Overview of VTK

Lecture #11  Hank Childs, University of Oregon
Announcements

• Quiz #2: Tues Feb 19th. On isolines

• 510 presentations: Tuesday March 10\textsuperscript{th}
  – 2x 5 minute presentations
  – Roll will be taken, 2.5 point penalty for missing
CIS 410/510: Project #6
Due Feb 16, 2020
(which means submitted by 6am on Feb 17, 2020)
Worth 8 points

Assignment:
Implement “marching tetrahedrons.” It is similar to the “marching quads” of project 5, but it will require that you extend this algorithm to three dimensions, specifically for tetrahedrons. (Tetrahedrons are the simplest cell type – 4 vertices.)

You will need to make your own conventions for edge numbering, build your own lookup tables, etc.

The skeleton code I provide:
- Has a Tetrahedron class
- Has an empty implementation of “IsosurfaceTet(...)
- Has the code that loads data and calls IsosurfaceTet

Note this prompt is sparser than previous ones. I figure you all are getting the hang of this by now.

What to upload?:
- Your proj6.cxx file with a working IsosurfaceTet function.
- A screenshot of you running your code and getting the correct output
Final Project: Decision Tree

- This is volume rendering
410 Decision Tree

• Have a custom vis project you want to do?
  – Upload a proposal to Canvas
  – Types of custom project:
    • Movie
      – If you think it sounds fun to make a movie, then pay close attention next two weeks
    • Algorithm, etc.

• If not:
  – Or do volume rendering
510 Decision Tree

• Have a custom vis project you want to do?
  – Upload a proposal to Canvas

• If not:
  – If you think it sounds fun to make a movie, then pay close attention next two weeks

• **AND** do volume rendering
When are proposals due?

• Sunday Feb 23\textsuperscript{rd}

• By that time you will have seen everything about making movies

• You will not have seen about volume rendering (lectures on Feb 25\textsuperscript{th} & 27\textsuperscript{th})

• If you see volume rendering lectures and want to switch to a custom project, then that is fine
Outline

• Quick introduction to VTK
• Foundational concepts
  – Object-oriented programming
  – Data flow networks
• Overview of key VTK modules
Outline

• Quick introduction to VTK
• Foundational concepts
  – Object-oriented programming
  – Data flow networks
• Overview of key VTK modules
The Visualization ToolKit

• Open Source library for visualization and graphics
• Written in C++
  – Bindings in Tcl, Java, Python
• Object-oriented
  – Few base types
  – Many, many derived types that provide functionality
• Strengths:
  – Very powerful
  – Designed for extensibility
• Weaknesses:
  – Extensibility design choice comes with performance tradeoffs
VTK: History

• Initially developed in 1993 by three General Electrics researchers
  – Schroeder, Martin, Lorensen

• Kitware Inc. started in 1998 to support VTK
  – world-wide community then begins participating in development

• Usage:
  – basis for many end-user visualization tools
    • VisIt, ParaView, MayaVi, and more ...
  – many, many custom tools
VTK: a mature infrastructure

Textbook based on VTK, used for many college courses

Guide devoted to effective usage of VTK software

Nightly regression tests

Online documentation
Outline

• Quick introduction to VTK
• Foundational concepts
  – Object-oriented programming
  – Data flow networks
• Overview of key VTK modules
Object-oriented programming (OOP)

• Programming paradigm with “objects”
• Objects contain:
  – data fields
  – methods
• OOP computer programs consist of objects interacting with each other
• C++: class defines the form of an object, and an instance of that class is the object

Hank’s opinion: object-oriented programming is a big deal for VTK because of subtyping and polymorphism
Subtyping

- Subtyping: concept from programming language theory
  - Wikipedia: “a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype”
- “Is a” test
  - If S is a sub-type of T, then S “is a” T
    - Example: T = fruit, S = apple
- Abstract types: define an interface, but no implementation
  This is implemented in C++ using classes, inheritance, and virtual functions.
Polymorphism (generic programming)

- Write code using abstract type
  - Abstract type has methods
  - No usage of concrete types

- Allows for extensibility
  - Can add many new concrete types afterwards
Polymorphism Example

