vtkm::Vec` objects.

```
1  // Create an array containg [(0,-3,75), (1,2,25), (3,7,-25)]
2  vtkm::cont::make_ArrayHandleCounting(vtkm::make_Vec(0, -3, 75),
3                                       vtkm::make_Vec(1, 5, -50),
4                                       3)
```

9.2.3 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 9.10: Using `ArrayHandleCast`.

```
1  template<typename T>
2  VTKM_CONT_EXPORT
3  void Foo(const std::vector<T> &inputData)
4  {
5    vtkm::cont::ArrayHandle<T> originalArray =
6        vtkm::cont::make_ArrayHandle(inputData);
7
8    vtkm::cont::ArrayHandleCast<vtkm::Float64, vtkm::cont::ArrayHandle<T> >
9        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 9.11: Using make_ArrayHandleCast.

```
vtkm::cont::make_ArrayHandleCast<vtkm::Float64>(originalArray)
```

9.2.4 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 9.12: Using ArrayHandlePermutation.

```
typedef vtkm::cont::ArrayHandle<vtkm::Id> IdArrayType;
typedef IdArrayType::PortalControl IdPortalType;

typedef vtkm::cont::ArrayHandle<vtkm::Float32> ValueArrayType;
typedef ValueArrayType::PortalControl ValuePortalType;

// 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 9.13: 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.*

9.2.5 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 9.14: Using `ArrayHandleZip`.

```
typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType1;
typedef ArrayType1::PortalControl PortalType1;

typedef vtkm::cont::ArrayHandle<vtkm::Float32> ArrayType2;
typedef ArrayType2::PortalControl PortalType2;

// 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::Float32> 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 9.15: Using make_ArrayHandleZip.

```
vtkm::cont::make_ArrayHandleZip(array1,array2)
```

9.2.6 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 11, 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 9.16: 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::make_Vec<vtkm::FloatDefault>(-5.0f, -3.0f, -3.0f),
      vtkm::make_Vec<vtkm::FloatDefault>(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 9.17: Using a ArrayHandleCartesianProduct.

```
typedef vtkm::cont::ArrayHandle<vtkm::Float32> AxisArrayType;
typedef AxisArrayType::PortalControl AxisPortalType;

// 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 9.18: 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.

9.2.7 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 9.2.5) except that values are combined into vtkm::Vecs instead of vtkm::Pairs. The created vtkm::Vecs 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 single template argument that is a "signature" for the arrays to be combined. These signatures can be tricky to prototype, so vtkm/cont/ArrayHandleCompositeVector.h/header also contains a helper struct named vtkm::cont::ArrayHandleCompositeVectorType to define the type. ArrayHandleCompositeVectorType takes a variable number of ArrayHandle types that compose the vector. ArrayHandleCompositeVectorType has an internal type named type that is the appropriately defined ArrayHandleCompositeVector.

The constructor for ArrayHandleCompositeVector takes instances of the array handles to combine along with

the component from each array to use. If the array handles being combined contain scalar data, then the appropriate component to use is 0.

Example 9.19: Using ArrayHandleCompositeVector.

```
// Create an array with [0, 1, 2, 3, 4]
typedef vtkm::cont::ArrayHandleIndex ArrayType1;
ArrayType1 array1(5);

// Create an array with [3, 1, 4, 1, 5]
typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType2;
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]
typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType3;
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]
typedef vtkm::cont::ArrayHandleConstant<vtkm::Id> ArrayType4;
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)].
typedef vtkm::cont::ArrayHandleCompositeVectorType<
    ArrayType1, ArrayType2, ArrayType3, ArrayType4>::type CompositeArrayType;
CompositeArrayType compositeArray(array1, 0,
                                  array2, 0,
                                  array3, 0,
                                  array4, 0);
```

The vtkm/cont/ArrayHandleCompositeVector.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleCompositeVector which takes two to four array handles and returns an ArrayHandleCompositeVector. This function can sometimes be used to avoid having to declare the full array type.

Example 9.20: Using make_ArrayHandleCompositeVector.

```
vtkm::cont::make_ArrayHandleCompositeVector(array1, 0,
                                            array2, 0,
                                            array3, 0,
                                            array4, 0)
```

ArrayHandleCompositeVector is often used to combine scalar arrays into vector arrays, but it can also be used to pull components out of other vector arrays. The following example uses this feature to convert an array of 2D x,y coordinates and an array of elevations to 3D x,y,z coordinates.

Example 9.21: Combining vector components with ArrayHandleCompositeVector.

```
template<typename CoordinateArrayType, typename ElevationArrayType>
VTKM_CONT_EXPORT
typename vtkm::cont::ArrayHandleCompositeVectorType<
```

```
    CoordinateArrayType, CoordinateArrayType, ElevationArrayType>::type
ElevateCoordianteArray(const CoordinateArrayType &coordinateArray,
                       const ElevationArrayType &elevationArray)
{
  VTKM_IS_ARRAY_HANDLE(CoordinateArrayType);
  VTKM_IS_ARRAY_HANDLE(ElevationArrayType);

  return vtkm::cont::make_ArrayHandleCompositeVector(coordinateArray, 0,
                                                     coordinateArray, 1,
                                                     elevationArray, 0);
}
```

9.2.8 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 `Vec`s are generated as the array is used. Writing `Vec`s 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 `Vec`s to create (the number of values to group together).

Example 9.22: Using `ArrayHandleGroupVec`.

```
  // Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
  typedef vtkm::cont::ArrayHandleIndex ArrayType;
  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 `Vec`s. You must specify the size of the `Vec`s as a template parameter when using `vtkm::cont::make_ArrayHandleGroupVec`.

Example 9.23: 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)
```

9.3 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.

9.3.1 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

DRAFT

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. 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 9.24: Functor that doubles an index.

```
1 struct DoubleIndexFunctor
2 {
3   VTKM_EXEC_CONT_EXPORT
4   vtkm::Id operator()(vtkm::Id index) const
5   {
6     return 2*index;
7   }
8 };
```

Once the functor is defined, an implicit array can be declared using the templated `vtkm::cont::ArrayHandleImplicit` class. The first template argument is the type of the array's values (which should match the return value for the functor), and the second template argument is the functor type.

Example 9.25: Declaring a `ArrayHandleImplicit`.

```
1   vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
2       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. When using this function, you also have to declare the first template argument, which is the array's value type, since this type does not appear in any of the arguments.

Example 9.26: Using `make_ArrayHandleImplicit`.

```
1   vtkm::cont::make_ArrayHandleImplicit<vtkm::Id>(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 9.27: Custom implicit array handle for even numbers.

```
1 #include <vtkm/cont/ArrayHandleImplicit.h>
2
```

DRAFT

```
class ArrayHandleDoubleIndex
    : public vtkm::cont::ArrayHandleImplicit<vtkm::Id, DoubleIndexFunctor>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
      ArrayHandleDoubleIndex,
      (vtkm::cont::ArrayHandleImplicit<vtkm::Id,DoubleIndexFunctor>));

  VTKM_CONT_EXPORT
  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 typedefs `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 9.27 is not templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. (The other macro is described in Section 9.3.2 on page 66). 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.

9.3.2 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 9.28: Functor to scale and bias a value.

```
template<typename T>
struct ScaleBiasFunctor
{
  VTKM_EXEC_CONT_EXPORT
  ScaleBiasFunctor(T scale = T(1), T bias = T(0))
    : Scale(scale), Bias(bias) {  }

  VTKM_EXEC_CONT_EXPORT
```

DRAFT

```
 9    T operator()(T x) const
10    {
11      return this->Scale*x + this->Bias;
12    }
13
14    T Scale;
15    T Bias;
16  };
```

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 the array's values (which should match the return value for the functor). The second template argument is the type of array being transformed. The third and final template argument is the type of functor used for the transformation.

That said, it is generally easier to use the vtkm::cont::make_ArrayHandleTransform convenience function. This function takes an array and a functor and returns a transformed array. When using this function, you also have to declare the first template argument, which is the transformed array's value type, since this type does not appear in any of the arguments.

Example 9.29: Using make_ArrayHandleTransform.

```
1   vtkm::cont::make_ArrayHandleTransform<vtkm::Float32>(
2           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 9.30: Custom transform array handle for scale and bias.

```
 1  #include <vtkm/cont/ArrayHandleTransform.h>
 2
 3  template<typename ArrayHandleType>
 4  class ArrayHandleScaleBias
 5      : public vtkm::cont::ArrayHandleTransform<
 6            typename ArrayHandleType::ValueType,
 7            ArrayHandleType,
 8            ScaleBiasFunctor<typename ArrayHandleType::ValueType> >
 9  {
10  public:
11    VTKM_ARRAY_HANDLE_SUBCLASS(
12        ArrayHandleScaleBias,
13        (ArrayHandleScaleBias<ArrayHandleType>),
14        (vtkm::cont::ArrayHandleTransform<
15           typename ArrayHandleType::ValueType,
16           ArrayHandleType,
17           ScaleBiasFunctor<typename ArrayHandleType::ValueType> >)
18        );
19
20    VTKM_CONT_EXPORT
21    ArrayHandleScaleBias(const ArrayHandleType &array,
22                         ValueType scale,
23                         ValueType bias)
24      : Superclass(array, ScaleBiasFunctor<ValueType>(scale, bias)) {  }
25  };
26
27  template<typename ArrayHandleType>
28  VTKM_CONT_EXPORT
29  ArrayHandleScaleBias<ArrayHandleType>
30  make_ArrayHandleScaleBias(const ArrayHandleType &array,
31                            typename ArrayHandleType::ValueType scale,
32                            typename ArrayHandleType::ValueType bias)
33  {
```

```
34    return ArrayHandleScaleBias<ArrayHandleType>(array, scale, bias);
35  }
```

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 typedefs 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 9.30 is templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS macro. (The other macro is described in Section 9.4 on page 77). 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.

9.3.3 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 9.3.2) 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 9.31: Derived array portal for concatenated arrays.

```
1   #include <vtkm/cont/ArrayHandle.h>
2   #include <vtkm/cont/ArrayPortal.h>
3
4   template<typename P1, typename P2>
5   class ArrayPortalConcatenate
6   {
7   public:
8     typedef P1 PortalType1;
9     typedef P2 PortalType2;
10    typedef typename PortalType1::ValueType ValueType;
11
12    VTKM_SUPPRESS_EXEC_WARNINGS
13    VTKM_EXEC_CONT_EXPORT
14    ArrayPortalConcatenate() : Portal1(), Portal2() {  }
15
16    VTKM_SUPPRESS_EXEC_WARNINGS
17    VTKM_EXEC_CONT_EXPORT
18    ArrayPortalConcatenate(const PortalType1 &portal1, const PortalType2 portal2)
19      : Portal1(portal1), Portal2(portal2) {  }
20
21    /// Copy constructor for any other ArrayPortalConcatenate with a portal type
22    /// that can be copied to this portal type. This allows us to do any type
23    /// casting that the portals do (like the non-const to const cast).
24    VTKM_SUPPRESS_EXEC_WARNINGS
```

DRAFT

```
25   template<typename OtherP1, typename OtherP2>
26   VTKM_EXEC_CONT_EXPORT
27   ArrayPortalConcatenate(const ArrayPortalConcatenate<OtherP1,OtherP2> &src)
28     : Portal1(src.GetPortal1()), Portal2(src.GetPortal2()) {  }
29 
30   VTKM_SUPPRESS_EXEC_WARNINGS
31   VTKM_EXEC_CONT_EXPORT
32   vtkm::Id GetNumberOfValues() const {
33     return
34         this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
35   }
36 
37   VTKM_SUPPRESS_EXEC_WARNINGS
38   VTKM_EXEC_CONT_EXPORT
39   ValueType Get(vtkm::Id index) const {
40     if (index < this->Portal1.GetNumberOfValues())
41     {
42       return this->Portal1.Get(index);
43     }
44     else
45     {
46       return this->Portal2.Get(index - this->Portal1.GetNumberOfValues());
47     }
48   }
49 
50   VTKM_SUPPRESS_EXEC_WARNINGS
51   VTKM_EXEC_CONT_EXPORT
52   void Set(vtkm::Id index, const ValueType &value) const {
53     if (index < this->Portal1.GetNumberOfValues())
54     {
55       this->Portal1.Set(index, value);
56     }
57     else
58     {
59       this->Portal2.Set(index - this->Portal1.GetNumberOfValues(), value);
60     }
61   }
62 
63   VTKM_EXEC_CONT_EXPORT
64   const PortalType1 &GetPortal1() const { return this->Portal1; }
65   VTKM_EXEC_CONT_EXPORT
66   const PortalType2 &GetPortal2() const { return this->Portal2; }
67 
68 private:
69   PortalType1 Portal1;
70   PortalType2 Portal2;
71 };
```

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 9.32: `Storage` for derived container of concatenated arrays.

```
1 template<typename ArrayHandleType1, typename ArrayHandleType2>
2 struct StorageTagConcatenate {  };
3 
4 namespace vtkm {
5 namespace cont {
6 namespace internal {
7 
8 template<typename ArrayHandleType1, typename ArrayHandleType2>
```

```
 9 class Storage<
10     typename ArrayHandleType1::ValueType,
11     StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2> >
12 {
13 public:
14   typedef typename ArrayHandleType1::ValueType ValueType;
15
16   typedef ArrayPortalConcatenate<
17       typename ArrayHandleType1::PortalControl,
18       typename ArrayHandleType2::PortalControl> PortalType;
19   typedef ArrayPortalConcatenate<
20       typename ArrayHandleType1::PortalConstControl,
21       typename ArrayHandleType2::PortalConstControl> PortalConstType;
22
23   VTKM_CONT_EXPORT
24   Storage() : Valid(false) {  }
25
26   VTKM_CONT_EXPORT
27   Storage(const ArrayHandleType1 array1, const ArrayHandleType2 array2)
28     : Array1(array1), Array2(array2), Valid(true) {  }
29
30   VTKM_CONT_EXPORT
31   PortalType GetPortal() {
32     VTKM_ASSERT(this->Valid);
33     return PortalType(this->Array1.GetPortalControl(),
34                       this->Array2.GetPortalControl());
35   }
36
37   VTKM_CONT_EXPORT
38   PortalConstType GetPortalConst() const {
39     VTKM_ASSERT(this->Valid);
40     return PortalConstType(this->Array1.GetPortalConstControl(),
41                            this->Array2.GetPortalConstControl());
42   }
43
44   VTKM_CONT_EXPORT
45   vtkm::Id GetNumberOfValues() const {
46     VTKM_ASSERT(this->Valid);
47     return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
48   }
49
50   VTKM_CONT_EXPORT
51   void Allocate(vtkm::Id numberOfValues) {
52     VTKM_ASSERT(this->Valid);
53     // This implementation of allocate, which allocates the same amount in both
54     // arrays, is arbitrary. It could, for example, leave the size of Array1
55     // alone and change the size of Array2. Or, probably most likely, it could
56     // simply throw an error and state that this operation is invalid.
57     vtkm::Id half = numberOfValues/2;
58     this->Array1.Allocate(numberOfValues-half);
59     this->Array2.Allocate(half);
60   }
61
62   VTKM_CONT_EXPORT
63   void Shrink(vtkm::Id numberOfValues) {
64     VTKM_ASSERT(this->Valid);
65     if (numberOfValues < this->Array1.GetNumberOfValues())
66     {
67       this->Array1.Shrink(numberOfValues);
68       this->Array2.Shrink(0);
69     }
70     else
71     {
72       this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
```

```
73      }
74    }
75
76    VTKM_CONT_EXPORT
77    void ReleaseResources() {
78      VTKM_ASSERT(this->Valid);
79      this->Array1.ReleaseResources();
80      this->Array2.ReleaseResources();
81    }
82
83    // Requried for later use in ArrayTransfer class.
84    VTKM_CONT_EXPORT
85    const ArrayHandleType1 &GetArray1() const {
86      VTKM_ASSERT(this->Valid);
87      return this->Array1;
88    }
89    VTKM_CONT_EXPORT
90    const ArrayHandleType2 &GetArray2() const {
91      VTKM_ASSERT(this->Valid);
92      return this->Array2;
93    }
94
95  private:
96    ArrayHandleType1 Array1;
97    ArrayHandleType2 Array2;
98    bool Valid;
99  };
100
101 }
102 }
103 } // namespace vtkm::cont::internal
```

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 9.33: Prototype for `vtkm::cont::internal::ArrayTransfer`.

```
1  namespace vtkm {
2  namespace cont {
3  namespace internal {
4
5  template<typename T,typename StorageTag,typename DeviceAdapterTag>
6  class ArrayTransfer;
7
8  }
9  }
10 } //namespace vtkm::cont::internal
```

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 9.33, the prototype for the constructor must be equivalent to the following.

Example 9.34: Prototype for ArrayTransfer constructor.

```
1 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` A `typedef` of 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.

DRAFT

`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 9.31 and 9.32, the following provides an `ArrayTransfer` appropriate for the derived storage.

Example 9.35: `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:
  typedef typename ArrayHandleType1::ValueType ValueType;

private:
  typedef StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2>
      StorageTag;
  typedef vtkm::cont::internal::Storage<ValueType,StorageTag>
      StorageType;

public:
  typedef typename StorageType::PortalType PortalControl;
  typedef typename StorageType::PortalConstType PortalConstControl;

  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::template ExecutionTypes<Device>::Portal,
      typename ArrayHandleType2::template ExecutionTypes<Device>::Portal>
    PortalExecution;
  typedef ArrayPortalConcatenate<
      typename ArrayHandleType1::template ExecutionTypes<Device>::PortalConst,
      typename ArrayHandleType2::template ExecutionTypes<Device>::PortalConst>
    PortalConstExecution;

  VTKM_CONT_EXPORT
  ArrayTransfer(StorageType *storage)
    : Array1(storage->GetArray1()), Array2(storage->GetArray2())
  {  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
  }

  VTKM_CONT_EXPORT
  PortalConstExecution PrepareForInput(bool vtkmNotUsed(updateData)) {
    return PortalConstExecution(this->Array1.PrepareForInput(Device()),
                                this->Array2.PrepareForInput(Device()));
  }

  VTKM_CONT_EXPORT
  PortalExecution PrepareForInPlace(bool vtkmNotUsed(updateData)) {
    return PortalExecution(this->Array1.PrepareForInPlace(Device()),
                           this->Array2.PrepareForInPlace(Device()));
  }

  VTKM_CONT_EXPORT
```

```
58    PortalExecution PrepareForOutput(vtkm::Id numberOfValues)
59    {
60      // This implementation of allocate, which allocates the same amount in both
61      // arrays, is arbitrary. It could, for example, leave the size of Array1
62      // alone and change the size of Array2. Or, probably most likely, it could
63      // simply throw an error and state that this operation is invalid.
64      vtkm::Id half = numberOfValues/2;
65      return PortalExecution(
66            this->Array1.PrepareForOutput(numberOfValues-half, Device()),
67            this->Array2.PrepareForOutput(half, Device()));
68    }
69
70    VTKM_CONT_EXPORT
71    void RetrieveOutputData(StorageType *vtkmNotUsed(storage)) const {
72      // Implementation of this method should be unnecessary. The internal
73      // array handles should automatically retrieve the output data as
74      // necessary.
75    }
76
77    VTKM_CONT_EXPORT
78    void Shrink(vtkm::Id numberOfValues) {
79      if (numberOfValues < this->Array1.GetNumberOfValues())
80      {
81        this->Array1.Shrink(numberOfValues);
82        this->Array2.Shrink(0);
83      }
84      else
85      {
86        this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
87      }
88    }
89
90    VTKM_CONT_EXPORT
91    void ReleaseResources() {
92      this->Array1.ReleaseResourcesExecution();
93      this->Array2.ReleaseResourcesExecution();
94    }
95
96  private:
97    ArrayHandleType1 Array1;
98    ArrayHandleType2 Array2;
99  };
100
101 }
102 }
103 } // namespace vtkm::cont::internal
```

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 9.36: `ArrayHandle` for derived storage of concatenated arrays.

```
1  template<typename ArrayHandleType1, typename ArrayHandleType2>
2  class ArrayHandleConcatenate
3      : public vtkm::cont::ArrayHandle<
4          typename ArrayHandleType1::ValueType,
5          StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2> >
6  {
7  public:
8    VTKM_ARRAY_HANDLE_SUBCLASS(
9        ArrayHandleConcatenate,
10       (ArrayHandleConcatenate<ArrayHandleType1,ArrayHandleType2>),
```

```
11        (vtkm::cont::ArrayHandle<
12           typename ArrayHandleType1::ValueType,
13           StorageTagConcatenate<ArrayHandleType1,ArrayHandleType2> >));
14
15 private:
16   typedef vtkm::cont::internal::Storage<ValueType,StorageTag> StorageType;
17
18 public:
19   VTKM_CONT_EXPORT
20   ArrayHandleConcatenate(const ArrayHandleType1 &array1,
21                          const ArrayHandleType2 &array2)
22     : Superclass(StorageType(array1, array2)) {  }
23 };
```

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 typedefs `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 9.36 is templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS` macro. (The other macro is described in Section 9.4 on page 77). 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.