• Abstract type: Shape
• Concrete types: Triangle, Square, Hexagon
• Methods:
  – Shape::GetArea()
  – Shape::GetNumberOfEdges()
• Key points:
  – programs can be written to Shape interface, with no need for knowledge of derived types
  – new concrete types of Shape (e.g., Octagon) can be added afterwards, and not affect existing code

VTK was successful in choosing abstract types that allowed for great extensibility
Also: if you learn abstract types, then you know VTK
Subtyping and Polymorphism

• Subtyping makes VTK developers more efficient – utilize base classes
  – I can write a new subtype and focus on just the parts that make that subtype special

• Polymorphism makes VTK users more efficient – library changes, but user code stays the same
  – I can write code about “Shapes” and don’t have to worry if new shapes are added
Outline

• Quick introduction to VTK
• Foundational concepts
  – Object-oriented programming
  – Data flow networks
• Overview of key VTK modules
Data Flow Overview

• VTK employs the data flow network paradigm

• Basic idea:
  – You have many modules to choose from
    • Hundreds!!
  – You compose some of these modules together to perform some desired functionality

• Advantages:
  – Customizability
  – Design fosters interoperability between modules to the extent possible
Data Flow Overview

- **Participants:**
  - Source: a module that produces data
    - It creates an output
  - Sink: a module that consumes data
    - It operates on an input
  - Filter: a module that transforms input data to create output data

- **Nominal inheritance hierarchy:**
  - A filter “is a” source
  - A filter “is a” sink
Example of data flow (image processing)

• Sources:
  – FileReader: reader from file
  – Color: generate image with one color

• Filters:
  – Crop: crop image, leaving only a sub-portion
  – Transpose: view image as a 2D matrix and transpose it
  – Invert: invert colors
  – Concatenate: paste two images together

• Sinks:
  – FileWriter: write to file
Example of data flow (image processing)
Example of data flow (image processing)

- **Participants:**
  - Source: a module that produces data
    - It creates an output
  - Sink: a module that consumes data
    - It operates on an input
  - Filter: a module that transforms input data to create output data

- **Pipeline**: a collection of sources, filters, and sinks connected together
Benefits of the Data Flow Design

• Extensible!
  – write infrastructure that knows about abstract types (source, sink, filter, and data object)
  – write as many derived types as you want

• Composable!
  – combine filters, sources and sinks in custom configurations

What do you think the benefits are?
Drawbacks of Data Flow Design

• Operations happen in stages
  – Extra memory needed for intermediate results
  – Not cache efficient

• Compartmentalization can limit possible optimizations

What do you think the drawbacks are?
Outline

• Quick introduction to VTK
• Foundational concepts
  – Object-oriented programming
  – Data flow networks
• Overview of key VTK modules
Key abstract types in VTK

- `vtkDataObject / vtkDataSet`
- `vtkAlgorithm`
- Graphics modules
Key abstract types in VTK

- vtkDataObject / vtkDataSet
- vtkAlgorithm
- Graphics modules
We are now talking about the data objects that flow between modules ... not just images.
While vtkDataObject allows VTK developers to add custom abstractions, almost all usage by new users of VTK is via vtkDataSet.
Key abstract type in VTK: vtkDataObject

I’ve gone 20 years using almost exclusively four concrete types of vtkDataObject.

Important derived types of vtkDataSet

- vtkStructuredGrid
- vtkUnstructuredGrid
- vtkRectilinearGrid
- vtkPolyData
Important methods associated with vtkDataSet

• int GetNumberOfCells();
• int GetNumberOfPoints();
• vtkCell *GetCell(int cellID);
• double *GetPoint(int pointID);
• vtkPointData *GetPointData();
  – Gets fields defined on points (vertices) of mesh
• vtkCellData *GetCellData();
  – Gets fields defined on cells (elements) of mesh
• vtkFieldData *GetFieldData();
  – Gets fields defined not on cells or points

Fields are flexible in VTK, including scalars, vectors, tensors, and fields of arbitrary length.

Polymorphism! ... each derived type implements this interface.