9.4 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.

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 9.37: Fictitious field storage used in custom array storage examples.

```
#include <deque>

struct FooFields {
  float Pressure;
  float Temperature;
  float Velocity[3];
  // And so on...
};

typedef std::deque<FooFields> FooFieldsDeque;
```

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 6.2 and is generally straightforward for simple containers like this. Here is an example implementation for our `FooFieldsDeque` container.

Example 9.38: Array portal to adapt a third-party container to VTK-m.

```
#include <vtkm/cont/internal/IteratorFromArrayPortal.h>
#include <vtkm/Assert.h>

// DequeType expected to be either FooFieldsDeque or const FooFieldsDeque
template<typename DequeType>
class ArrayPortalFooPressure
{
public:
  typedef float ValueType;

  VTKM_CONT_EXPORT
  ArrayPortalFooPressure() : Container(NULL) {  }

  VTKM_CONT_EXPORT
  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_EXPORT
  ArrayPortalFooPressure(const ArrayPortalFooPressure<OtherDequeType> &other)
    : Container(other.GetContainer()) {  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT_EXPORT
  ValueType Get(vtkm::Id index) const {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    return (*this->Container)[index].Pressure;
  }

  VTKM_CONT_EXPORT
  void Set(vtkm::Id index, ValueType value) const {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    (*this->Container)[index].Pressure = value;
  }

  // Here for the copy constructor.
```

```
44    VTKM_CONT_EXPORT
45    DequeType *GetContainer() const { return this->Container; }
46
47  private:
48    DequeType *Container;
49  };
```

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 9.39: Prototype for vtkm::cont::internal::Storage.

```
1  namespace vtkm {
2  namespace cont {
3  namespace internal {
4
5  template<typename T, class StorageTag>
6  class Storage;
7
8  }
9  }
10 } // namespace vtkm::cont::internal
```

The vtkm::cont::internal::Storage must define the following items.

ValueType A **typedef** of 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 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. An 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 9.38.

Example 9.40: 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:
  typedef float ValueType;

  typedef ArrayPortalFooPressure<FooFieldsDeque> PortalType;
  typedef ArrayPortalFooPressure<const FooFieldsDeque> PortalConstType;

  VTKM_CONT_EXPORT
  Storage() : Container(NULL) {  }

  VTKM_CONT_EXPORT
  Storage(FooFieldsDeque *container) : Container(container) {  }

  VTKM_CONT_EXPORT
  PortalType GetPortal() { return PortalType(this->Container); }

  VTKM_CONT_EXPORT
  PortalConstType GetPortalConst() const {
    return PortalConstType(this->Container);
  }

  VTKM_CONT_EXPORT
  vtkm::Id GetNumberOfValues() const {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT_EXPORT
  void Allocate(vtkm::Id numberOfValues) {
    this->Container->resize(numberOfValues);
  }

  VTKM_CONT_EXPORT
  void Shrink(vtkm::Id numberOfValues) {
    this->Container->resize(numberOfValues);
  }

  VTKM_CONT_EXPORT
  void ReleaseResources() { this->Container->clear(); }

private:
  FooFieldsDeque *Container;
};

}
}
} // namespace vtkm::cont::internal
```

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 9.41: Array handle to adapt a third-party container to VTK-m.

```
class ArrayHandleFooPressure
    : public vtkm::cont::ArrayHandle<float, StorageTagFooPressure>
{
private:
  typedef vtkm::cont::internal::Storage<float, StorageTagFooPressure>
      StorageType;

public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
      ArrayHandleFooPressure,
      (vtkm::cont::ArrayHandle<float, StorageTagFooPressure>));

  VTKM_CONT_EXPORT
  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 typedefs `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 9.41 is not templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. (The other macro is described in Section 9.3.2 on page 66). 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. Note, however, that when writing to an array handle, it is necessary to call `GetPortalControl` or `GetPortalConstControl` to flush data from the execution environment to the control environment. [SHOULD PROBABLY MAKE THIS EASIER.]

Example 9.42: Using an `ArrayHandle` with custom container.

```
VTKM_CONT_EXPORT
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::worklet::DispatcherMapField<vtkm::worklet::PointElevation>
      dispatcher(elevation);
  dispatcher.Invoke(grid.GetCoordinateSystem().GetData(), pressureHandle);

  // Make sure the values are flushed back to the control environment.
  pressureHandle.GetPortalConstControl();

  // Now the pressure field is in the fields container.
}
```

Most of the code in VTK-m will create `ArrayHandle`s 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 9.43: 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 dynamic arrays (Chapter 10) or data sets (Chapter 11). When this happens, the array's type information, including the storage used, is lost. VTK-m will have to guess the storage, and if you do not tell VTK-m to try your custom storage, you will get a runtime error when the array is used. The most common ways of doing this are to change the default storage tag (described here), adding the storage tag to the default storage list (Section 10.3) or specifying the storage tag in the policy when executing filters ([Add reference when documented.]).

DRAFT

CHAPTER

TEN

DYNAMIC 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 DynamicArrayHandle class provides a mechanism to manage arrays of data with unspecified types.

vtkm::cont::DynamicArrayHandle holds a reference to an array. Unlike ArrayHandle, DynamicArrayHandle is *not* templated. Instead, it uses C++ run-type type information to store the array without type and cast it when appropriate.

A DynamicArrayHandle can be established by constructing it with or assigning it to an ArrayHandle. The following example demonstrates how a DynamicArrayHandle might be used to load an array whose type is not known until run-time.

Example 10.1: Creating a DynamicArrayHandle.

```
VTKM_CONT_EXPORT
vtkm::cont::DynamicArrayHandle
LoadDynamicArray(const void *buffer, vtkm::Id length, std::string type)
{
  vtkm::cont::DynamicArrayHandle 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;
}
```

10.1 Querying and Casting

Data pointed to by a DynamicArrayHandle 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 10.2: Non type-specific queries on DynamicArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle 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::Vec<vtkm::Float32,3> > vectorBuffer(20);
vtkm::cont::DynamicArrayHandle 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 DynamicArrayHandle. 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, DynamicArrayHandle has a method named NewInstance that creates a new empty array with the same underlying type as the original array.

Example 10.3: Using DynamicArrayHandle::NewInstance().

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle dynamicHandle(
      vtkm::cont::make_ArrayHandle(scalarBuffer));

// This creates a new empty array of type Float32.
vtkm::cont::DynamicArrayHandle newDynamicArray = dynamicHandle.NewInstance();
```

Before the data with a DynamicArrayHandle can be accessed, the type and storage of the array must be established. This is usually done internally within VTK-m when a worklet [OR FILTER?] is invoked. However, it is also possible to query the types and cast to a concrete ArrayHandle.

You can query the component type and storage type using the IsArrayHandleType, IsSameType, and IsTypeAndStorage methods. IsArrayHandleType takes an example array handle type and returns whether the underlying array matches the given static array type. IsSameType behaves the same as IsArrayHandleType but accepts an instances of an ArrayHandle object to automatically resolve the template parameters. IsTypeAndStorage takes an example component type and an example storage type as arguments and returns whether the underlying array matches both types.

Example 10.4: Querying the component and storage types of a DynamicArrayHandle.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::ArrayHandle<vtkm::Float32> concreteHandle =
    vtkm::cont::make_ArrayHandle(scalarBuffer);
vtkm::cont::DynamicArrayHandle dynamicHandle(concreteHandle);

// This returns true
bool isFloat32Array = dynamicHandle.IsSameType(concreteHandle);

// This returns false
bool isIdArray =
    dynamicHandle.IsArrayHandleType<vtkm::cont::ArrayHandle<vtkm::Id> >();

```

```
13  // This returns true
14  bool isFloat32 =
15      dynamicHandle.IsTypeAndStorage<vtkm::Float32,VTKM_DEFAULT_STORAGE_TAG>();
16
17  // This returns false
18  bool isId =
19      dynamicHandle.IsTypeAndStorage<vtkm::Id,VTKM_DEFAULT_STORAGE_TAG>();
20
21  // This returns false
22  bool isErrorStorage = dynamicHandle.IsTypeAndStorage<
23                            vtkm::Float32,
24                            vtkm::cont::internal::StorageTagError>();
```

Once the type of the DynamicArrayHandle is known, it can be cast to a concrete ArrayHandle, which has access to the data as described in Chapter 6. The easiest way to do this is to use the CopyTo method. This templated method takes a reference to an ArrayHandle as an argument and sets that array handle to point to the array in DynamicArrayHandle. If the given types are incorrect, then CopyTo throws a vtkm::cont::ErrorControlBadValue exception.

Example 10.5: Casting a DynamicArrayHandle to a concrete ArrayHandle.

```
1  dynamicHandle.CopyTo(concreteHandle);
```

Common Errors

Remember that ArrayHandle and DynamicArrayHandle 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.

10.2 Casting to Unknown Types

Using CopyTo is fine as long as the correct types are known, but often times they are not. For this use case DynamicArrayHandle 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 10.6: Operating on DynamicArrayHandle with CastAndCall.

```
1  struct PrintArrayContentsFunctor
2  {
3    template<typename T, typename Storage>
4    VTKM_CONT_EXPORT
5    void operator()(const vtkm::cont::ArrayHandle<T,Storage> &array) const
6    {
7      this->PrintArrayPortal(array.GetPortalConstControl());
8    }
9
10 private:
11   template<typename PortalType>
12   VTKM_CONT_EXPORT
13   void PrintArrayPortal(const PortalType &portal) const
14   {
15     for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
16     {
17       // All ArrayPortal objects have ValueType for the type of each value.
```


DRAFT

```
18       typedef typename PortalType::ValueType ValueType;
19
20       ValueType value = portal.Get(index);
21
22       vtkm::IdComponent numComponents =
23           vtkm::VecTraits<ValueType>::GetNumberOfComponents(value);
24       for (vtkm::IdComponent componentIndex = 0;
25            componentIndex < numComponents;
26            componentIndex++)
27       {
28         std::cout << " "
29                   << vtkm::VecTraits<ValueType>::GetComponent(value,
30                                                               componentIndex);
31       }
32       std::cout << std::endl;
33     }
34   }
35 };
36
37 template<typename DynamicArrayType>
38 void PrintArrayContents(const DynamicArrayType &array)
39 {
40   array.CastAndCall(PrintArrayContentsFunctor());
41 }
```

Common Errors

It is possible to store any form of ArrayHandle *in a* DynamicArrayHandle*, 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.*

10.3 Specifying Cast Lists

The CastAndCall method can only check a finite number of 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 5.7. There are separate lists for value types and for storage types.

Common type lists for value are defined in vtkm/TypeListTag.h and are documented in Section 5.7.2. This header also defines VTKM_DEFAULT_TYPE_LIST_TAG, which defines the default list of value types tried in CastAndCall.

Common storage lists are defined in vtkm/cont/StorageListTag.h. There is only one common storage distributed with VTK-m: StorageBasic. A list tag containing this type is defined as vtkm::cont::StorageListTagBasic.

As with other lists, it is possible to create new storage type lists using the existing type lists and the list bases from Section 5.7.1.

The vtkm/cont/StorageListTag.h header also defines a macro named VTKM_DEFAULT_STORAGE_LIST_TAG that defines a default list of types to use in classes like DynamicArrayHandle. This list can be overridden by defining the VTKM_DEFAULT_STORAGE_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.

There is a form of CastAndCall that accepts tags for the list of component types and storage types. This can

be used when the specific lists are known at the time of the call. However, when creating generic operations like the `PrintArrayContents` function in Example 10.6, passing these tags is inconvenient at best.

To address this use case, `DynamicArrayHandle` has a pair of methods named `ResetTypeList` and `ResetStorageList`. These methods return a new object with that behaves just like a `DynamicArrayHandle` with identical state except that the cast and call functionality uses the specified component type or storage type instead of the default. (Note that `PrintArrayContents` in Example 10.6 is templated on the type of `DynamicArrayHandle`. This is to accommodate using the objects from the `Reset*List` methods, 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 `ResetTypeList`.

Example 10.7: Trying all component types in a `DynamicArrayHandle`.

```
1 |   PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagAll()));
```

Likewise, if you happen to know a particular type of the dynamic array, that can be specified to reduce the amount of object code created by templates in the compiler.

Example 10.8: Specifying a single component type in a `DynamicArrayHandle`.

```
1 |   PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagId()));
```

Storage type lists can be changed similarly.

Example 10.9: Specifying different storage types in a `DynamicArrayHandle`.

```
 1 | struct MyIdStorageList :
 2 |     vtkm::ListTagBase<
 3 |         vtkm::cont::StorageTagBasic,
 4 |         vtkm::cont::ArrayHandleIndex::StorageTag>
 5 | { };
 6 |
 7 | void PrintIds(vtkm::cont::DynamicArrayHandle array)
 8 | {
 9 |   PrintArrayContents(array.ResetStorageList(MyIdStorageList()));
10 | }
```

Common Errors

The `ResetTypeList` *and* `ResetStorageList` *do not change the object they are called on. Rather, they return a new object with different type information. Calling these methods has no effect unless you do something with the returned value.*

Both the component type list and the storage type list can be modified by chaining these reset calls.

Example 10.10: Specifying both component and storage types in a `DynamicArrayHandle`.

```
1 |   PrintArrayContents(dynamicArray.
2 |                        ResetTypeList(vtkm::TypeListTagId()).
3 |                        ResetStorageList(MyIdStorageList()));
```

The `ResetTypeList` and `ResetStorageList` work by returning a `vtkm::cont::DynamicArrayHandleBase` object. `DynamicArrayHandleBase` specifies the value and storage tag lists as template arguments and otherwise behaves just like `DynamicArrayHandle`.

Did you know?

I lied earlier when I said at the beginning of this chapter that `DynamicArrayHandle` *is a class that is not templated. This symbol is really just a* `typedef` *of* `DynamicArrayHandleBase`*. Because the* `DynamicArrayHandle` *fully specifies the template arguments, it behaves like a class, but if you get a compiler error it will show up as* `DynamicArrayHandleBase`*.*

Most code does not need to worry about working directly with `DynamicArrayHandleBase`. However, it is sometimes useful to declare it in templated functions that accept dynamic array handles so that works with every type list. The function in Example 10.6 did this by making the dynamic 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 dynamic array handles will result in strange and hard to diagnose errors. Instead, we can define the same function using `DyamicArrayHandleBase` so that the template will only match dynamic array handle types.

Example 10.11: Using `DynamicArrayHandleBase` to accept generic dynamic array handles.

```
template<typename TypeList, typename StorageList>
void PrintArrayContents(
    const vtkm::cont::DynamicArrayHandleBase<TypeList,StorageList> &array)
{
  array.CastAndCall(PrintArrayContentsFunctor());
}
```

DRAFT

CHAPTER

ELEVEN

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 must have at least one cell set, but can have more than one cell set defined. This makes it possible to define groups of cells with different properties. For example, a simulation might model some subset of elements as boundary that contain properties the other elements do not. Another example is the representation of a molecule that requires atoms and bonds, each having very different properties associated with them.

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.

11.1 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 2.

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.

11.1.1 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 $101 \times 101 \times 26$ points.

Example 11.1: Creating a uniform grid.

```
1 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3 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 11.2: Creating a uniform grid with custom origin and spacing.

```
1 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3 vtkm::cont::DataSet dataSet =
4     dataSetBuilder.Create(
5       vtkm::Id3(101, 101, 26),
6       vtkm::Vec<vtkm::FloatDefault,3>(-50.0, -50.0, -50.0),
7       vtkm::Vec<vtkm::FloatDefault,3>(1.0, 1.0, 4.0));
```

11.1.2 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, in which case the data are shallow copied.

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

Example 11.3: Creating a rectilinear grid.

```
1  // Make x coordinates range from -4 to 4 with tighter spacing near 0.
2  std::vector<vtkm::Float32> xCoordinates;
3  for (vtkm::Float32 x = -2.0f; x <= 2.0f; x += 0.02f)
4  {
5    xCoordinates.push_back(vtkm::CopySign(x*x, x));
6  }
7
8  // Make y coordinates range from 0 to 2 with tighter spacing near 2.
9  std::vector<vtkm::Float32> yCoordinates;
10 for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
11 {
```

DRAFT

```
12      yCoordinates.push_back(vtkm::Sqrt(y));
13    }
14
15    // Make z coordinates rangefrom -1 to 1 with even spacing.
16    std::vector<vtkm::Float32> zCoordinates;
17    for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
18    {
19      zCoordinates.push_back(z);
20    }
21
22    vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;
23
24    vtkm::cont::DataSet dataSet = dataSetBuilder.Create(xCoordinates,
25                                                        yCoordinates,
26                                                        zCoordinates);
```

11.1.3 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.

The cells of an explicit mesh are defined by providing the shape, number of indices, and the points that comprise it for each cell. These three things are stored in separate arrays. Figure 11.1 shows an example of an explicit mesh and the arrays that can be used to define it.

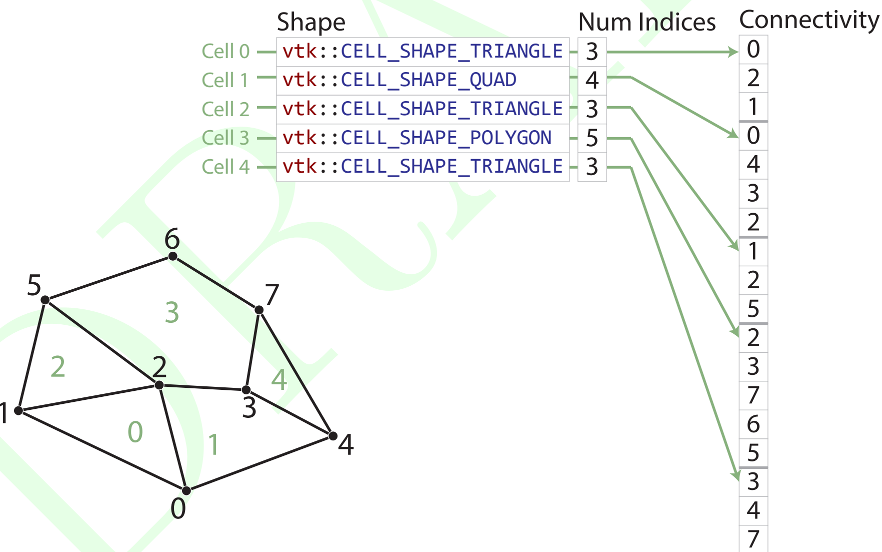


Figure 11.1: An example explicit mesh.

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 11.1.

Example 11.4: Creating an explicit mesh with `DataSetBuilderExplicit`.

```
1    // Array of point coordinates.
```

```
2   std::vector<vtkm::Vec<vtkm::Float32,3> > pointCoordinates;
3   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.1f, 0.0f, 0.0f));
4   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.2f, 0.4f, 0.0f));
5   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.9f, 0.6f, 0.0f));
6   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.4f, 0.5f, 0.0f));
7   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.8f, 0.3f, 0.0f));
8   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(0.4f, 1.0f, 0.0f));
9   pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.0f, 1.2f, 0.0f));
10  pointCoordinates.push_back(vtkm::Vec<vtkm::Float32,3>(1.5f, 0.9f, 0.0f));
11
12  // Array of shapes.
13  std::vector<vtkm::UInt8> shapes;
14  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
15  shapes.push_back(vtkm::CELL_SHAPE_QUAD);
16  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
17  shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
18  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
19
20  // Array of number of indices per cell.
21  std::vector<vtkm::IdComponent> numIndices;
22  numIndices.push_back(3);
23  numIndices.push_back(4);
24  numIndices.push_back(3);
25  numIndices.push_back(5);
26  numIndices.push_back(3);
27
28  // Connectivity array.
29  std::vector<vtkm::Id> connectivity;
30  connectivity.push_back(0); // Cell 0
31  connectivity.push_back(2);
32  connectivity.push_back(1);
33  connectivity.push_back(0); // Cell 1
34  connectivity.push_back(4);
35  connectivity.push_back(3);
36  connectivity.push_back(2);
37  connectivity.push_back(1); // Cell 2
38  connectivity.push_back(2);
39  connectivity.push_back(5);
40  connectivity.push_back(2); // Cell 3
41  connectivity.push_back(3);
42  connectivity.push_back(7);
43  connectivity.push_back(6);
44  connectivity.push_back(5);
45  connectivity.push_back(3); // Cell 4
46  connectivity.push_back(4);
47  connectivity.push_back(7);
48
49  // Copy these arrays into a DataSet.
50  vtkm::cont::DataSetBuilderExplicit dataSetBuilder;
51
52  vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointCoordinates,
53                                                      shapes,
54                                                      numIndices,
55                                                      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 11.1 except this time using DataSetBuilderExplicitIterative.

Example 11.5: Creating an explicit mesh with DataSetBuilderExplicitIterative.

```
1   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();
```

11.1.4 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`, 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 11.6: 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);
```

11.2 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.) 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 17.1 starting on page 139.

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.

11.2.1 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 11.2 demonstrates this arrangement.

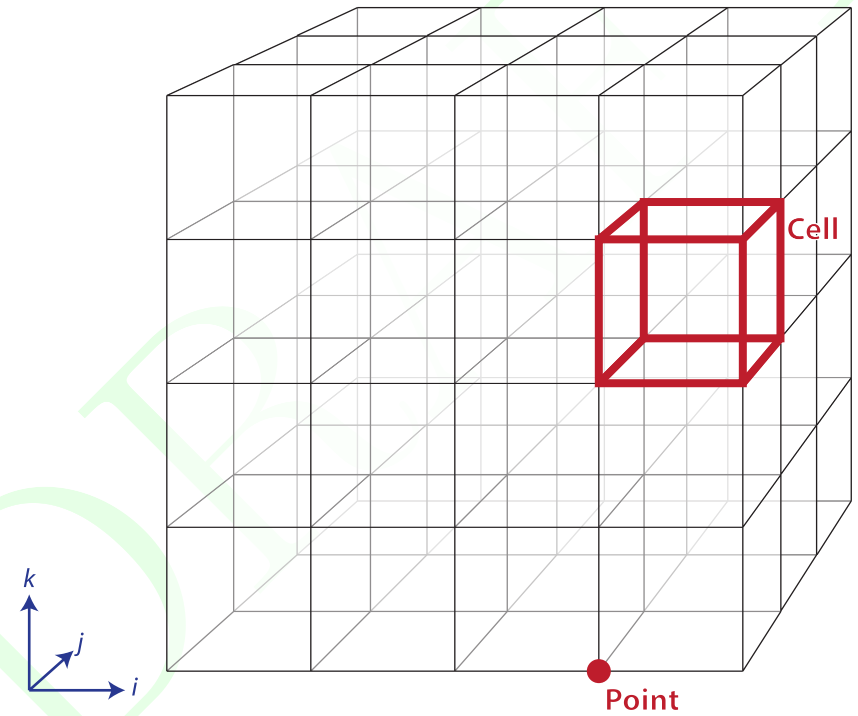


Figure 11.2: The arrangement of points and cells in a 3D structured grid.

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

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 11.4 starting on page 94.

11.2.2 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. 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. (Cell shapes are discussed in detail in Section 17.1 starting on page 139.) The second array identifies how many points are in each cell. The third array has a sequence of point indices that make up each cell. Figure 11.3 shows a simple example of an explicit cell set.

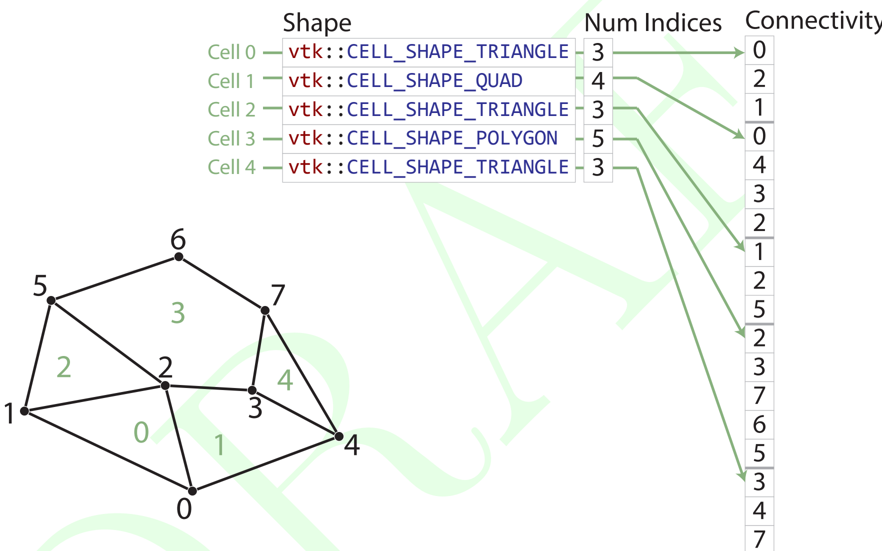


Figure 11.3: Example of cells in a `CellSetExplict` and the arrays that define them.

An explicit cell set may also have other topological arrays such as an array of offsets of each cell into the connectivity array or an array of cells incident on each point. Although these arrays can be provided, they are optional and can be internally derived from the shape, num indices, and connectivity arrays.

`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.

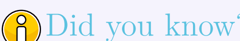
An important specialization of an explicit cell set is `vtkm::cont::CellSetSingleType`. `CellSetSingleType` is an explicit cell set constrained to contain cells that all have the same shape and all have the same number of points. So for example if you are creating a surface that you know will contain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

11.2.3 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.

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 11.7: Subsampling a data set with CellSetPermutation.

```
1  // Create a simple data set.
2  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
3  vtkm::cont::DataSet originalDataSet =
4      dataSetBuilder.Create(vtkm::Id3(33,33,26));
5  vtkm::cont::CellSetStructured<3> originalCellSet;
6  originalDataSet.GetCellSet().CopyTo(originalCellSet);
7
8  // Create a permutation array for the cells. Each value in the array refers
9  // to a cell in the original cell set. This particular array selects every
10 // 10th cell.
11 vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);
12
13 // Create a permutation of that cell set containing only every 10th cell.
14 vtkm::cont::CellSetPermutation<
15       vtkm::cont::CellSetStructured<3>,
16       vtkm::cont::ArrayHandleCounting<vtkm::Id> >
17     permutedCellSet(permutationArray, originalCellSet);
```

11.2.4 Dynamic Cell Sets

vtkm::cont::DataSet must hold an arbitrary collection of vtkm::cont::CellSet objects, which it cannot do while knowing their types at compile time. To manage storing CellSets without knowing their types, DataSet actually holds references using vtkm::cont::DynamicCellSet.

DynamicCellSet is similar in nature to DynamicArrayHandle except that it, of course, holds CellSets instead of ArrayHandles. The interface for the two classes is similar, and you should review the documentation for DynamicArrayHandle (in Chapter 10 starting on page 79) to understand DynamicCellSet.

vtkm::cont::DynamicCellSet has a method named GetCellSet that returns a const reference to the held cell set as the abstract CellSet class. This can be used to easily access the virtual methods in the CellSet interface. You can also create a new instance of a cell set with the same type using the NewInstance method.

The DynamicCellSet::IsType() method can be used to determine whether the cell set held in the dynamic cell set is of a given type. If the cell set type is known, DynamicCellSet::CastTo() can be used to safely downcast the cell set object.

DRAFT

When a typed version of the cell set stored in the DynamicCellSet is needed but the type is not known, which happens regularly in the internal workings of VTK-m, the CastAndCall method can be used to make this transition. CastAndCall works by taking a functor and calls it with the appropriately cast cell set object.

The CastAndCall method works by attempting to cast to a known set of types. This set of types used is defined by the macro VTKM_DEFAULT_CELL_SET_LIST_TAG, which is declared in vtkm/cont/CellSetListTag.h. This list can be overridden globally by defining the VTKM_DEFAULT_CELL_SET_LIST_TAG macro *before* any VTK-m headers are included.

The set of types used in a CastAndCall can also be changed only for a particular instance of a dynamic cell set by calling its ResetCellSetList. This method takes a list of cell types and returns a new dynamic array handle of a slightly different type that will use this new list of cells for dynamic casting.

11.2.5 Blocks and Assemblies

Rather than just one cell set, a vtkm::cont::DataSet can hold multiple cell sets. This can be used to construct multiblock data structures or assemblies of parts. Multiple cell sets can also be used to represent subsets of the data with particular properties such as all cells filled with a material of a certain type. Or these multiple cells might represent particular features in the data, such as the set of faces representing a boundary in the simulation.

11.2.6 Zero Cell Sets

It is also possible to construct a vtkm::cont::DataSet that contains no cell set objects whatsoever. This can be used to manage data that does not contain any topological structure. For example, a collection of series that come from columns in a table could be stored as multiple fields in a data set with no cell set.

11.3 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. Field holds its data with a DynamicArrayHandle, which itself is a container for an ArrayHandle. Field also maintains the association and, optionally, the name of a cell set for which the field is valid.

The data array can be retrieved as a DynamicArrayHandle using the GetData method of Field. Field also has a convenience method named GetBounds that finds the range of values stored in the field array.

11.4 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.

DRAFT

`CoordinateSystem` constrains the field to be associated with points and typically has 3D floating point vectors for values.

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.

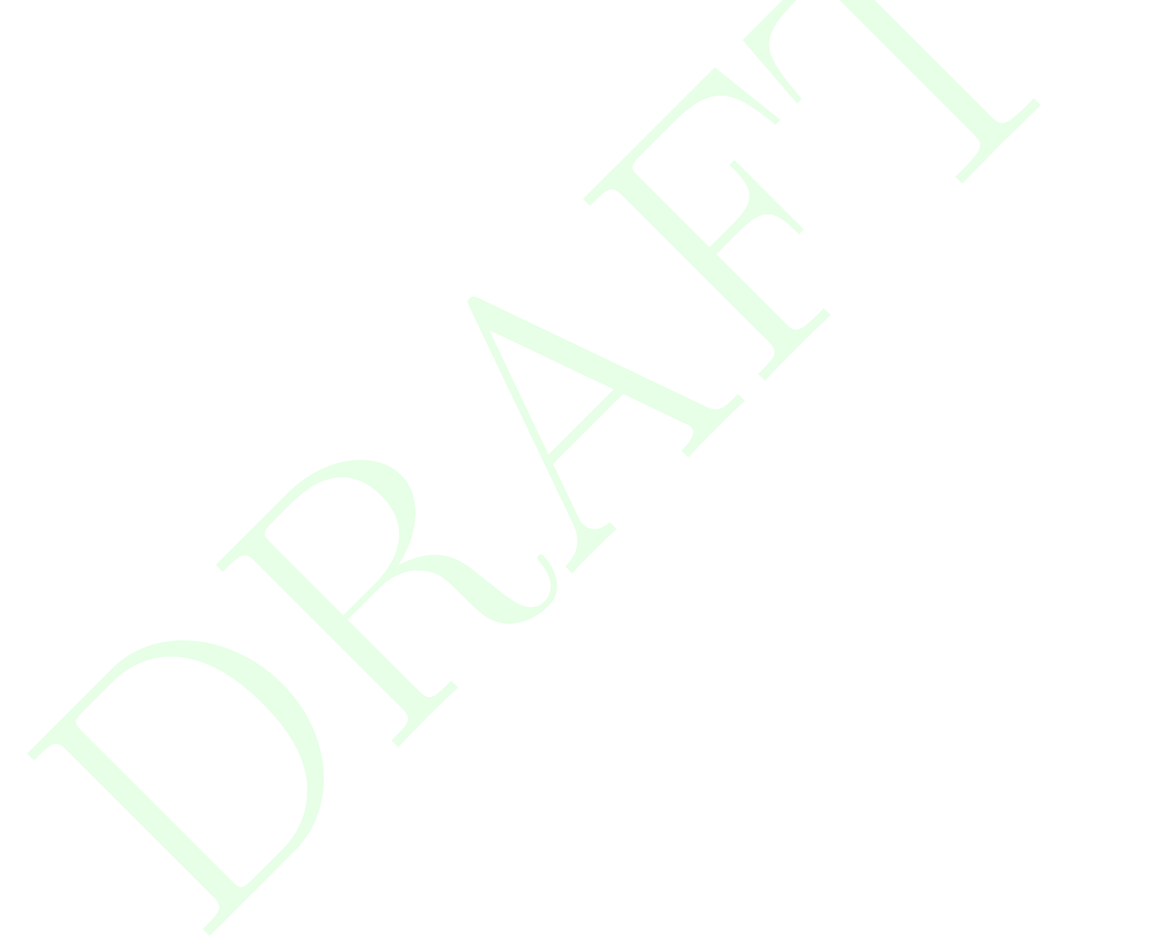
It is also valid to have a `DataSet` with no coordinate system. This is useful when the structure is not rooted in physical space. For example, if the cell set is representing a graph structure, there might not be any physical space that has meaning for the graph.

DRAFT

CHAPTER

TWELVE

FILTER POLICIES


DRAFT

DRAFT

CHAPTER

THIRTEEN

OPENGL INTEROPERABILITY

Part III

Developing with VTK-m

DRAFT

CHAPTER

FOURTEEN

WORKLETS

DRAFT

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. This chapter explains the basic mechanics of defining and using worklets.

14.1 Worklet Types

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. Details on how to create worklets of each type are given in Section 14.5. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 18.

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 sibling 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::WorkletMapPointToCell` 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.

14.2 Dispatchers

Worklets, both those provided by VTK-m as listed in Section 14.3 and ones created by a user as described in Section 14.4, 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 invocation is done through a set of *dispatcher* objects. 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 two template parameters: the worklet class being invoked, which is always the first argument, and the device adapter tag, which is always the last argument and will be set to the default device adapter if not specified.

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 expected arguments for worklets provided by VTK-m are documented in Section 14.3. Also, for any worklet, the `Invoke` arguments can be gleaned from the control signature, which is described in Section 14.4.1.

The following is a list of the dispatchers defined in VTK-m. The dispatcher classes correspond to the list of worklet types specified in Section 14.1. Many examples of using these dispatchers are provided in Section 14.3.

`vtkm::worklet::DispatcherMapField` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapField`. The dispatcher class has two template arguments: the worklet type and the device adapter (optional).

`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::WorkletMapPointToCell`). The dispatcher class has two template arguments: the worklet type and the device adapter (optional).

14.3 Provided Worklets

[Write this once some worklets are provided.]

14.4 Creating Worklets

A worklet is created by implementing a `class` or `struct` with the following features.

1. The class must contain a `ControlSignature` `typedef`, which specifies what arguments are expected when invoking the class with a dispatcher in the control environment.
2. The class must contain an `ExecutionSignature` `typedef`, which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.
3. The class must contain an `InputDomain` `typedef`, which identifies which input parameter defines the input domain of the data.
4. The class may define a scatter operation to override a 1:1 mapping from input to output.

DRAFT

5. The class must contain an overload of the parenthesis operator, which is the method that is executed in the execution environment.

6. The class must publicly inherit from a base worklet class that specifies the type of operation being performed.

Figure 14.1 demonstrates all of the required components of a worklet.

Defines dispatching method

```
class TriangulateCell : public vtkm::worklet::WorkletMapPointToCell
{
public:
  typedef void ControlSignature(TopologyIn topology,
                                ExecObject tables,
                                FieldOutCell<> connectivityOut);
  typedef void ExecutionSignature(CellShape, PointIndices, _2, _3, VisitIndex);
  typedef _1 InputDomain;

  typedef vtkm::worklet::ScatterCounting ScatterType;
  VTKM_CONT_EXPORT
  ScatterType GetScatter() const
  {
    return this->Scatter;
  }

  template<typename CellShapeTag,
           typename ConnectivityInVec,
           typename ConnectivityOutVec>
  VTKM_EXEC_EXPORT
  void operator()(
      CellShapeTag shape,
      const ConnectivityInVec &connectivityIn,
      const internal::TriangulateTablesExecutionObject<DeviceAdapter> &tables,
      ConnectivityOutVec &connectivityOut,
      vtkm::IdComponent visitIndex) const
  {
```

Defines how input arrays and structures are interpreted

Specifies domain argument (optional)

Defines how data are assigned to threads

Defines mapping from input domain to output domain (optional)

Algorithms are just functions that run on a single instance of the input

Figure 14.1: Annotated example of a worklet declaration.

14.4.1 Control Signature

The control signature of a worklet is the `typedef` of a function prototype named `ControlSignature`. The function prototype matches the calling specification used with the dispatcher `Invoke` function.

Example 14.1: A `ControlSignature`.