But using this general interface can cost performance. Fixes?
vtkRectilinearGrid

- Points are implicit
- Cell connectivity is implicit
- Grid of \( Ni \times Nj \times Nk \) takes \( (Ni + Nj + Nk) \) storage
vtkStructuredGrid

- Also called “curvilinear mesh”
- Points are explicit
- Cell connectivity is implicit
- Grid of Ni x Nj x Nk takes $3 \times (Ni \times Nj \times Nk)$ storage
vtkUnstructuredGrid

- Many supported cell types
  - Tetrahedron, hexahedron, wedge, pyramid, triangle, quadrilateral, higher order, more...
- Points are explicit
- Cells are explicit
- Grid of N points and M cells, with K points per cell takes $3N + MK$ storage
vtkPolyData

• Identical to vtkUnstructuredGrid, but cell types are limited to polygonal data
• This is useful for graphics purposes, when rendering surfaces
Key abstract types in VTK

- vtkDataObject / vtkDataSet
- vtkAlgorithm
- Graphics modules
Key abstract type in vtk: vtkAlgorithm

• While data flow has clear concepts for “Source”, “Sink”, and “Filter”, VTK has a single class “vtkAlgorithm”
  – Previously had differentiated types

• vtkAlgorithm:
  – has zero, one, or more inputs
    • void SetInputConnection(vtkAlgorithmOutput *); // port 0
    • void SetInputData(int port, vtkAlgorithmOutput *);
  – has zero, one, or more outputs
    • vtkAlgorithmOutput *GetOutputPort(void); // port 0
    • vtkAlgorithmOutput *GetOutput(int);
First program

```c
#include <vtkDataSetReader.h>
#include <vtkContourFilter.h>
#include <vtkDataSetWriter.h>

int main()
{
    vtkDataSetReader *rdr = vtkDataSetReader::New();
    rdr->SetFileName("noise.vtk");

    // Contour the data.
    vtkContourFilter *cf = vtkContourFilter::New();
    cf->SetNumberOfContours(1);
    cf->SetValue(0, 3.0);
    cf->SetInputConnection(rdr->GetOutputPort());

    vtkDataSetWriter *wrtr = vtkDataSetWriter::New();
    wrtr->SetFileName("contour.vtk");
    wrtr->SetInputConnection(cf->GetOutputPort());
    wrtr->Write();
}
```

```
fawcett:VTK_ex childs$ cat contour_no_graphics.C
#include <vtkDataSetReader.h>
#include <vtkContourFilter.h>
#include <vtkDataSetWriter.h>

int main()
{
    vtkDataSetReader *rdr = vtkDataSetReader::New();
    rdr->SetFileName("noise.vtk");

    // Contour the data.
    vtkContourFilter *cf = vtkContourFilter::New();
    cf->SetNumberOfContours(1);
    cf->SetValue(0, 3.0);
    cf->SetInputConnection(rdr->GetOutputPort());

    vtkDataSetWriter *wrtr = vtkDataSetWriter::New();
    wrtr->SetFileName("contour.vtk");
    wrtr->SetInputConnection(cf->GetOutputPort());
    wrtr->Write();
}
```

```
fawcett:VTK_ex childs$ make
 g++ -g -o contour_no_graphics -I/Users/childs/visit/vtk/6.0.0/i386-apple-darwin10_gcc-4.2/include/vtk-6.0 contour_no_graphics.C -L/Users/childs/visit/vtk/6.0.0/i386-apple-darwin10_gcc-4.2/lib -lvtkRenderingFreeTypeOpenGL-6.0 -lvtkRenderingFreeType-6.0 -lvtkInteractionStyle-6.0 -lvtkRenderingOpenGL-6.0 -lvtkIOImage-6.0 -lvtkIOLegacy-6.0 -lvtkIOCore-6.0 -lvtkRenderingCore-6.0 -lvtkFiltersCore-6.0 -lvtkCommonDataModel-6.0 -lvtkCommonMisc-6.0 -lvtkCommonExecutionModel-6.0 -lvtkCommonCore-6.0
fawcett:VTK_ex childs$ ./contour_no_graphics
fawcett:VTK_ex childs$ ls -l contour.vtk
-rw-r--r--  1 childs  staff  1383911 Jul  6 15:47 contour.vtk
```
First program