```
1  typedef void ControlSignature(FieldIn<VecAll> inputVectors,
2                                FieldOut<Scalar> outputMagnitudes);
```

The return type of the function prototype is always `void` because the dispatcher `Invoke` functions do not return values. 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 Section 14.5.

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

DRAFT

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.

Type List Tags

Some tags are templated to have modifiers. For example, `Field` tags have a template argument that is set to a type list tag defining what types of field data are supported. (See Section 5.7.2 for a description of type lists.) In fact, this type list modifier is so common that the following convenience subtags used with `Field` tags are defined for all worklet types.

> **Did you know?**
> *Any type list will work as modifiers for* `ControlSignature` *tags. However, these common type lists are provided for convenience and to make the* `ControlSignature` *shorter and more readable.*

`AllTypes` All possible types.

`CommonTypes` 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. The same as `vtkm::TypeListTagCommon`.

`IdType` Contains the single item `vtkm::Id`. The same as `vtkm::TypeListTagId`.

`Id2Type` Contains the single item `vtkm::Id2`. The same as `vtkm::TypeListTagId2`.

`Id3Type` Contains the single item `vtkm::Id3`. The same as `vtkm::TypeListTagId3`.

`Index` All types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`. The same as `vtkm::TypeListTagIndex`.

`FieldCommon` A list containing all the types generally used for fields. It is the combination of `Scalar`, `Vec2`, `Vec3`, and `Vec4`. The same as `vtkm::TypeListTagField`.

`Scalar` Types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`). The same as `vtkm::TypeListTagFieldScalar`.

`ScalarAll` 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. The same as `vtkm::TypeListTagScalarAll`.

`Vec2` Types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec2`.

`Vec3` Types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec3`.

`Vec4` Types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec4`.

`VecAll` All `vtkm::Vec` classes with standard integers or floating points as components and lengths between 2 and 4. The same as `vtkm::TypeListTagVecAll`.

`VecCommon` 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. The same as `vtkm::TypeListTagVecCommon`.

DRAFT

14.4.2 Execution Signature

Like the control signature, the execution signature of a worklet is the `typedef` of a function prototype named `ExecutionSignature`. The function prototype must match the parenthesis operator (described in Section 14.4.4) in terms of arity and argument semantics.

Example 14.2: An `ExecutionSignature`.

```
1  typedef _2 ExecutionSignature(_1);
```

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.

14.4.3 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. One of the arguments given to the dispatcher's `Invoke` in the control environment must specify the domain.

A worklet identifies the argument specifying the domain with a `typedef` named `InputDomain`. The `InputDomain` must be `typedef`ed 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 14.3: An `InputDomain` declaration.

```
1  typedef _1 InputDomain;
```

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 `TopologyIn` 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.

14.4.4 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 14.4: An overloaded parenthesis operator of a worklet.

```
1  template<typename T, vtkm::IdComponent Size>
```

DRAFT

```
VTKM_EXEC_EXPORT
T operator()(const vtkm::Vec<T,Size> &inVector) 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_EXPORT (or VTKM_EXEC_CONT_EXPORT) modifier. Third, the method must be declared `const` to help preserve thread safety.

14.5 Worklet Type Reference

There are multiple worklet types provided by VTK-m, each designed to support a particular type of operation. Section 14.1 gave a brief overview of each type of worklet. This section gives a much more detailed reference for each of the worklet types including identifying the generic superclass that a worklet instance should derive, listing the signature tags and their meanings, and giving an example of the worklet in use.

14.5.1 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 `WorkletMapField` subclass is invoked with a `vtkm::worklet::DispatcherMapField`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

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 `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array.

`FieldIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

`FieldInOut` This tag represents field that is both an input and an output. A `FieldInOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

DRAFT

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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, and this same object is given to the worklet. Execution objects are discussed in detail in Section 14.7 starting on page 123.

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 14.8).

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 14.5: Implementation and use of a field map worklet.

```
1  #include <vtkm/worklet/DispatcherMapField.h>
2  #include <vtkm/worklet/WorkletMapField.h>
3
4  #include <vtkm/cont/ArrayHandle.h>
5  #include <vtkm/cont/DynamicArrayHandle.h>
6
7  #include <vtkm/VectorAnalysis.h>
8
9
10 class Magnitude : public vtkm::worklet::WorkletMapField
```

```
11 {
12 public:
13   typedef void ControlSignature(FieldIn<VecAll> inputVectors,
14                                 FieldOut<Scalar> outputMagnitudes);
15   typedef _2 ExecutionSignature(_1);
16
17   typedef _1 InputDomain;
18
19   template<typename T, vtkm::IdComponent Size>
20   VTKM_EXEC_EXPORT
21   T operator()(const vtkm::Vec<T,Size> &inVector) const
22   {
23     return vtkm::Magnitude(inVector);
24   }
25 };
26
27 VTKM_CONT_EXPORT
28 vtkm::cont::DynamicArrayHandle
29 InvokeMagnitude(vtkm::cont::DynamicArrayHandle input)
30 {
31   vtkm::cont::ArrayHandle<vtkm::Float64> output;
32
33   vtkm::worklet::DispatcherMapField<Magnitude> dispatcher;
34   dispatcher.Invoke(input, output);
35
36   return vtkm::cont::DynamicArrayHandle(output);
37 }
```

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 Sections 14.6 and 14.7, 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 14.6: Leveraging field maps and field maps for general processing.

```
1 struct ReverseArrayCopy : vtkm::worklet::WorkletMapField
2 {
3   typedef void ControlSignature(FieldIn<> inputArray,
4                                 WholeArrayOut<> outputArray);
5   typedef void ExecutionSignature(_1, _2, WorkIndex);
6   typedef _1 InputDomain;
7
8   template<typename InputType, typename OutputArrayPortalType>
9   VTKM_EXEC_EXPORT
10   void operator()(const InputType &inputValue,
11                   const OutputArrayPortalType &outputArrayPortal,
12                   vtkm::Id workIndex) const
13   {
14     vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
15     if (outIndex >= 0)
16     {
17       outputArrayPortal.Set(outIndex, inputValue);
18     }
19     else
20     {
21       this->RaiseError("Output array not sized correctly.");
22     }
23   }
24 };
25
26 template<typename T, typename Storage>
```

```
27 VTKM_CONT_EXPORT
28 vtkm::cont::ArrayHandle<T>
29 InvokeReverseArrayCopy(const vtkm::cont::ArrayHandle<T,Storage> &inArray)
30 {
31   vtkm::cont::ArrayHandle<T> outArray;
32   outArray.Allocate(inArray.GetNumberOfValues());
33
34   vtkm::worklet::DispatcherMapField<ReverseArrayCopy> dispatcher;
35   dispatcher.Invoke(inArray, outArray);
36
37   return outArray;
38 }
```

14.5.2 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. Regardless of their names, they are all defined in vtkm/worklet/WorkletMapTopology.h and are all invoked with `vtkm::worklet::DispatcherMapTopology`.

Point to Cell Map

A worklet deriving `vtkm::worklet::WorkletMapPointToCell` 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 `WorkletMapPointToCell` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A point to cell map worklet supports the following tags in the parameters of its `ControlSignature`.

`TopologyIn` This tag represents the cell set that defines the collection of cells the map will operate on. A `TopologyIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Section ??.)

There must be exactly one `TopologyIn` 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 `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

FieldInCell This tag represents an input field that is associated with the cells. A FieldInCell argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. 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.

FieldInCell has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

FieldOutCell This tag represents an output field, which is necessarily associated with cells. A FieldOutCell argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOutCell has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOutCell has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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, and this same object is given to the worklet. Execution objects are discussed in detail in Section 14.7 starting on page 123.

DRAFT

A field 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 cell. (Cell shapes and the operations you can do with cells are discussed in Section ??.) This is the same value that gets provided if you reference the `TopologyIn` 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 14.8).

Point to cell field maps are a powerful construct that allow you to interpolate point fields throughout the space of the data set. 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 14.7: Implementation and use of a map point to cell worklet.

```
 1 #include <vtkm/worklet/DispatcherMapTopology.h>
 2 #include <vtkm/worklet/WorkletMapTopology.h>
 3
 4 #include <vtkm/cont/DataSet.h>
 5 #include <vtkm/cont/DataSetFieldAdd.h>
 6
 7 #include <vtkm/exec/CellInterpolate.h>
 8 #include <vtkm/exec/ParametricCoordinates.h>
 9
10
11 class CellCenter : public vtkm::worklet::WorkletMapPointToCell
12 {
13 public:
14   typedef void ControlSignature(TopologyIn cellSet,
15                                 FieldInPoint<> inputPointField,
16                                 FieldOut<> outputCellField);
17   typedef _3 ExecutionSignature(_1, PointCount, _2);
18
19   typedef _1 InputDomain;
20
21   template<typename CellShape,
22            typename InputPointFieldType>
23   VTKM_EXEC_EXPORT
24   typename InputPointFieldType::ComponentType
25   operator()(CellShape shape,
26              vtkm::IdComponent numPoints,
27              const InputPointFieldType &inputPointField) const
28   {
29     vtkm::Vec<vtkm::FloatDefault,3> parametricCenter =
30         vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, *this);
31     return vtkm::exec::CellInterpolate(inputPointField,
32                                        parametricCenter,
33                                        shape,
```

```
34                                        *this);
35   }
36 };
37
38 VTKM_CONT_EXPORT
39 void FindCellCenters(vtkm::cont::DataSet &dataSet)
40 {
41   vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault,3> > cellCentersArray;
42
43   vtkm::worklet::DispatcherMapTopology<CellCenter> dispatcher;
44   dispatcher.Invoke(dataSet.GetCellSet(),
45                     dataSet.GetCoordinateSystem().GetData(),
46                     cellCentersArray);
47
48   vtkm::cont::DataSetFieldAdd dataSetFieldAdd;
49   dataSetFieldAdd.AddCellField(dataSet, "cell_center", cellCentersArray);
50 }
```

Cell To Point Map

A worklet deriving `vtkm::worklet::WorkletMapCellToPoint` 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 `WorkletMapCellToPoint` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A cell to point map worklet supports the following tags in the parameters of its `ControlSignature`.

`TopologyIn` This tag represents the cell set that defines the collection of points the map will operate on. A `TopologyIn` argument expects a `CellSet` subclass or a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`.

There must be exactly one `TopologyIn` 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 `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

DRAFT

FieldOutPoint This tag represents an output field, which is necessarily associated with points. A FieldOutPoint argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOutPoint has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOutPoint has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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, and this same object is given to the worklet. Execution objects are discussed in detail in Section 14.7 starting on page 123.

A field map 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.

DRAFT

`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 14.8).

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 14.8: Implementation and use of a map cell to point worklet.

```
1  #include <vtkm/worklet/DispatcherMapTopology.h>
2  #include <vtkm/worklet/WorkletMapTopology.h>
3
4  #include <vtkm/cont/DataSet.h>
5  #include <vtkm/cont/DataSetFieldAdd.h>
6  #include <vtkm/cont/DynamicArrayHandle.h>
7  #include <vtkm/cont/DynamicCellSet.h>
8  #include <vtkm/cont/Field.h>
9
10
11 class AverageCellField : public vtkm::worklet::WorkletMapCellToPoint
12 {
13 public:
14   typedef void ControlSignature(TopologyIn cellSet,
15                                 FieldInCell<> inputCellField,
16                                 FieldOut<> outputPointField);
17   typedef void ExecutionSignature(CellCount, _2, _3);
18
19   typedef _1 InputDomain;
20
21   template<typename InputCellFieldType, typename OutputFieldType>
22   VTKM_EXEC_EXPORT
23   void
24   operator()(vtkm::IdComponent numCells,
25              const InputCellFieldType &inputCellField,
26              OutputFieldType &fieldAverage) const
27   {
28     // TODO: This trickery with calling DoAverage with an extra fabricated type
29     // is to handle when the dynamic type resolution provides combinations that
30     // are incompatible. On the todo list for VTK-m is to allow you to express
31     // types that are the same for different parameters of the control
32     // signature. When that happens, we can get rid of this hack.
33     typedef typename InputCellFieldType::ComponentType InputComponentType;
34     this->DoAverage(numCells,
35                     inputCellField,
36                     fieldAverage,
37                     vtkm::ListTagBase<InputComponentType,OutputFieldType>());
38   }
39
40 private:
41   template<typename InputCellFieldType, typename OutputFieldType>
42   VTKM_EXEC_EXPORT
43   void DoAverage(vtkm::IdComponent numCells,
44                  const InputCellFieldType &inputCellField,
45                  OutputFieldType &fieldAverage,
46                  vtkm::ListTagBase<OutputFieldType,OutputFieldType>) const
```

```
47    {
48      fieldAverage = OutputFieldType(0);
49
50      for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
51      {
52        fieldAverage = fieldAverage + inputCellField[cellIndex];
53      }
54
55      fieldAverage = fieldAverage / OutputFieldType(numCells);
56    }
57
58    template<typename T1, typename T2, typename T3>
59    VTKM_EXEC_EXPORT
60    void DoAverage(vtkm::IdComponent, T1, T2, T3) const
61    {
62      this->RaiseError("Incompatible types for input and output.");
63    }
64  };
65
66  VTKM_CONT_EXPORT
67  vtkm::cont::DataSet
68  ConvertCellFieldsToPointFields(const vtkm::cont::DataSet &inData)
69  {
70    vtkm::cont::DataSet outData;
71
72    // Copy parts of structure that should be passed through.
73    for (vtkm::Id cellSetIndex = 0;
74         cellSetIndex < inData.GetNumberOfCellSets();
75         cellSetIndex++)
76    {
77      outData.AddCellSet(inData.GetCellSet(cellSetIndex));
78    }
79    for (vtkm::Id coordSysIndex = 0;
80         coordSysIndex < inData.GetNumberOfCoordinateSystems();
81         coordSysIndex++)
82    {
83      outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSysIndex));
84    }
85
86    // Copy all fields, converting cell fields to point fields.
87    for (vtkm::Id fieldIndex = 0;
88         fieldIndex < inData.GetNumberOfFields();
89         fieldIndex++)
90    {
91      vtkm::cont::Field inField = inData.GetField(fieldIndex);
92      if (inField.GetAssociation() == vtkm::cont::Field::ASSOC_CELL_SET)
93      {
94        vtkm::cont::DynamicArrayHandle inFieldData = inField.GetData();
95        vtkm::cont::DynamicCellSet inCellSet =
96            inData.GetCellSet(inField.GetAssocCellSet());
97
98        vtkm::cont::DynamicArrayHandle outFieldData = inFieldData.NewInstance();
99        vtkm::worklet::DispatcherMapTopology<AverageCellField> dispatcher;
100       dispatcher.Invoke(inCellSet, inFieldData, outFieldData);
101
102       vtkm::cont::DataSetFieldAdd::AddCellField(outData,
103                                                 inField.GetName(),
104                                                 outFieldData,
105                                                 inField.GetAssocCellSet());
106     }
107     else
108     {
109       outData.AddField(inField);
110     }
```

```
111   }
112
113   return outData;
114 }
```

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 "from" topology element, and the second template parameter specifies the "to" topology element. The worklet is scheduled such that each instance is associated with a particular "to" topology element and has access to incident "from" topology elements.

These from and to 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::TopologyElementTagPoint, vtkm::TopologyElementTagCell>
```

is equivalent to the vtkm::worklet::WorkletMapPointToCell class except the signature tags have different names. The names used in the specific topology map superclasses (such as WorkletMapPointToCell) 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.

TopologyIn This tag represents the cell set that defines the collection of elements the map will operate on. A TopologyIn argument expects a CellSet subclass or a DynamicCellSet in the associated parameter of the dispatcher's Invoke. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Section ??.)

There must be exactly one TopologyIn argument, and the worklet's InputDomain must be set to this argument.

FieldInFrom This tag represents an input field that is associated with the "from" elements. A FieldInFrom argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. The size of the array must be exactly the number of "from" elements.

Each invocation of the worklet gets a Vec-like object containing the field values for all the "from" elements incident with the "to" element being visited. If the field is a vector field, then the provided object is a Vec of Vecs.

FieldInFrom has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

FieldInTo This tag represents an input field that is associated with the "to" element. A FieldInTo argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. 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.

FieldInTo has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

FieldOut This tag represents an output field, which is necessarily associated with "to" elements. A FieldOut argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

FieldInOut This tag represents field that is both an input and an output, which is necessarily associated with "to" elements. A FieldInOut argument expects an ArrayHandle or a DynamicArrayHandle in the associated parameter of the dispatcher's Invoke. 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 has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayIn has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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 dispatcher's Invoke. 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 14.6 starting on page 120.

WholeArrayInOut has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 14.4.1 starting on page 106.

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, and this same object is given to the worklet. Execution objects are discussed in detail in Section 14.7 starting on page 123.

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 "to" element. (Cell shapes and the operations you can do with cells are discussed in Section ??.) This is the same value that gets provided if you reference the TopologyIn parameter.

If the "to" 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.

FromCount This tag produces a vtkm::IdComponent equal to the number of "from" elements incident on the "to" element being visited. The Vecs provided from a FieldInFrom parameter will be the same size as FromCount.

FromIndices This tag produces a Vec-like object of vtkm::Ids giving the indices for all incident "from" elements. The order of the entries is consistent with the values of all other FieldInFrom 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 14.8).

14.6 Whole Arrays

As documented in Section 14.5, each worklet type has a set of parameter types that can be used to pass data to and from the worklet invocation. But what happens if you want to pass data that cannot be expressed in these predefined mechanisms. Chapter 18 describes how to create completely new worklet types and parameter tags. However, designing such a system for a one-time use is overkill.

Instead, all VTK-m worklets provide a couple of mechanisms that allow you to pass arbitrary data to a worklet. In this section, we will explore a *whole array* argument that provides random access to an entire array. In the following section we describe an even more general mechanism to describe any execution object.

We have already seen a demonstration of using a whole array in Example 14.6 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 14.9: Using WholeArrayIn to access a lookup table in a worklet.

```
1 #include <vtkm/cont/ArrayHandle.h>
2 #include <vtkm/cont/DataSet.h>
```

```
3
4  #include <vtkm/worklet/DispatcherMapTopology.h>
5  #include <vtkm/worklet/WorkletMapTopology.h>
6
7  #include <vtkm/CellShape.h>
8  #include <vtkm/Math.h>
9  #include <vtkm/VectorAnalysis.h>
10
11 static const vtkm::Id TRIANGLE_QUALITY_TABLE_DIMENSION = 8;
12 static const vtkm::Id TRIANGLE_QUALITY_TABLE_SIZE =
13     TRIANGLE_QUALITY_TABLE_DIMENSION*TRIANGLE_QUALITY_TABLE_DIMENSION;
14
15 VTKM_CONT_EXPORT
16 vtkm::cont::ArrayHandle<vtkm::Float32> GetTriangleQualityTable()
17 {
18   // Use these precomputed values for the array. A real application would
19   // probably use a larger array, but we are keeping it small for demonstration
20   // purposes.
21   static vtkm::Float32 triangleQualityBuffer[TRIANGLE_QUALITY_TABLE_SIZE] = {
22     0, 0,        0,        0,        0,        0,        0,        0,
23     0, 0,        0,        0,        0,        0,        0,        0.244311,
24     0, 0,        0,        0,        0,        0,        0.432985, 0.470588,
25     0, 0,        0,        0,        0,        0.542168, 0.659231, 0.664078,
26     0, 0,        0,        0,        0.579721, 0.754247, 0.821543, 0.815365,
27     0, 0,        0,        0.542168, 0.754247, 0.874598, 0.925667, 0.920712,
28     0, 0,        0.432985, 0.659231, 0.821543, 0.925667, 0.976641, 0.980996,
29     0, 0.244311, 0.470588, 0.664078, 0.815365, 0.920712, 0.980996, 1
30   };
31
32   return vtkm::cont::make_ArrayHandle(triangleQualityBuffer,
33                                       TRIANGLE_QUALITY_TABLE_SIZE);
34 }
35
36 template<typename T>
37 VTKM_EXEC_CONT_EXPORT
38 vtkm::Vec<T,3> TriangleEdgeLengths(const vtkm::Vec<T,3> &point1,
39                                    const vtkm::Vec<T,3> &point2,
40                                    const vtkm::Vec<T,3> &point3)
41 {
42   return vtkm::make_Vec(vtkm::Magnitude(point1-point2),
43                         vtkm::Magnitude(point2-point3),
44                         vtkm::Magnitude(point3-point1));
45 }
46
47 VTKM_SUPPRESS_EXEC_WARNINGS
48 template<typename PortalType, typename T>
49 VTKM_EXEC_CONT_EXPORT
50 vtkm::Float32 LookupTriangleQuality(const PortalType &triangleQualityPortal,
51                                     const vtkm::Vec<T,3> &point1,
52                                     const vtkm::Vec<T,3> &point2,
53                                     const vtkm::Vec<T,3> &point3)
54 {
55   vtkm::Vec<T,3> edgeLengths = TriangleEdgeLengths(point1, point2, point3);
56
57   // To reduce the size of the table, we just store the quality of triangles
58   // with the longest edge of size 1. The table is 2D indexed by the length
59   // of the other two edges. Thus, to use the table we have to identify the
60   // longest edge and scale appropriately.
61   T smallEdge1 = vtkm::Min(edgeLengths[0], edgeLengths[1]);
62   T tmpEdge = vtkm::Max(edgeLengths[0], edgeLengths[1]);
63   T smallEdge2 = vtkm::Min(edgeLengths[2], tmpEdge);
64   T largeEdge = vtkm::Max(edgeLengths[2], tmpEdge);
65
66   smallEdge1 /= largeEdge;
```

```
67    smallEdge2 /= largeEdge;
68
69    // Find index into array.
70    vtkm::Id index1 = static_cast<vtkm::Id>(
71          vtkm::Floor(smallEdge1*(TRIANGLE_QUALITY_TABLE_DIMENSION-1)+0.5));
72    vtkm::Id index2 = static_cast<vtkm::Id>(
73          vtkm::Floor(smallEdge2*(TRIANGLE_QUALITY_TABLE_DIMENSION-1)+0.5));
74    vtkm::Id totalIndex = index1 + index2*TRIANGLE_QUALITY_TABLE_DIMENSION;
75
76    return triangleQualityPortal.Get(totalIndex);
77  }
78
79  struct TriangleQualityWorklet : vtkm::worklet::WorkletMapPointToCell
80  {
81    typedef void ControlSignature(TopologyIn cells,
82                                  FieldInPoint<Vec3> pointCoordinates,
83                                  WholeArrayIn<Scalar> triangleQualityTable,
84                                  FieldOutCell<Scalar> triangleQuality);
85    typedef _4 ExecutionSignature(CellShape, _2, _3);
86    typedef _1 InputDomain;
87
88    template<typename CellShape,
89             typename PointCoordinatesType,
90             typename TriangleQualityTablePortalType>
91    VTKM_EXEC_EXPORT
92    vtkm::Float32 operator()(
93        CellShape shape,
94        const PointCoordinatesType &pointCoordinates,
95        const TriangleQualityTablePortalType &triangleQualityTable) const
96    {
97      if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
98      {
99        this->RaiseError("Only triangles are supported for triangle quality.");
100       return vtkm::Nan32();
101     }
102
103     return LookupTriangleQuality(triangleQualityTable,
104                                  pointCoordinates[0],
105                                  pointCoordinates[1],
106                                  pointCoordinates[2]);
107   }
108 };
109
110 // Normally we would encapsulate this call in a filter, but for demonstrative
111 // purposes we are just calling the worklet directly.
112 template<typename DeviceAdapterTag>
113 VTKM_CONT_EXPORT
114 vtkm::cont::ArrayHandle<vtkm::Float32>
115 RunTriangleQuality(vtkm::cont::DataSet dataSet,
116                    DeviceAdapterTag)
117 {
118   vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualityTable =
119       GetTriangleQualityTable();
120
121   vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
122
123   vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet,DeviceAdapterTag>
124       dispatcher;
125   dispatcher.Invoke(dataSet.GetCellSet(),
126                     dataSet.GetCoordinateSystem().GetData(),
127                     triangleQualityTable,
128                     triangleQualities);
129
130   return triangleQualities;
```

```
131 }
```

14.7 Execution Objects

Although passing whole arrays into a worklet is a convenient way to provide data to a worklet that is not divided by the input or output domain, it is sometimes not the best structure to represent data. Thus, all worklets support a second type of argument called an *execution object*, or exec object for short, that passes the given 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::exec::ExecutionObjectBase`. Also, it must be possible to copy the object from the control environment to the execution environment and be usable in the execution environment, and any method of the execution object used within the worklet must be declared with `VTKM_EXEC_EXPORT` or `VTKM_EXEC_CONT_EXPORT`.

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 `PrepareForInput` method in `vtkm::cont::ArrayHandle` as described in Section **??**. Other VTK-m data objects, such as the subclasses of `vtkm::cont::CellSet`, have similar methods.

Returning to the example we have in Section 14.6, we are computing triangle quality quickly by looking up the value in a table. In Example 14.9 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 that has the table stored inside and methods to compute the triangle quality. The following example uses the table built in Example 14.9 to create such an object.

Example 14.10: Using `ExecObject` to access a lookup table in a worklet.

```
1  template<typename DeviceAdapterTag>
2  class TriangleQualityTable : public vtkm::exec::ExecutionObjectBase
3  {
4  public:
5    VTKM_CONT_EXPORT
6    TriangleQualityTable()
7    {
8      this->TablePortal =
9          GetTriangleQualityTable().PrepareForInput(DeviceAdapterTag());
10   }
11
12   template<typename T>
13   VTKM_EXEC_EXPORT
14   vtkm::Float32 GetQuality(const vtkm::Vec<T,3> &point1,
15                            const vtkm::Vec<T,3> &point2,
16                            const vtkm::Vec<T,3> &point3) const
17   {
18     return LookupTriangleQuality(this->TablePortal, point1, point2, point3);
19   }
20
21 private:
22   typedef vtkm::cont::ArrayHandle<vtkm::Float32> TableArrayType;
23   typedef typename TableArrayType::ExecutionTypes<DeviceAdapterTag>::PortalConst
24       TableArrayPortalType;
25   TableArrayPortalType TablePortal;
26 };
27
28 struct TriangleQualityWorklet2 : vtkm::worklet::WorkletMapPointToCell
```

```
29 {
30   typedef void ControlSignature(TopologyIn cells,
31                                 FieldInPoint<Vec3> pointCoordinates,
32                                 ExecObject triangleQualityTable,
33                                 FieldOutCell<Scalar> triangleQuality);
34   typedef _4 ExecutionSignature(CellShape, _2, _3);
35   typedef _1 InputDomain;
36
37   template<typename CellShape,
38            typename PointCoordinatesType,
39            typename TriangleQualityTableType>
40   VTKM_EXEC_EXPORT
41   vtkm::Float32 operator()(
42       CellShape shape,
43       const PointCoordinatesType &pointCoordinates,
44       const TriangleQualityTableType &triangleQualityTable) const
45   {
46     if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
47     {
48       this->RaiseError("Only triangles are supported for triangle quality.");
49       return vtkm::Nan32();
50     }
51
52     return triangleQualityTable.GetQuality(pointCoordinates[0],
53                                            pointCoordinates[1],
54                                            pointCoordinates[2]);
55   }
56 };
57
58 // Normally we would encapsulate this call in a filter, but for demonstrative
59 // purposes we are just calling the worklet directly.
60 template<typename DeviceAdapterTag>
61 VTKM_CONT_EXPORT
62 vtkm::cont::ArrayHandle<vtkm::Float32>
63 RunTriangleQuality2(vtkm::cont::DataSet dataSet,
64                     DeviceAdapterTag)
65 {
66   TriangleQualityTable<DeviceAdapterTag> triangleQualityTable;
67
68   vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
69
70   vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet2,DeviceAdapterTag>
71       dispatcher;
72   dispatcher.Invoke(dataSet.GetCellSet(),
73                     dataSet.GetCoordinateSystem().GetData(),
74                     triangleQualityTable,
75                     triangleQualities);
76
77   return triangleQualities;
78 }
```

14.8 Scatter

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::WorkletMapPointToCell` 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* for a worklet. 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. The worklet provides a number at runtime that defines 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 worklet provides 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.

To define a scatter procedure, the worklet must provide two items. The first item is a typedef named ScatterType. The ScatterType must be typedefed to one of the aforementioned Scatter* classes. The second item is a const method named GetScatter that returns an object of type ScatterType.

Example 14.11: Declaration of a scatter type in a worklet.

```
1   typedef vtkm::worklet::ScatterCounting ScatterType;
2 
3   VTKM_CONT_EXPORT
4   ScatterType GetScatter() const { return this->Scatter; }
```

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.

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::cont::ScatterUniform of size 2 and using the VisitIndex to determine from which array to pull a value.

Example 14.12: Using ScatterUniform.

```
1  struct InterleaveArrays : vtkm::worklet::WorkletMapField
2  {
3    typedef void ControlSignature(FieldIn<>, FieldIn<>, FieldOut<>);
4    typedef void ExecutionSignature(_1, _2, _3, VisitIndex);
5    typedef _1 InputDomain;
6 
7    typedef vtkm::worklet::ScatterUniform ScatterType;
8 
9    VTKM_CONT_EXPORT
10   ScatterType GetScatter() const { return vtkm::worklet::ScatterUniform(2); }
11 
12   template<typename T>
13   VTKM_EXEC_EXPORT
14   void operator()(const T &input0,
15                   const T &input1,
16                   T &output,
17                   vtkm::IdComponent visitIndex) const
18   {
```

DRAFT

```
19      if (visitIndex == 0)
20      {
21        output = input0;
22      }
23      else // visitIndex == 1
24      {
25        output = input1;
26      }
27    }
28  };
```

The second example takes a collection of point coordinates and clips them by an axis-aligned bounding box. It does this using a vtkm::cont::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 14.13: Using ScatterCounting.

```
1  class ClipPointsCount : public vtkm::worklet::WorkletMapField
2  {
3  public:
4    typedef void ControlSignature(FieldIn<Vec3> points,
5                                  FieldOut<IdComponentType> count);
6    typedef _2 ExecutionSignature(_1);
7    typedef _1 InputDomain;
8
9    template<typename T>
10   VTKM_CONT_EXPORT
11   ClipPointsCount(const vtkm::Vec<T,3> &boundsMin,
12                   const vtkm::Vec<T,3> &boundsMax)
13     : BoundsMin(boundsMin[0], boundsMin[1], boundsMin[2]),
14       BoundsMax(boundsMax[0], boundsMax[1], boundsMax[2])
15   {  }
16
17   template<typename T>
18   VTKM_EXEC_EXPORT
19   vtkm::IdComponent operator()(const vtkm::Vec<T,3> &point) const
20   {
21     return static_cast<vtkm::IdComponent>((this->BoundsMin[0] < point[0]) &&
22                                           (this->BoundsMin[1] < point[1]) &&
23                                           (this->BoundsMin[2] < point[2]) &&
24                                           (this->BoundsMax[0] > point[0]) &&
25                                           (this->BoundsMax[1] > point[1]) &&
26                                           (this->BoundsMax[2] > point[2]));
27   }
28
29 private:
30   vtkm::Vec<vtkm::FloatDefault,3> BoundsMin;
31   vtkm::Vec<vtkm::FloatDefault,3> BoundsMax;
32 };
33
34 class ClipPointsGenerate : public vtkm::worklet::WorkletMapField
35 {
36 public:
37   typedef void ControlSignature(FieldIn<Vec3> inPoints,
38                                 FieldOut<Vec3> outPoints);
39   typedef void ExecutionSignature(_1, _2);
40   typedef _1 InputDomain;
41
42   typedef vtkm::worklet::ScatterCounting ScatterType;
43
44   VTKM_CONT_EXPORT
45   ScatterType GetScatter() const { return this->Scatter; }
46
```

```
47    template<typename CountArrayType, typename DeviceAdapterTag>
48    VTKM_CONT_EXPORT
49    ClipPointsGenerate(const CountArrayType &countArray, DeviceAdapterTag)
50      : Scatter(countArray, DeviceAdapterTag())
51    {
52      VTKM_IS_ARRAY_HANDLE(CountArrayType);
53    }
54
55    template<typename InType, typename OutType>
56    VTKM_EXEC_EXPORT
57    void operator()(const vtkm::Vec<InType,3> &inPoint,
58                    vtkm::Vec<OutType,3> &outPoint) const
59    {
60      // The scatter ensures that this method is only called for input points
61      // that are passed to the output (where the count was 1). Thus, in this
62      // case we know that we just need to copy the input to the output.
63      outPoint = vtkm::Vec<OutType,3>(inPoint[0], inPoint[1], inPoint[2]);
64    }
65
66  private:
67    ScatterType Scatter;
68  };
69
70  // Normally we would encapsulate these calls in a filter, but for demonstrative
71  // purposes we are just calling the worklet directly.
72  template<typename T, typename Storage, typename DeviceAdapterTag>
73  VTKM_CONT_EXPORT
74  vtkm::cont::ArrayHandle<vtkm::Vec<T,3> >
75  RunClipPoints(const vtkm::cont::ArrayHandle<vtkm::Vec<T,3>, Storage> &pointArray,
76                vtkm::Vec<T,3> boundsMin,
77                vtkm::Vec<T,3> boundsMax,
78                DeviceAdapterTag)
79  {
80    vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray;
81
82    ClipPointsCount workletCount(boundsMin, boundsMax);
83    vtkm::worklet::DispatcherMapField<ClipPointsCount, DeviceAdapterTag>
84        dispatcherCount(workletCount);
85    dispatcherCount.Invoke(pointArray, countArray);
86
87    vtkm::cont::ArrayHandle<vtkm::Vec<T,3> > clippedPointsArray;
88
89    ClipPointsGenerate workletGenerate(countArray, DeviceAdapterTag());
90    vtkm::worklet::DispatcherMapField<ClipPointsGenerate, DeviceAdapterTag>
91        dispatcherGenerate(workletGenerate);
92    dispatcherGenerate.Invoke(pointArray, clippedPointsArray);
93
94    return clippedPointsArray;
95  }
```

14.9 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 14.14: Raising an error in the execution environment.

```
struct SquareRoot : vtkm::worklet::WorkletMapField
{
public:
  typedef void ControlSignature(FieldIn<Scalar>, FieldOut<Scalar>);
  typedef _2 ExecutionSignature(_1);

  template<typename T>
  VTKM_EXEC_EXPORT
  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 5.4 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.

DRAFT

CHAPTER

FIFTEEN

CREATING FILTERS

DRAFT

CHAPTER

SIXTEEN

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.

16.1 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.

DRAFT

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.

DRAFT

`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 $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards 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).

16.2 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::Vecs 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.

16.3 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 16.1: Creating a Matrix.

```
1  vtkm::Matrix<vtkm::Float32, 2, 3> matrix;
2
3  // Using parenthesis notation.
4  matrix(0,0) = 0.0f;
5  matrix(0,1) = 1.0f;
6  matrix(0,2) = 2.0f;
7
8  // Using bracket notation.
9  matrix[1][0] = 10.0f;
10 matrix[1][1] = 11.0f;
11 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.

DRAFT

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.

16.4 Newton's Method

VTK-m's matrix methods (documented in Section 16.3) 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.

Example 16.2: Using NewtonsMethod to solve a small system of nonlinear equations.

```
1 // A functor for the mathematical function f(x) = [dot(x,x),x[0]*x[1]]
2 struct FunctionFunctor
3 {
4   template<typename T>
5   VTKM_EXEC_CONT_EXPORT
6   vtkm::Vec<T,2> operator()(const vtkm::Vec<T,2> &x) const
7   {
8     return vtkm::make_Vec(vtkm::dot(x,x), x[0]*x[1]);
```

```
9     }
10  };
11
12  // A functor for the Jacobian of the mathematical function
13  // f(x) = [dot(x,x),x[0]*x[1]], which is
14  //   | 2*x[0] 2*x[1] |
15  //   |   x[1]   x[0] |
16  struct JacobianFunctor
17  {
18    template<typename T>
19    VTKM_EXEC_CONT_EXPORT
20    vtkm::Matrix<T,2,2> operator()(const vtkm::Vec<T,2> &x) const
21    {
22      vtkm::Matrix<T,2,2> jacobian;
23      jacobian(0,0) = 2*x[0];
24      jacobian(0,1) = 2*x[1];
25      jacobian(1,0) = x[1];
26      jacobian(1,1) = x[0];
27
28      return jacobian;
29    }
30  };
31
32  VTKM_EXEC_EXPORT
33  void SolveNonlinear()
34  {
35    // Use Newton's method to solve the nonlinear system of equations:
36    //
37    //    x^2 + y^2 = 2
38    //    x*y = 1
39    //
40    // There are two possible solutions, which are (x=1,y=1) and (x=-1,y=-1).
41    // The one found depends on the starting value.
42    vtkm::Vec<vtkm::Float32,2> answer1 =
43        vtkm::NewtonsMethod(JacobianFunctor(),
44                            FunctionFunctor(),
45                            vtkm::make_Vec(2.0f, 1.0f),
46                            vtkm::make_Vec(1.0f, 0.0f));
47    // answer1 is [1,1]
48
49    vtkm::Vec<vtkm::Float32,2> answer2 =
50        vtkm::NewtonsMethod(JacobianFunctor(),
51                            FunctionFunctor(),
52                            vtkm::make_Vec(2.0f, 1.0f),
53                            vtkm::make_Vec(0.0f, -2.0f));
54    // answer2 is [-1,-1]
55  }
```

CHAPTER

SEVENTEEN

WORKING WITH CELLS

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See Section 11.2 starting on page 90 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.

17.1 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 17.1 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.

17.1.1 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

DRAFT

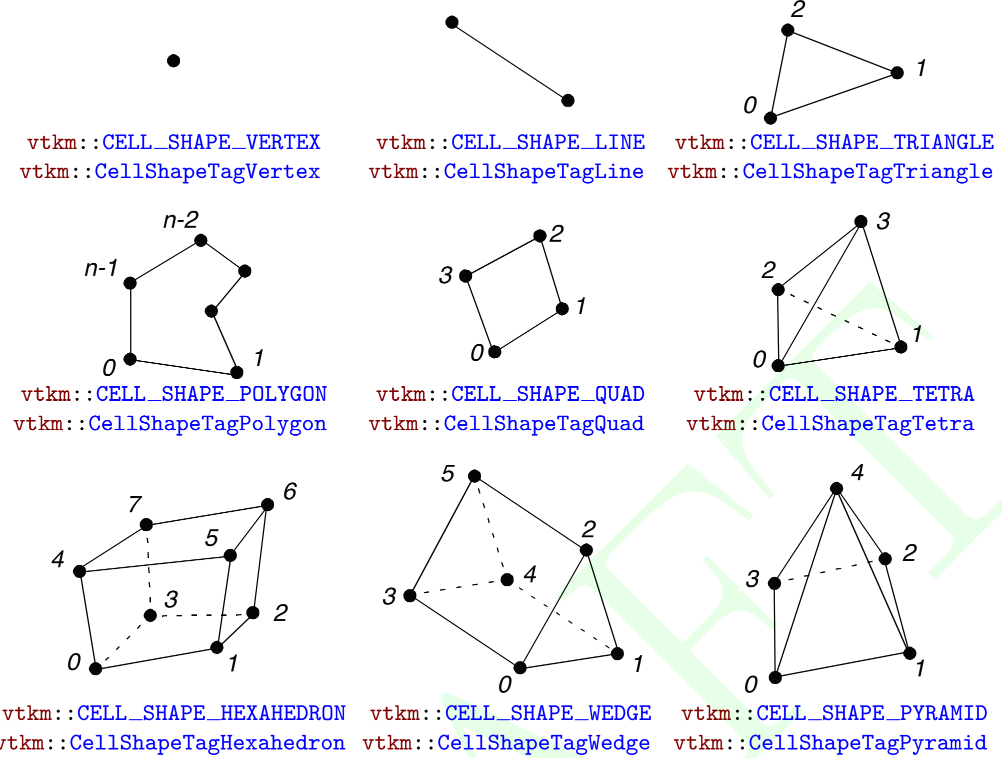


Figure 17.1: Basic Cell Shapes

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 17.1: Using CellShapeIdToTag.