Modules have many options for how they execute. These options are encoded as attributes in the module and modified using “Setter” functions.
VTK forces all VTK objects to be allocated using dynamic memory (the heap).
VTK memory management

• VTK uses reference counting for all objects (vtkAlgorithm, vtkDataObject, etc)

• Rules:
  – All new objects have a reference count of 1
  – Register() increments the reference count
  – Delete() deletes the reference count
  – When reference count hits 0, the object is deleted

• VTK shares arrays between vtkDataObjects, to save on memory...
  – ... which means they can’t store arrays on stack, since the arrays could go out of scope (dangling pointer)

VTK has recently introduced a templated type, vtkSmartPointer, to assist with reference counting.
First program (leak free version)

```c
#include <vtkDataSetReader.h>
#include <vtkContourFilter.h>
#include <vtkDataSetWriter.h>

int main()
{
    vtkDataSetReader *rdr = vtkDataSetReader::New();
    rdr->SetFileName("noise.vtk");

    // Contour the data.
    vtkContourFilter *cf = vtkContourFilter::New();
    cf->SetNumberOfContours(1);
    cf->SetValue(0, 3.0);
    cf->SetInputConnection(rdr->GetOutputPort());

    vtkDataSetWriter *wrtr = vtkDataSetWriter::New();
    wrtr->SetFileName("contour.vtk");
    wrtr->SetInputConnection(cf->GetOutputPort());
    wrtr->Write();

    rdr->Delete();
    cf->Delete();
    wrtr->Delete();
}
```
The pipeline is constructed via `SetInputConnection()` and `GetOutputPort()` calls.

How does VTK control execution?
VTK’s Execution Model

• Key method: Update()
  – Update() requests a module to get its output “up-to-date”, i.e., to calculate it

• But what if that modules inputs are not up-to-date?
  – Part of an Update() is to call Update() on all the inputs to a module

• In the example program, “Write()” knows to request its input is up-to-date, which propagates up the pipeline
First program

```c
#include <vtkDataSetReader.h>
#include <vtkContourFilter.h>
#include <vtkDataSetWriter.h>

int main()
{
    vtkDataSetReader *rdr = vtkDataSetReader::New();
    rdr->SetFileName("noise.vtk");

    // Contour the data.
    vtkContourFilter *cf = vtkContourFilter::New();
    cf->SetNumberOfContours(1);
    cf->SetValue(0, 3.0);
    cf->SetInputConnection(rdr->GetOutputPort());

    vtkDataSetWriter *wrtr = vtkDataSetWriter::New();
    wrtr->SetFileName("contour.vtk");
    wrtr->SetInputConnection(cf->GetOutputPort());
    wrtr->Write();
}
```

1) wrtr asks cf to Update()
2) cf asks rdr to Update()
3) rdr reads from the file
4) cf calculates contour
5) wrtr writes file
**VTK & Time Stamps**

- VTK prevents unnecessarily re-calculation of the pipeline
  - It uses time stamps to keep track of when a module or its input was modified, and when the last time was it calculated its outputed.
First program

```
#include <vtkDataSetReader.h>
#include <vtkContourFilter.h>
#include <vtkDataSetWriter.h>

int main()
{
    vtkDataSetReader *rdr = vtkDataSetReader::New();
    rdr->SetFileName("noise.vtk");

    // Contour the data.
    vtkContourFilter *cf = vtkContourFilter::New();
    cf->SetNumberOfContours(1);
    cf->SetValue(0, 3.0);
    cf->SetInputConnection(rdr->GetOutputPort());

    vtkDataSetWriter *wrtr = vtkDataSetWriter::New();
    wrtr->SetFileName("contour.vtk");
    wrtr->SetInputConnection(cf->GetOutputPort());
    wrtr->Write();

    cf->SetValue(0, 3.5);
    wrtr->SetFileName("contour2.vtk");
    wrtr->Write();

    rdr->Delete();
    cf->Delete();
    wrtr->Delete();
}
```

1) wrtr asks cf to Update()
2) cf asks rdr to Update()
3) rdr reads from the file
4) cf calculates contour
5) wrtr writes file
Topology of pipelines