```
1  void CellFunction(vtkm::CellShapeTagTriangle)
2  {
3    std::cout << "In CellFunction for triangles." << std::endl;
4  }
5
6  void DoSomethingWithACell()
7  {
8    // Calls CellFunction overloaded with a vtkm::CellShapeTagTriangle.
9    CellFunction(vtkm::CellShapeIdToTag<vtkm::CELL_SHAPE_TRIANGLE>::Tag());
10 }
```

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 17.2.

DRAFT

17.1.2 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 17.2: Using CellTraits to implement a polygon normal estimator.

```
1  namespace detail {
2
3  VTKM_SUPPRESS_EXEC_WARNINGS
4  template<typename PointCoordinatesVector, typename WorkletType>
5  VTKM_EXEC_CONT_EXPORT
6  typename PointCoordinatesVector::ComponentType
7  CellNormalImpl(const PointCoordinatesVector &pointCoordinates,
8                 vtkm::CellTopologicalDimensionsTag<2>,
9                 const WorkletType &worklet)
10 {
11   if (pointCoordinates.GetNumberOfComponents() >= 3)
12   {
13     return vtkm::TriangleNormal(pointCoordinates[0],
14                                 pointCoordinates[1],
15                                 pointCoordinates[2]);
16   }
17   else
18   {
19     worklet.RaiseError("Degenerate polygon.");
20     return typename PointCoordinatesVector::ComponentType();
21   }
22 }
23
24 VTKM_SUPPRESS_EXEC_WARNINGS
25 template<typename PointCoordinatesVector,
26          vtkm::IdComponent Dimensions,
27          typename WorkletType>
28 VTKM_EXEC_CONT_EXPORT
29 typename PointCoordinatesVector::ComponentType
30 CellNormalImpl(const PointCoordinatesVector &,
31                vtkm::CellTopologicalDimensionsTag<Dimensions>,
32                const WorkletType &worklet)
33 {
34   worklet.RaiseError("Only polygons supported for cell normals.");
35   return typename PointCoordinatesVector::ComponentType();
36 }
37
38 } // namespace detail
```

```
39
40 VTKM_SUPPRESS_EXEC_WARNINGS
41 template<typename CellShape,
42          typename PointCoordinatesVector,
43          typename WorkletType>
44 VTKM_EXEC_CONT_EXPORT
45 typename PointCoordinatesVector::ComponentType
46 CellNormal(CellShape,
47            const PointCoordinatesVector &pointCoordinates,
48            const WorkletType &worklet)
49 {
50   return detail::CellNormalImpl(
51         pointCoordinates,
52         typename vtkm::CellTraits<CellShape>::TopologicalDimensionsTag(),
53         worklet);
54 }
55
56 VTKM_SUPPRESS_EXEC_WARNINGS
57 template<typename PointCoordinatesVector,
58          typename WorkletType>
59 VTKM_EXEC_CONT_EXPORT
60 typename PointCoordinatesVector::ComponentType
61 CellNormal(vtkm::CellShapeTagGeneric shape,
62            const PointCoordinatesVector &pointCoordinates,
63            const WorkletType &worklet)
64 {
65   switch(shape.Id)
66   {
67     vtkmGenericCellShapeMacro(
68           return CellNormal(CellShapeTag(), pointCoordinates, worklet));
69     default:
70       worklet.RaiseError("Unknown cell type.");
71       return typename PointCoordinatesVector::ComponentType();
72   }
73 }
```

17.2 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).

17.3 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 17.3: Interpolating field values to a cell's center.

```
struct CellCenters : vtkm::worklet::WorkletMapPointToCell
{
  typedef void ControlSignature(TopologyIn,
                                FieldInPoint<> inputField,
                                FieldOutCell<> outputField);
  typedef void ExecutionSignature(CellShape, PointCount, _2, _3);
  typedef _1 InputDomain;

  template<typename CellShapeTag,typename FieldInVecType,typename FieldOutType>
  VTKM_EXEC_EXPORT
  void operator()(CellShapeTag shape,
                  vtkm::IdComponent pointCount,
                  const FieldInVecType &inputField,
                  FieldOutType &outputField) const
  {
    vtkm::Vec<vtkm::FloatDefault,3> center =
        vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField = vtkm::exec::CellInterpolate(inputField, center, shape, *this);
  }
};
```

17.4 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 17.4: Computing the derivative of the field at cell centers.

```
1  struct CellDerivatives : vtkm::worklet::WorkletMapPointToCell
2  {
3    typedef void ControlSignature(TopologyIn,
4                                  FieldInPoint<> inputField,
5                                  FieldInPoint<Vec3> pointCoordinates,
6                                  FieldOutCell<> outputField);
7    typedef void ExecutionSignature(CellShape, PointCount, _2, _3, _4);
8    typedef _1 InputDomain;
9
10   template<typename CellShapeTag,
11            typename FieldInVecType,
12            typename PointCoordVecType,
13            typename FieldOutType>
14   VTKM_EXEC_EXPORT
15   void operator()(CellShapeTag shape,
16                   vtkm::IdComponent pointCount,
17                   const FieldInVecType &inputField,
18                   const PointCoordVecType &pointCoordinates,
19                   FieldOutType &outputField) const
20   {
21     vtkm::Vec<vtkm::FloatDefault,3> center =
22         vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
23     outputField = vtkm::exec::CellDerivative(inputField,
24                                              pointCoordinates,
25                                              center,
26                                              shape,
27                                              *this);
28   }
29 };
```

DRAFT

Part IV

Advanced Development

DRAFT

CHAPTER

EIGHTEEN

ADVANCED WORKLET CUSTOMIZATION

Chapter 14 describes the basics of creating and using worklets. Many visualization algorithms can be implemented using VTK-m's existing worklet types and features. However, new algorithms and designs may require features not provided by VTK-m's current worklet set. In such cases it is possible to directly design filters using the lower level device adapter operations [AS DESCRIBED IN SECTION BLA]. But by adding features to the worklet mechanisms, new designs can be integrated better with the other VTK-m features and can be repurposed in interesting ways for other algorithms.

This chapter provides the information necessary to create new mechanisms for worklets. It first describes the interface for getting data from the control environment objects to the data passed to a worklet invocation and back. It then describes how to modify these mechanisms to create new data movement structures and new worklet types.

18.1 Transferring Arguments from Control to Execution

From the `ControlSignature` and `ExecutionSignature` defined in worklets, VTK-m uses template metaprogramming to build the code required to manage data from control to execution environment. This management is handled by three classes that provide type checking, transportation, and fetching.

[I'VE BEEN THINKING THAT ONE MORE FEATURE THAT THESE CLASSES SHOULD PROVIDE IS THE ABILITY TO RETURN THE SIZE OF THE DOMAIN. THAT WOULD MAKE THINGS SIMPLER AND SAFER FOR GETTING THE INPUT DOMAIN SIZE AND CHECKING THE REMAINING DOMAIN SIZES.]

18.1.1 Type Checks

Before attempting to move data from the control to the execution environment, the VTK-m dispatchers 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.

DRAFT

vtkm::cont::arg::TypeCheckTagArray True if the type is a vtkm::cont::ArrayHandle. TypeCheckTagArray also has a template parameter that is a type list. The ArrayHandle must also have a value type contained in this type list.

vtkm::cont::arg::TypeCheckTagExecObject True if the type is an execution object. All execution objects must derive from vtkm::exec::ExecutionObjectBase and must be copyable through memcpy or similar mechanism.

Here are some trivial examples of using TypeCheck. Typically these checks are done internally in the base VTK-m dispatcher code, so these examples are for demonstration only.

Example 18.1: Behavior of vtkm::cont::arg::TypeCheck.

```
1  struct MyExecObject : vtkm::exec::ExecutionObjectBase { vtkm::Id Value; };
2
3  void DoTypeChecks()
4  {
5    using vtkm::cont::arg::TypeCheck;
6    using vtkm::cont::arg::TypeCheckTagArray;
7    using vtkm::cont::arg::TypeCheckTagExecObject;
8
9    bool check1 = TypeCheck<TypeCheckTagExecObject, MyExecObject>::value; // true
10   bool check2 = TypeCheck<TypeCheckTagExecObject, vtkm::Id>::value;     // false
11
12   typedef vtkm::cont::ArrayHandle<vtkm::Float32> ArrayType;
13
14   bool check3 =                                                        // true
15       TypeCheck<TypeCheckTagArray<vtkm::TypeListTagField>, ArrayType>::value;
16   bool check4 =                                                        // false
17       TypeCheck<TypeCheckTagArray<vtkm::TypeListTagIndex>, ArrayType>::value;
18   bool check5 = TypeCheck<TypeCheckTagExecObject, ArrayType>::value;    // false
19 }
```

18.1.2 Transport

After all the argument types are checked, the base dispatcher 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 typedef 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 the control-side object and the size of the domain and returns an execution-side object. This operator is called in the control environment, and the returned object must be ready to be passed to 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::TransportTagArrayIn Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInput method. 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 returned execution object is an array portal.

`vtkm::cont::arg::TransportTagExecObject` Simply returns the given execution object, which should be ready to load onto the device.

Here are some trivial examples of using `Transport`. Typically this movement is done internally in the base VTK-m dispatcher code, so these examples are for demonstration only.

Example 18.2: Behavior of `vtkm::cont::arg::Transport`.

```
1  struct MyExecObject : vtkm::exec::ExecutionObjectBase { vtkm::Id Value; };
2
3  typedef vtkm::cont::ArrayHandle<vtkm::Id> ArrayType;
4
5  void DoTransport(const MyExecObject &inExecObject,
6                   const ArrayType &inArray,
7                   const ArrayType &outArray)
8  {
9    typedef VTKM_DEFAULT_DEVICE_ADAPTER_TAG Device;
10
11   using vtkm::cont::arg::Transport;
12   using vtkm::cont::arg::TransportTagArrayIn;
13   using vtkm::cont::arg::TransportTagArrayOut;
14   using vtkm::cont::arg::TransportTagExecObject;
15
16   // The executive object transport just passes the object through.
17   typedef Transport<TransportTagExecObject,MyExecObject,Device>
18       ExecObjectTransport;
19   MyExecObject passedExecObject = ExecObjectTransport()(inExecObject, 10);
20
21   // The array in transport returns a read-only array portal.
22   typedef Transport<TransportTagArrayIn,ArrayType,Device> ArrayInTransport;
23   ArrayInTransport::ExecObjectType inPortal = ArrayInTransport()(inArray, 10);
24
25   // The array out transport returns an allocated array portal.
26   typedef Transport<TransportTagArrayOut,ArrayType,Device> ArrayOutTransport;
27   ArrayOutTransport::ExecObjectType outPortal =ArrayOutTransport()(outArray,10);
28 }
```

18.1.3 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 parameter is an `Invocation` type that provides details about how the worklet is being dispatched including a list of execution object parameters passed to the invocation. The fourth parameter is a `vtkm::IdComponent` that points to the invocation parameter that the data should be fetched from.

A `Fetch` contains a `typedef` 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` `typedef` 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.

[Don't forget to add index entries for both fetch and aspect where appropriate.]

`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::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.

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::AspectTagWorkIndex` Simply returns the domain (or thread) index ignoring any associated data. This aspect is used to implement the `WorkIndex` execution signature tag.

18.2 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 `Invoke` method of the dispatcher is expected to support arguments that match these arguments, and part of the dispatching operation may require these arguments to be augmented before the worklet is scheduled. This leaves dispatchers with the tricky task of managing some collection of arguments of unknown size and unknown types.

[`FunctionInterface` is in the `vtkm::internal` interface. I still can't decide if it should be moved to the `vtkm` interface.]

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`.

18.2.1 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 18.3: Declaring `vtkm::internal::FunctionInterface`.

```
1 // FunctionInterfaces matching some common POSIX functions.
2 vtkm::internal::FunctionInterface<size_t(const char *)>
3     strlenInterface;
4
5 vtkm::internal::FunctionInterface<char *(char *, const char *s2, size_t)>
6     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 18.4: 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);
```

18.2.2 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 18.5: Getting the arity of a FunctionInterface.

```
BOOST_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 18.6: 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;
}
```

DRAFT

Likewise, there is a `SetParmeter` method for changing parameters. The same rules for indexing and template specification apply.

Example 18.7: Using `FunctionInterface::SetParameter()`.

```
1  void SetFirstParameterResolved(
2      vtkm::internal::FunctionInterface<void(std::string)> &interface,
3      const std::string &newFirstParameter)
4  {
5    // The following two uses of SetParameter are equivalent
6    interface.SetParameter<1>(newFirstParameter);
7    interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
8  }
9
10 template<typename FunctionSignature, typename T>
11 void SetFirstParameterTemplated(
12     vtkm::internal::FunctionInterface<FunctionSignature> &interface,
13     T newFirstParameter)
14 {
15   // The following two uses of SetParameter are equivalent
16   interface.template SetParameter<1>(newFirstParameter);
17   interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
18 }
```

18.2.3 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 18.8: Invoking a `FunctionInterface`.

```
1    vtkm::internal::FunctionInterface<size_t(const char *)> strlenInterface;
2    strlenInterface.SetParameter<1>("Hello world");
3
4    strlenInterface.InvokeCont(strlen);
5
6    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 18.9: Invoking a `FunctionInterface` with a transform.

```
1  // Our transform converts C strings to integers, leaves everything else alone.
2  struct TransformFunctor
3  {
4    template<typename T>
5    VTKM_CONT_EXPORT
6    const T &operator()(const T &x) const
7    {
8      return x;
9    }
10
11   VTKM_CONT_EXPORT
12   const vtkm::Int32 operator()(const char *x) const
13   {
14     return atoi(x);
```

```
15      }
16    };
17
18    // The function we are invoking simply compares two numbers.
19    struct IsSameFunctor
20    {
21      template<typename T1, typename T2>
22      VTKM_CONT_EXPORT
23      bool operator()(const T1 &x, const T2 &y) const
24      {
25        return x == y;
26      }
27    };
28
29    void TryTransformedInvoke()
30    {
31      vtkm::internal::FunctionInterface<bool(const char *, vtkm::Int32)>
32          functionInterface =
33            vtkm::internal::make_FunctionInterface<bool>((const char *)"42",
34                                                         (vtkm::Int32)42);
35
36      functionInterface.InvokeCont(IsSameFunctor(), TransformFunctor());
37
38      bool isSame = functionInterface.GetReturnValue();     // isSame = true
39    }
```

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 18.10: Getting return value from `FunctionInterface` safely.

```
1     template<typename ResultType, bool Valid> struct PrintReturnFunctor;
2
3     template<typename ResultType>
4     struct PrintReturnFunctor<ResultType, true>
5     {
6       VTKM_CONT_EXPORT
7       void operator()(
8           const vtkm::internal::FunctionInterfaceReturnContainer<ResultType> &x)
9       const
10      {
11        std::cout << x.Value << std::endl;
12      }
13    };
14
15    template<typename ResultType>
16    struct PrintReturnFunctor<ResultType, false>
17    {
18      VTKM_CONT_EXPORT
19      void operator()(
20          const vtkm::internal::FunctionInterfaceReturnContainer<ResultType> &)
21      const
22      {
23        std::cout << "No return type." << std::endl;
24      }
25    };
```

```

template<typename FunctionInterfaceType>
void PrintReturn(const FunctionInterfaceType &functionInterface)
{
  typedef typename FunctionInterfaceType::ResultType ResultType;
  typedef vtkm::internal::FunctionInterfaceReturnContainer<ResultType>
      ReturnContainerType;

  PrintReturnFunctor<ResultType, ReturnContainerType::VALID> printReturn;
  printReturn(functionInterface.GetReturnValueSafe());
}
```

18.2.4 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 18.11: Appending parameters to a `FunctionInterface`.

```
  using vtkm::internal::FunctionInterface;
  using vtkm::internal::make_FunctionInterface;

  typedef FunctionInterface<void(std::string, vtkm::Id)>
      InitialFunctionInterfaceType;
  InitialFunctionInterfaceType initialFunctionInterface =
      make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

  typedef FunctionInterface<void(std::string, vtkm::Id, std::string)>
      AppendedFunctionInterfaceType1;
  AppendedFunctionInterfaceType1 appendedFunctionInterface1 =
      initialFunctionInterface.Append(std::string("foobar"));
  // appendedFunctionInterface1 has parameters ("Hello World", 42, "foobar")

  typedef InitialFunctionInterfaceType::AppendType<vtkm::Float32>::type
      AppendedFunctionInterfaceType2;
  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 18.12: Replacing parameters in a `FunctionInterface`.

```
  using vtkm::internal::FunctionInterface;
  using vtkm::internal::make_FunctionInterface;

  typedef FunctionInterface<void(std::string, vtkm::Id)>
      InitialFunctionInterfaceType;
  InitialFunctionInterfaceType initialFunctionInterface =
      make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

  typedef FunctionInterface<void(vtkm::Float32, vtkm::Id)>
      ReplacedFunctionInterfaceType1;
```

```
  ReplacedFunctionInterfaceType1 replacedFunctionInterface1 =
      initialFunctionInterface.Replace<1>(vtkm::Float32(3.141));
  // replacedFunctionInterface1 has parameters (3.141, 42)

  typedef InitialFunctionInterfaceType::ReplaceType<2, std::string>::type
      ReplacedFunctionInterfaceType2;
  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 18.13: 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.

  typedef typename FunctionInterfaceType::template ParameterType<1>::type
      ArrayValueType;

  // 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;
}
```

18.2.5 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 18.14: Using a static transform of function interface class.

```
struct ParametersToPointersFunctor {
  template<typename T, vtkm::IdComponent Index>
  struct ReturnType {
    typedef const T *type;
  };

  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  const T *operator()(const T &x, vtkm::internal::IndexTag<Index>) const {
    return &x;
  }
};

template<typename FunctionInterfaceType>
VTKM_CONT_EXPORT
typename FunctionInterfaceType::
    template StaticTransformType<ParametersToPointersFunctor>::type
ParametersToPointers(const 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 `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 18.15: Using a dynamic transform of a function interface.

```
struct UnpackNumbersTransformFunctor {
```

```
2    template<typename InputType,
3             typename ContinueFunctor,
4             vtkm::IdComponent Index>
5    VTKM_CONT_EXPORT
6    void operator()(const InputType &input,
7                    const ContinueFunctor &continueFunction,
8                    vtkm::internal::IndexTag<Index>) const
9    {
10     continueFunction(input);
11   }
12
13   template<typename ContinueFunctor, vtkm::IdComponent Index>
14   VTKM_CONT_EXPORT
15   void operator()(const std::string &input,
16                   const ContinueFunctor &continueFunction,
17                   vtkm::internal::IndexTag<Index>) const
18   {
19     if ((input[0] >= '0') && (input[0] <= '9'))
20     {
21       std::stringstream stream(input);
22       vtkm::FloatDefault value;
23       stream >> value;
24       continueFunction(value);
25     }
26     else
27     {
28       continueFunction(input);
29     }
30   }
31 };
32
33 struct UnpackNumbersFinishFunctor {
34   template<typename FunctionInterfaceType>
35   VTKM_CONT_EXPORT
36   void operator()(FunctionInterfaceType &functionInterface) const
37   {
38     // Do something
39   }
40 };
41
42 template<typename FunctionInterfaceType>
43 void DoUnpackNumbers(const FunctionInterfaceType &functionInterface)
44 {
45   functionInterface.DynamicTransformCont(UnpackNumbersTransformFunctor(),
46                                          UnpackNumbersFinishFunctor());
47 }
```

One common use for the FunctionInterface dynamic transform is to convert parameters of virtual polymorphic type like vtkm::cont::DynamicArrayHandle 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 18.16: Using DynamicTransform to cast dynamic arrays in a function interface.

```
1  template<typename Device>
2  struct ArrayCopyFunctor {
3    template<typename Signature>
4    VTKM_CONT_EXPORT
5    void operator()(
6        vtkm::internal::FunctionInterface<Signature> functionInterface) const
7    {
8      functionInterface.InvokeCont(*this);
9    }
10
```

```
  template<typename T, class CIn, class COut>
  VTKM_CONT_EXPORT
  void operator()(const vtkm::cont::ArrayHandle<T, CIn> &input,
                  vtkm::cont::ArrayHandle<T, COut> &output) const
  {
    vtkm::cont::DeviceAdapterAlgorithm<Device>::Copy(input, output);
  }

  template<typename TIn, typename TOut, class CIn, class COut>
  VTKM_CONT_EXPORT
  void operator()(const vtkm::cont::ArrayHandle<TIn, CIn> &,
                  vtkm::cont::ArrayHandle<TOut, COut> &) const
  {
    throw vtkm::cont::ErrorControlBadType(
          "Arrays to copy must be the same type.");
  }
};

template<typename Device>
void CopyDynamicArrays(vtkm::cont::DynamicArrayHandle input,
                       vtkm::cont::DynamicArrayHandle output,
                       Device)
{
  vtkm::internal::FunctionInterface<void(vtkm::cont::DynamicArrayHandle,
                                         vtkm::cont::DynamicArrayHandle)>
      functionInterface =
        vtkm::internal::make_FunctionInterface<void>(input, output);

  functionInterface.DynamicTransformCont(
        vtkm::cont::internal::DynamicTransform(), ArrayCopyFunctor<Device>());
}
```

18.2.6 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 18.17: Using the `ForEach` feature of `FunctionInterface`.

```
struct PrintArgumentFunctor{
  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT_EXPORT
  void operator()(const T &argument, vtkm::internal::IndexTag<Index>) const
  {
    std::cout << Index << ":" << argument << " ";
  }
};

template<typename FunctionInterfaceType>
VTKM_CONT_EXPORT
void PrintArguments(const FunctionInterfaceType &functionInterface)
{
  std::cout << "( ";
```

```
15      functionInterface.ForEachCont(PrintArgumentFunctor());
16      std::cout << ")" << std::endl;
17    }
```

18.3 Invocation Objects

18.4 Creating New ControlSignature Tags

18.5 Creating New ExecutionSignature Tags

18.6 Creating New Worklet Types

18.6.1 New Worklet Superclasses

18.6.2 Dispatch Workflow

18.6.3 New Dispatch Classes

DRAFT

Part V

Appendix

DRAFT

APPENDIX

A

CODING CONVENTIONS

Several developers contribute to VTK-m and we welcome others who are interested to also contribute to the project. To ensure readability and consistency in the code, we have adopted the following coding conventions. Many of these conventions are adapted from the coding conventions of the VTK project. This is because many of the developers are familiar with VTK coding and because we expect VTK-m to have continual interaction with VTK.

- All code contributed to VTK-m must be compatible with VTK-m's BSD license.
- Copyright notices should appear at the top of all source, configuration, and text files. The statement should have the following form (with the year replaced with the year the file was created):

  ```
  //============================================================================
  //  Copyright (c) Kitware, Inc.
  //  All rights reserved.
  //  See LICENSE.txt for details.
  //  This software is distributed WITHOUT ANY WARRANTY; without even
  //  the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
  //  PURPOSE.  See the above copyright notice for more information.
  //
  //  Copyright 2014 Sandia Corporation.
  //  Copyright 2014 UT-Battelle, LLC.
  //  Copyright 2014. Los Alamos National Security
  //
  //  Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation,
  //  the U.S. Government retains certain rights in this software.
  //
  //  Under the terms of Contract DE-AC52-06NA25396 with Los Alamos National
  //  Laboratory (LANL), the U.S. Government retains certain rights in
  //  this software.
  //============================================================================
  ```

  The CopyrightStatement test checks all files for a similar statement. The test will print out a suggested text that can be copied and pasted to any file that has a missing copyright statement (with appropriate replacement of comment prefix). Exceptions to this copyright statement (for example, third-party files with different but compatible statements) can be added to LICENSE.txt.

- All include files should use include guards. starting right after the copyright statement. The naming convention of the include guard macro is that it should start with `vtk_m` be followed with the path name, starting from the top-level source code directory under vtkm, with non alphanumeric characters, such as `/` and `.` replaced with underscores. The `#endif` part of the guard at the bottom of the file should include the guard name in a comment. For example, the vtkm/cont/ArrayHandle.h header contains the guard

```
#ifndef vtk_m_cont_ArrayHandle_h
#define vtk_m_cont_ArrayHandle_h
```

at the top and

```
#endif //vtk_m_cont_ArrayHandle_h
```

- VTK-m has several nested namespaces. The declaration of each namespace should be on its own line, and the code inside the namespace bracket should not be indented. The closing brace at the bottom of the namespace should be documented with a comment identifying the namespace. Namespaces can be grouped as desired. The following is a valid use of namespaces.

```
namespace vtkm {
namespace cont {

namespace detail {

class InternalClass;

} // namespace detail

class ExposedClass;

}
} // namespace vtkm::cont
```

- Multiple inheritance is not allowed in VTK-m classes.
- Any functional public class should be in its own header file with the same name as the class. The file should be in a directory that corresponds to the namespace the class is in. There are several exceptions to this rule.
  - Templated classes and template specialization often require the implementation of the class to be broken into pieces. Sometimes a specialization is placed in a header with a different name.
  - Many VTK-m toolkit features are not encapsulated in classes. Functions may be collected by purpose or co-located with associated class.
  - Although tags are technically classes, they behave as an enumeration for the compiler. Multiple tags that make up this enumeration are collected together.
  - Some classes, such as `vtkm::Vec` are meant to behave as basic types. These are sometimes collected together as if they were related `typedef`s. The `vtkm/Types.h` header is a good example of this.
- The indentation follows the Allman style. The curly brace (scope delimiter) for a block is placed on the line following the prototype or control statement and is indented with the outer scope (i.e. the curly brace does not line up with the code in the block). This differs from VTK style, but was agreed on by the developers as the more common style. Indentations are two spaces.
- Conditional clauses (including loop conditionals such as `for` and `while`) must be in braces below the conditional. That is, instead of

```
if (test) { clause; }
```

use

```
if (test)
{
  clause;
}
```

DRAFT

The rational for this requirement is to make it obvious whether the clause is executed when stepping through the code with the debugger. The one exception to this rule is when the clause contains a control-flow statement with obvious side effects such as `return` or `break`. However, even if the clause contains a single statement and is on the same line, the clause should be surrounded by braces.

- Use two space indentation.
- Tabs are not allowed. Only use spaces for indentation. No one can agree on what the size of a tab stop is, so it is better to not use them at all.
- There should be no trailing whitespace in any line.
- Use only alphanumeric characters in names. Use capitalization to demarcate words within a name (camel case). The exception is preprocessor macros and constant numbers that are, by convention, represented in all caps and a single underscore to demarcate words.
- Namespace names are in all lowercase. They should be a single word that designates its meaning.
- All class, method, member variable, and functions should start with a capital letter. Local variables should start in lower case and then use camel case. Exceptions can be made when such naming would conflict with previously established conventions in other library. (For example, `make_ArrayHandle` corresponds to `make_pair` in the standard template library.)
- All class, function, and member names that have multiple words in their descriptions should be listed from general to specific. For example, if a class is a k-d tree that is used to locate points, the preferred name would be `LocatorPointKDTree`. This naming convention makes it easier to find both known and unknown classes in alphabetic lists.
- Always spell out words in names; do not use abbreviations except in cases where the shortened form is widely understood and a name in its own right (e.g. OpenMP).
- Always use descriptive names in all identifiers, including local variable names. Particularly avoid meaningless names of a few characters (e.g. `x`, `foo`, or `tmp`) or numbered names with no meaning to the number or order (e.g. `value1`, `value2`,...). Also avoid the meaningless for loop variable names `i`, `j`, `k`, etc. Instead, use a name that identifies what type of index is being referenced such as `pointIndex`, `vertexIndex`, `componentIndex`, etc.
- Classes are documented with Doxygen-style comments before classes, methods, and functions.
- Exposed classes should not have public instance variables outside of exceptional situations. Access is given by convention through methods with names starting with `Set` and `Get` or through overloaded operators.
- References to classes and functions should be fully qualified with the namespace. This makes it easier to establish classes and functions from different packages and to find source and documentation for the referenced class. As an exception, if one class references an internal or detail class clearly associated with it, the reference can be shortened to `internal::` or `detail::`.
- use `this->` inside of methods when accessing class methods and instance variables to distinguish between local variables and instance variables.
- Include statements should generally be in alphabetical order. They can be grouped by package and type.
- Namespaces should not be brought into global scope or the scope of any VTK-m package namespace with the "using" keyword. It should also be avoided in class, method, and function scopes (fully qualified namespace references are preferred).
- All code must be valid by the C++03 and C++11 specifications. It must also compile on older compilers that support C++98. Code that uses language features not available in C++98 must have a second implementation that works around the limitations of C++98. The VTKM_FORCE_ANSI turns on a compiler check for ANSI compatibility in gcc and clang compilers.
- Limit all lines to 80 characters whenever possible.

DRAFT

- New code must include regression tests that will run on the dashboards. Generally a new class will have an associated "UnitTest" that will test the operation of the test directly. There may be other tests necessary that exercise the operation with different components or on different architectures.

- All code must compile and run without error or warning messages on the nightly dashboards, which should include Windows, Mac, and Linux.

- Use `vtkm::Id` in lieu of `int` or `long` for data structure indices and `vtkm::IdComponent` for component indices of `vtkm::Vec` and related classes (like `vtkm::VecVariable` and `vtkm::Matrix`).

- Whenever possible, use templates to resolve data types like `float`, `double`, or vectors to make code as flexible as possible. If a specific data type is required, prefer the VTK-m–provided types like `vtkm::Float32` and `vtkm::Float64` over the standard C types like `float` or `double`. `vtkm::FloatDefault` can be used in cases where there is no reasonable way to specify data precision (for example, when generating coordinates for uniform grids), but should be use sparingly.

- All functions and methods defined within the Dax toolkit should be declared with `VTKM_CONT_EXPORT`, `VTKM_EXEC_EXPORT`, or `VTKM_EXEC_CONT_EXPORT`.

We should note that although these conventions impose a strict statute on VTK-m coding, these rules (other than those involving licensing and copyright) are not meant to be dogmatic. Examples can be found in the existing code that break these conventions, particularly when the conventions stand in the way of readability (which is the point in having them in the first place). For example, it is often the case that it is more readable for a complicated `typedef` to stretch a few characters past 80 even if it pushes past the end of a display.

DRAFT

INDEX

π, 133, 134
_1, 107, 109, 113, 115, 120
_2, 107, 109, 113, 115, 120
__device__, 18
__host__, 18

Abs, 131
absolute value, 131
ACos, 131
ACosH, 131
algorithm, 40–42, 45–49
Allocate, 33, 40
AllTypes, 106
arccosine, 131
arcsine, 131
arctangent, 131
arg namespace, 147–149
arity, 151
ArrayHandle, xiii, xiv, 17, 29, 30, 33, 39, 53, 54, 72, 73, 76, 123, 148
ArrayHandle.h, 17, 163
ArrayHandleCartesianProduct, 59
ArrayHandleCast, 56
ArrayHandleCast.h, 56
ArrayHandleCompositeVector, 60, 73
ArrayHandleCompositeVector.h, 61
ArrayHandleCompositeVectorType, 60
ArrayHandleConstant, 55
ArrayHandleConstant.h, 55
ArrayHandleCounting, 55, 63
ArrayHandleCounting.h, 56
ArrayHandleGroupVec, 62
ArrayHandleGroupVec.h, 62
ArrayHandleImplicit, 63
ArrayHandleImplicit.h, 63
ArrayHandleIndex, 55
ArrayHandlePermutation, 57
ArrayHandlePermutation.h, 57
ArrayHandleTransform, 64, 65
ArrayHandleUniformPointCoordinates, 59
ArrayHandleZip, 58
ArrayHandleZip.h, 58
ArrayManagerExecution, xiv, 43
ArrayManagerExecutionShareWithControl, 44
ArrayPortalFromIterators, 31
ArrayPortalToIteratorBegin, 32
ArrayPortalToIteratorEnd, 32
ArrayPortalToIterators, 32
ArrayPortalToIterators.h, 32
ArrayTransfer, xv, 69, 70
array handle, 29–36, 53–78
    adapting, 73–78
    allocate, 33
    Cartesian product, 59–60
    cast, 56–57
    composite vector arrays, 60–62
    constant, 55
    counting, 55–56
    derived, 66–73
    dynamic, 79–84
    execution environment, 34–36
    fancy, 54–73
    group vector, 62
    implicit, 62–64
    index, 55
    permutation, 57–58
    populate, 34
    portal, 31–33
    rectilinear point coordinates, 59–60
    storage, 53–78
        default, 54, 77
    subclassing, 63, 72, 76
    transform, 64–66
    uniform point coordinates, 59
    zip, 58–59
array manager execution, 43–45
array portal, 31–33
array transfer, 69–72
ASin, 131
ASinH, 131
aspect, 149–150
    default, 149
    work index, 150
AspectTagDefault, 149

DRAFT

AspectTagWorkIndex, 150
assert, 19, 128
Assert.h, 19
ATan, 131
ATan2, 131
ATanH, 131

Cartesian product array handle, 59–60
cast array handle, 56–57
Cbrt, 132
Ceil, 132
ceiling, 132
cell
  derivative, 143–144
  gradient, 143–144
  interpolation, 143
  parametric coordinates, 142–143
  world coordinates, 142–143
CELL_SHAPE_EMPTY, 139
CELL_SHAPE_HEXAHEDRON, 140
CELL_SHAPE_LINE, 140
CELL_SHAPE_POLYGON, 140
CELL_SHAPE_PYRAMID, 140
CELL_SHAPE_QUAD, 140
CELL_SHAPE_TETRA, 140
CELL_SHAPE_TRIANGLE, 140
CELL_SHAPE_VERTEX, 140
CELL_SHAPE_WEDGE, 140
CellCount, 115
CellDerivative, 143
CellDerivative.h, 143
CellIndices, 116
CellInterpolate, 143
CellInterpolate.h, 143
CellSet, 91, 93, 123
CellSetExplicit, 92
CellSetListTag.h, 94
CellSetPermutation, 93
CellSetSingleType, 92
CellSetStructured, 91
CellShape, 113, 120
CellShape.h, 139
CellShapeIdToTag, 139
CellShapeTagEmpty, 139
CellShapeTagGeneric, 139
CellShapeTagHexahedron, 140
CellShapeTagLine, 140
CellShapeTagPolygon, 140
CellShapeTagPyramid, 140
CellShapeTagQuad, 140
CellShapeTagTetra, 140
CellShapeTagTriangle, 140
CellShapeTagVertex, 140
CellShapeTagWedge, 140
CellTopologicalDimensionsTag, 141
CellTraits, 141
CellTraits.h, 141
CellTraitsTagSizeFixed, 141
CellTraitsTagSizeVariable, 141
cell set, 85, 90–94
  dynamic, 93–94
  explicit, 92
  permutation, 93
  shape, 91
  single type, 92
  structured, 91–92
cell shape, 91, 139–142
cell to point map worklet, 114–118
cell traits, 141–142
CMake configuration
  VTKM_FORCE_ANSI, 165
  VTKM_USE_64BIT_IDS, 20
  VTKM_USE_DOUBLE_PRECISION, 20
column, 135
CommonTypes, 106
ComponentType, 23
composite vector arrays array handle, 60–62
ConfigureFor32.h, 20
ConfigureFor64.h, 20
constant array handle, 55
constant export, 18
cont namespace, 16, 17
control signature, ix, xvi, 104–109, 111, 113–115, 118, 120, 123, 147, 150, 159
control environment, 16
control signature, 105–106
  execution object, 123–124
  type list tags, 106
  whole array, 120–123
CoordinateSystem, 94
coordinate system, 85, 94–95
copy, 40
CopySign, 132
Cos, 132
CosH, 132
cosine, 132
counting array handle, 55–56
Cross, 134
cross product, 134
cube root, 132
CUDA, 18, 37, 39
cuda namespace, 17

DataSet, 7–9, 85, 86, 93, 94
DataSetBuilderExplicit, 87
DataSetBuilderExplicitIterative, 88
DataSetBuilderRectilinear, 86
DataSetBuilderUniform, 86
DataSetFieldAdd, 89
data set, 85–95
  Building, 85–90
  cell set, *see* cell set

  - coordinate system, *see* coordinate system
  - field, *see* field
- data set filter, 9
- derivative, 143–144
- derived storage, 66–73
- detail namespace, 17
- determinant, 135
- DeviceAdapter.h, 37
- DeviceAdapterAlgorithm, xiv, 35, 40, 45
- DeviceAdapterAlgorithmGeneral, 45
- DeviceAdapterCuda.h, 39
- DeviceAdapterOpenMP.h, 39
- DeviceAdapterSerial.h, 39
- DeviceAdapterTag.h, 43
- DeviceAdapterTagCuda, 39
- DeviceAdapterTagOpenMP, 39
- DeviceAdapterTagSerial, 39
- DeviceAdapterTagTBB, 39
- DeviceAdapterTBB.h, 39
- DeviceAdapterTimerImplementation, 49
- device adapter, 37–50
  - algorithm, 40–42, 45–49
  - array manager, 43–45
  - tag, 43
  - timer, 49–50
- device adapter tag, 37–40
  - provided, 39
- DimensionalityTag, 22
- dispatcher, 104
- DispatcherMapField, 104, 108
- DispatcherMapTopology, 104, 111, 114
- dot, 21
- DynamicArrayHandle, 27, 79, 157
- DynamicArrayHandleBase, 83
- DynamicCellSet, 93
- DynamicPointCoordinates, 157
- DynamicTransform, 157
- dynamic array handle, 79–84
  - cast, 81–84
  - construct, 79
  - new instance, 80
  - query, 79, 80
- dynamic cell set, 93–94