• Each module can have multiple inputs, multiple outputs

• Multiple sinks are fine
  – Call Update() on each

• Cycles are technically OK, but can be problematic
Key abstract types in VTK

- vtkDataObject / vtkDataSet
- vtkAlgorithm
- Graphics modules
Graphics Modules

- 90+% of VTK source code is sources, sinks, and filters.
- <10% is graphics / windowing.
- ... but ~50% of most “getting started” programs involve graphics / windows
5 Abstractions for Graphics / Windowing

1. **RenderWindow**: a window
2. **Renderer**: the place inside a window where you can render
   – There can be multiple renderers within a window
3. **Actor**: something that can be placed into a renderer
4. **Mapper**: maps data to geometric primitives
   – One mapper can be associated with multiple actors
5. **RenderWindowInteractor**: defines what button clicks, mouse movements, etc. should do
Example with graphics / windowing (pt 1)

```cpp
int main()
{
  // The following lines create a sphere represented by polygons.
  //
  vtkSmartPointer<vtkSphereSource> sphere =
    vtkSmartPointer<vtkSphereSource>::New();
sphere->SetThetaResolution(100);
sphere->SetPhiResolution(50);

  // The mapper is responsible for pushing the geometry into the graphics
  // library. It may also do color mapping, if scalars or other attributes
  // are defined.
  //
  vtkSmartPointer<vtkPolyDataMapper> sphereMapper =
    vtkSmartPointer<vtkPolyDataMapper>::New();
sphereMapper->SetInputConnection(sphere->GetOutputPort());

  vtkSmartPointer<vtkActor> sphere1 =
    vtkSmartPointer<vtkActor>::New();
sphere1->SetMapper(sphereMapper);
sphere1->GetProperty()->SetColor(1,0,0);

  vtkSmartPointer<vtkActor> sphere2 =
    vtkSmartPointer<vtkActor>::New();
sphere2->SetMapper(sphereMapper);
sphere2->GetProperty()->SetColor(0,1,0);
sphere2->AddPosition(1.25,0,0);
```

Adapted from SpecularSpheres.cxx in VTK source code
Example with graphics / windowing (pt 2)

```cpp
// Create the graphics structure. The renderer renders into the
// render window. The render window interactor captures mouse events
// and will perform appropriate camera or actor manipulation
// depending on the nature of the events.
//
vtkSmartPointer<vtkRenderer> ren1 =
  vtkSmartPointer<vtkRenderer>::New();
vtkSmartPointer<vtkRenderWindow> renWin =
  vtkSmartPointer<vtkRenderWindow>::New();
renWin->AddRenderer(ren1);
vtkSmartPointer<vtkRenderWindowInteractor> iren =
  vtkSmartPointer<vtkRenderWindowInteractor>::New();
iren->SetRenderWindow(renWin);

// Add the actors to the renderer, set the background and size.
//
ren1->AddActor(sphere1);
ren1->AddActor(sphere2);
ren1->SetBackground(0.1, 0.2, 0.4);
renWin->SetSize(400, 200);

ren1->GetActiveCamera()->SetFocalPoint(0, 0, 0);
ren1->GetActiveCamera()->SetPosition(0, 0, 1);
ren1->GetActiveCamera()->SetViewUp(0, 1, 0);
ren1->GetActiveCamera()->ParallelProjectionOn();
ren1->ResetCamera();
ren1->GetActiveCamera()->SetParallelScale(1.5);

// This starts the event loop and invokes an initial render.
//
iren->Initialize();
iren->Start();

return EXIT_SUCCESS;
```
More Example Programs

• Many example programs in VTK download
• Some C++, some Python
• Challenge is typically figuring out how to map what you want to do to VTK modules
  – How to find the right module?
  – How to set up the module’s options?
  – Good reference for these questions:
    • http://www.vtk.org/doc/release/6.2/html/
Summary

• VTK is open source, written in C++, and is supported by a large community
• It employs the data flow paradigm
• It has many modules (readers, filters, mappers), which makes it very powerful
• It is well-suited to for many tasks including:
  – foundation for visualization tools
  – one-off visual explorations of data
  – custom visualization tools, especially when considering the effort to incorporate it