- environment, 15, 16
  - control, 16
  - execution, 15, 16
- Epsilon, 132
- Error, 18
- ErrorControlBadAllocation, 19
- ErrorControlBadType, 19
- ErrorControlBadValue, 19, 70, 81
- ErrorControlInternal, 19, 70
- ErrorExecution, 19, 41, 45, 128
- ErrorIO, 19
- ErrorMessageBuffer, 45
- errors, 18–19, 127–128
  - assert, 19, 128
  - control environment, 18–19
  - execution environment, 19, 41, 127–128
  - worklet, 127–128
- exec namespace, 16, 17
- ExecObject, xvi, 109, 110, 112, 115, 119, 123
- execution
  - control, 16
- execution signature, ix, xvi, 104, 107–109, 113, 115, 119, 125, 147, 159
- ExecutionObjectBase, 109, 112, 115, 119, 123, 148
- ExecutionTypes, 33
- execution array manager, 43–45
- execution environment, 15, 16
- execution object, 123–124
- execution signature, 107
- Exp, 132
- Exp10, 132
- Exp2, 132
- ExplicitCellSet, 92
- explicit cell set, 92
  - single type, 92
- explicit mesh, 87
- ExpM1, 132
- exponential, 132
- export
  - constant, 18
  - control, 17, 18, 108, 123, 166
  - execution, 17, 18, 108, 123, 166

- fancy array handle, 54–73
- Fetch, 149
- fetch, 149–150
  - aspect, *see* aspect
  - direct input array, 150
  - direct output array, 150
  - execution object, 150
- FetchTagArrayDirectIn, 150
- FetchTagArrayDirectOut, 150
- FetchTagExecObject, 150
- Field, 94
- field, 9, 85, 94
- FieldCommon, 106
- FieldIn, 107, 108
- FieldInCell, 112, 114
- FieldInFrom, 118
- FieldInOut, 108, 109, 112, 115, 119
- FieldInOutCell, 112
- FieldInOutPoint, 115
- FieldInPoint, 111, 114
- FieldInTo, 119
- FieldOut, 108, 112, 115, 119
- FieldOutCell, 112
- FieldOutPoint, 115
- FieldPointIn, 105, 143

DRAFT

field filter, 9
field map worklet, 103, 108–111
file I/O, 7–8
    read, 7–8
    write, 8
filter, 9, 15
    data set, 9
    field, 9
filter namespace, 17
Float32, xiii, 20, 22, 26, 106, 133, 166
Float64, 20, 26, 106, 133, 166
FloatDefault, 20, 142, 166
Floor, 132
floor, 132
FMod, 132
FromCount, 120
FromIndices, 120
function export, 17, 18, 108, 123, 166
functional array, 62–64
FunctionInterface, xvi, 150
FunctionInterfaceReturnContainer, 153
function interface, 150–159
    append parameter, 154
    dynamic transform, 156–158
    for each, 158–159
    invoke, 152–153
    replace parameter, 154–155
    static transform, 155–156
function signature, 150
functor, 15, 63
FunctorBase, 41, 128

GetComponent, 23
GetPortalConstControl, 33
GetPortalControl, 33, 34
gradient, 143–144
group vector array handle, 62

h, 60
HasMultipleComponents, 23
hexahedron, 140
hyperbolic arccossine, 131
hyperbolic arcsine, 131
hyperbolic cosine, 132
hyperbolic sine, 134
hyperbolic tangent, 131, 134

I/O, 7–8
Id, 20, 23, 26, 41, 42, 55, 63, 70, 106, 109, 113, 116, 120, 166
Id2, 20, 26, 106
Id2Type, 106
Id3, xiii, 20, 23, 26, 41, 59, 106
Id3Type, 106
IdComponent, 20, 109, 113, 115, 116, 120, 125, 139, 141, 149, 155, 166
identity matrix, 135
IdType, 106
image, 85
implicit array handle, 62–64
implicit storage, 62–64
Index, 106
IndexTag, 151, 155, 156
index array handle, 55
Infinity, 132
input domain, 107
input domain, xvi, 104, 107–109, 111, 114, 118
Int16, 20
Int32, 20
Int64, 20
Int8, 20
Intel Threading Building Blocks, 38, 39
internal namespace, 17, 150
interoperability, 17
interpolation, 143
inverse cosine, 131
inverse hyperbolic cosine, 131
inverse hyperbolic sine, 131
inverse hyperbolic tangent, 131
inverse matrix, 136
inverse sine, 131
inverse tangent, 131
invoke, 104
io namespace, 7, 17, 85
IsFinite, 132
IsInf, 132
IsNan, 132
IsNegative, 132

kernel, 15

Lerp, 134
less, 24
line, 140
linear interpolation, 134
linear system, 136
ListForEach, 27
lists, 25–28
    storage, 82
    types, 26–27
ListTag.h, 25, 27
ListTagBase, 25
ListTagEmpty, 25
ListTagJoin, 25
Log, 133
Log10, 133
Log1P, 133
Log2, 133
logarithm, 133
lower bounds, 40

Magnitude, 134

DRAFT

MagnitudeSquared, 134
make_ArrayHandle, 30
make_ArrayHandleCartesianProduct, 60
make_ArrayHandleCast, 56
make_ArrayHandleCompositeVector, 61
make_ArrayHandleConstant, 55
make_ArrayHandleCounting, 56
make_ArrayHandleGroupVec, 62
make_ArrayHandleImplicit, 63
make_ArrayHandlePermutation, 57
make_ArrayHandleTransform, 65
make_ArrayHandleZip, 58
make_FunctionInterface, xvi, 151
make_Pair, 21
make_Vec, 20
map, 103
map cell to point, 114–118
map field, 108–111
map point to cell, 111–114
map topology, 118–120
math, 131–137
Math.h, 131
Matrix, 135, 136, 166
matrix, 135–136
Matrix.h, 135
MatrixDeterminant, 135
MatrixGetColumn, 135
MatrixGetRow, 135
MatrixIdentity, 135
MatrixInverse, 136
MatrixMultiply, 136
MatrixRow, 135
MatrixSetColumn, 136
MatrixSetRow, 136
MatrixTranspose, 136
Max, 133
maximum, 133
metaprogramming, 25
method export, 17, 18, 108, 123, 166
Min, 133
minimum, 133
ModF, 133

namespace, 16
    detail, 17
    internal, 17
    vtkm, 16, 17, 131, 139, 150
    vtkm::cont, 16, 17
    vtkm::cont::arg, 147, 148
    vtkm::cont::cuda, 17
    vtkm::cont::tbb, 17
    vtkm::exec, 16, 17
    vtkm::exec::arg, 149
    vtkm::filter, 17
    vtkm::internal, 150
    vtkm::io, 7, 17, 85
    vtkm::io::reader, 7
    vtkm::io::writer, 8
    vtkm::opengl, 17
    vtkm::rendering, 17
    vtkm::worklet, 17
Nan, 133
natural logarithm, 133
NDEBUG, 19
negative, 132
NegativeInfinity, 133
Newton's method, 136–137
NewtonsMethod, 136
NewtonsMethod.h, 136
Normal, 135
Normalize, 135
not a number, 133
NUM_COMPONENTS, 20, 23
NumericTag, 22

OpenGL, 17
opengl namespace, 17
OpenMP, 38, 39

packages,
See Also:
namespace16, 16–17
Pair, 21, 58, 60
ParametricCoordinates.h, 142
ParametricCoordinatesCenter, 142
ParametricCoordinatesPoint, 142
ParametricCoordinatesToWorldCoordinates, 143
parametric coordinates, 142–143
permutation cell set, 93
permuted array handle, 57–58
pervasive parallelism, 15
Pi, 133
Pi_2, 133
Pi_3, 133
Pi_4, 133
PointCount, 113
PointIndices, 113
point to cell map worklet, 111–114
point to cell worklet, 103
polygon, 140
PortalConstControl, 33
PortalControl, 33
Pow, 133
power, 133
PrepareForInPlace, 34
PrepareForInput, 34
PrepareForOutput, 34, 39
pyramid, 140

quadrilateral, 140

RaiseError, 41

DRAFT

- RCbrt, 133
- reader namespace, 7
- read file, 7–8
- reciprocal cube root, 133
- reciprocal square root, 134
- rectilinear grid, 86
- rectilinear point coordinates array handle, 59–60
- reduce, 41
- reduce by key, 41
- regular grid, 85
- Remainder, 133
- remainder, 132, 133
- RemainderQuotient, 134
- rendering namespace, 17
- RMagnitude, 135
- Round, 134
- round down, *see* floor
- round up, *see* ceiling
- row, 135
- RSqrt, 134

- Scalar, 106
- ScalarAll, 106
- scan
  - exclusive, 41
  - inclusive, 41
- scatter, 124–127
- scatter type, 125
- ScatterCounting, 125, 126
- ScatterIdentity, 125
- ScatterUniform, 125
- schedule, 41
- serial, 38, 39
- SetComponent, 23
- shape, 91, 139–142
- signature, 150
  - control, ix, xvi, 104–109, 111, 113–115, 118, 120, 123, 147, 150, 159
  - execution, ix, xvi, 104, 107–109, 113, 115, 119, 125, 147, 159
- signature tags, 105
  - _1, 107, 109, 113, 115, 120
  - _2, 107, 109, 113, 115, 120
  - AllTypes, 106
  - CellCount, 115
  - CellIndices, 116
  - CellShape, 113, 120
  - CommonTypes, 106
  - ExecObject, xvi, 109, 110, 112, 115, 119, 123
  - FieldCommon, 106
  - FieldIn, 107, 108
  - FieldInCell, 112, 114
  - FieldInFrom, 118
  - FieldInOut, 108, 109, 112, 115, 119
  - FieldInOutCell, 112
  - FieldInOutPoint, 115
  - FieldInPoint, 111, 114
  - FieldInTo, 119
  - FieldOut, 108, 112, 115, 119
  - FieldOutCell, 112
  - FieldOutPoint, 115
  - FieldPointIn, 105, 143
  - FromCount, 120
  - FromIndices, 120
  - Id2Type, 106
  - Id3Type, 106
  - IdType, 106
  - Index, 106
  - PointCount, 113
  - PointIndices, 113
  - Scalar, 106
  - ScalarAll, 106
  - TopologyIn, 107, 111, 114, 118
  - Vec2, 106
  - Vec3, 106
  - Vec4, 106
  - VecAll, 106
  - VecCommon, 106
  - VisitIndex, 109, 113, 116, 120, 125
  - WholeArrayIn, xvi, 109, 112, 115, 119, 120
  - WholeArrayInOut, 109, 112, 115, 119
  - WholeArrayOut, 109, 110, 112, 115, 119
  - WorkIndex, 107, 109, 110, 113, 116, 120, 150
- SignBit, 134
- Sin, 134
- sine, 134
- single type cell set, 92
- SinH, 134
- SolveLinearSystem, 136
- sort, 41
  - by key, 41
- Sqrt, 134
- square root, 134
- Storage, xv, 43, 67, 70, 75
- storage, 53–78
  - adapting, 73–78
  - default, 54, 77
  - derived, 66–73
  - implicit, 62–64
- storage lists, 82
- StorageBasic.h, 54
- StorageListTag.h, 82
- StorageListTagBasic, 82
- StorageTagBasic, 54
- stream compact, 41
- structured cell set, 91–92
- synchronize, 42

- tag, 21
  - cell shape, 139–140
  - device adapter, 37–40
    - provided, 39

DRAFT

- dimensionality, 22
- lists, 25–28
- multiple components, 23
- numeric, 22
- shape, 139–140
- single component, 23
- storage lists, 82
- topology element, 118
- type lists, 26–27
- type traits, 22–23
- vector traits, 23–24

Tan, 134
tangent, 134
TanH, 134
TBB, 38, 39
tbb namespace, 17
template metaprogramming, 25
tetrahedron, 140
Timer, xiv, 49, 51
timer, 49–52
TopologyElementTag.h, 118
TopologyElementTagCell, 118
TopologyElementTagEdge, 118
TopologyElementTagFace, 118
TopologyElementTagPoint, 118
TopologyIn, 107, 111, 114, 118
topology element tag, 118
topology map worklet, 103, 118–120
ToVec, 23
traits, 21–25
transformed array, 64–66
Transport, xvi, 148, 149
transport, 148–149
- execution object, 149
- input array, 148
- output array, 148

TransportTagArrayIn, 148
TransportTagArrayOut, 148
TransportTagExecObject, 149
transpose matrix, 136
triangle, 140
TriangleNormal, 135
TwoPi, 134
type lists, 26–27
TypeCheck, xvi, 147, 148
TypeCheckTagArray, 148
TypeCheckTagExecObject, 148
TypeListTag.h, 26, 27, 82
TypeListTagAll, 27
TypeListTagCommon, 27, 106
TypeListTagField, 26, 106
TypeListTagFieldScalar, 26, 106
TypeListTagFieldVec2, 26, 106
TypeListTagFieldVec3, 26, 106
TypeListTagFieldVec4, 26, 106
TypeListTagId, 26, 106
TypeListTagId2, 26, 106
TypeListTagId3, 26, 106
TypeListTagIndex, 26, 106
TypeListTagScalarAll, 26, 106
TypeListTagVecAll, 27, 106
TypeListTagVecCommon, 26, 106
Types.h, 17, 19, 27, 164
TypeTraits, xiii, 22
TypeTraitsIntegerTag, 22
TypeTraitsRealTag, 22
TypeTraitsScalarTag, 22
TypeTraitsVectorTag, 22
type check, 147–148
- array, 147
- execution object, 148

type list tags, 106

UInt16, 20
UInt32, 20
UInt64, 20
UInt8, 20
uniform grid, 85
uniform point coordinates array handle, 59
unique, 42
unstructured grid, 87
upper bounds, 42

Vec, xiii, xiv, 20, 21, 23, 26, 27, 56, 60, 62, 79, 106, 134–136, 142, 143, 164, 166
Vec2, 106
Vec3, 106
Vec4, 106
VecAll, 106
VecCommon, 106
VectorAnalysis.h, 134
vector analysis, 134–135
VecTraits, xiii, 23
VecTraitsTagMultipleComponents, 23
VecTraitsTagSingleComponent, 23
VecVariable, 166
vertex, 140
VisitIndex, 109, 113, 116, 120, 125
visit index, 125
VTKDataSetReader, 7
VTKDataSetWriter, 8
vtkm namespace, 16, 17, 131, 139, 150
vtkm/cont/ArrayHandleCartesianProduct.h/h, 60
vtkm/cont/ArrayHandleCompositeVector.h/h, 60
vtkm/cont/cuda/DeviceAdapterCuda.h, 39
vtkm/cont/internal/DeviceAdapterTag.h, 43
vtkm/cont/tbb/DeviceAdapterTBB.h, 39
vtkm/cont/ArrayHandle.h, 17, 163
vtkm/cont/ArrayHandleCast.h, 56
vtkm/cont/ArrayHandleCompositeVector.h, 61
vtkm/cont/ArrayHandleConstant.h, 55

DRAFT

vtkm/cont/ArrayHandleCounting.h, 56
vtkm/cont/ArrayHandleGroupVec.h, 62
vtkm/cont/ArrayHandleImplicit.h, 63
vtkm/cont/ArrayHandlePermutation.h, 57
vtkm/cont/ArrayHandleZip.h, 58
vtkm/cont/ArrayPortalToIterators.h, 32
vtkm/cont/CellSetListTag.h, 94
vtkm/cont/DeviceAdapter.h, 37
vtkm/cont/DeviceAdapterSerial.h, 39
vtkm/cont/StorageBasic.h, 54
vtkm/cont/StorageListTag.h, 82
vtkm/exec/CellDerivative.h, 143
vtkm/exec/CellInterpolate.h, 143
vtkm/exec/ParametricCoordinates.h, 142
vtkm/internal/ConfigureFor32.h, 20
vtkm/internal/ConfigureFor64.h, 20
vtkm/openmp/cont/DeviceAdapterOpenMP.h, 39
vtkm/worklet/WorkletMapTopology.h, 111
vtkm::cont, 16, 17
vtkm::cont::arg, 147, 148
vtkm::cont::cuda, 17
vtkm::cont::tbb, 17
vtkm::exec, 16, 17
vtkm::exec::arg, 149
vtkm::filter, 17
vtkm::internal, 150
vtkm::io, 7, 17, 85
vtkm::io::reader, 7
vtkm::io::writer, 8
vtkm::opengl, 17
vtkm::rendering, 17
vtkm::worklet, 17
VTKM_ARRAY_HANDLE_SUBCLASS, 64, 66, 73, 77
VTKM_ARRAY_HANDLE_SUBCLASS_NT, 64, 66, 73, 77
VTKM_ASSERT, xiii, 19, 128
VTKM_CONT_EXPORT, 17, 18, 166
VTKM_DEFAULT_CELL_SET_LIST_TAG, 94
VTKM_DEFAULT_DEVICE_ADAPTER_TAG, 39
VTKM_DEFAULT_STORAGE_LIST_TAG, 82
VTKM_DEFAULT_STORAGE_TAG, 54, 78
VTKM_DEFAULT_TYPE_LIST_TAG, 27, 82
VTKM_DEVICE_ADAPTER, 38, 39
VTKM_DEVICE_ADAPTER_CUDA, 38
VTKM_DEVICE_ADAPTER_ERROR, 38, 39
VTKM_DEVICE_ADAPTER_OPENMP, 38
VTKM_DEVICE_ADAPTER_SERIAL, 38
VTKM_DEVICE_ADAPTER_TBB, 38
VTKM_EXEC_CONSTANT_EXPORT, 18
VTKM_EXEC_CONT_EXPORT, 17, 18, 108, 123, 166
VTKM_EXEC_EXPORT, 17, 18, 108, 123, 166
VTKM_FORCE_ANSI, 165
VTKM_IS_CELL_SHAPE_TAG, 139
VTKM_IS_DEVICE_ADAPTER_TAG, 40
VTKM_MAX_BASE_LIST, 25
VTKM_NO_64BIT_IDS, 20
VTKM_NO_DOUBLE_PRECISION, 20
VTKM_STORAGE, 54, 78
VTKM_STORAGE_BASIC, 54
VTKM_STORAGE_UNDEFINED, 78
VTKM_SUPPRESS_EXEC_WARNINGS, 18
VTKM_USE_64BIT_IDS, 20
VTKM_USE_DOUBLE_PRECISION, 20
VTKM_VALID_DEVICE_ADAPTER, 43
vtkm/Assert.h, 19
vtkm/CellShape.h, 139
vtkm/CellTraits.h, 141
vtkm/ListTag.h, 25, 27
vtkm/Math.h, 131
vtkm/Matrix.h, 135
vtkm/NewtonsMethod.h, 136
vtkm/TopologyElementTag.h, 118
vtkm/TypeListTag.h, 26, 27, 82
vtkm/Types.h, 17, 19, 27, 164
vtkm/VectorAnalysis.h, 134
vtkmGenericCellShapeMacro, 140

wedge, 140
WholeArrayIn, xvi, 109, 112, 115, 119, 120
WholeArrayInOut, 109, 112, 115, 119
WholeArrayOut, 109, 110, 112, 115, 119
whole array, 120–123
WorkIndex, 107, 109, 110, 113, 116, 120, 150
worklet, 15, 103–128
- control signature, 105–106
- creating, 104–128
- error handling, 127–128
- execution object, 123–124
- execution signature, 107
- input domain, 107
- scatter, 124–127
- whole array, 120–123

worklet namespace, 17
WorkletMapCellToPoint, 114
WorkletMapField, 103, 104, 108, 124
WorkletMapPointToCell, 103, 104, 111, 118, 124
WorkletMapTopology, 103, 104, 118
WorkletMapTopology.h, 111
worklet types, 103, 108–120
- cell to point map, 114–118
- field map, 103, 108–111
- point to cell, 103
- point to cell map, 111–114
- topology map, 103, 118–120

WorldCoordinatesToParametricCoordinates, 143
world coordinates, 142–143
writer namespace, 8
write file, 8

zipped array handles, 58–59

DRAFT