VTK Visualization Pipeline
Overview

- Actually, hard to “get hang of”/intuition without using/programming
  - So, we’ll do some more

- VTK Visualization Pipeline – context

- Data and process objects

- An example – 3d surface visualization

- Execution control

- Visualization Packages
VTK Visualization Pipeline

• Pipeline has two phases:
  – Visualization phase
    • Processes up to and including mapping of data to geometry
  – Graphics phase
    • Creating the scene

• Or, visualization (VTK) transforms data into pictures
Recall ... VTK Applications (big picture)

- 2 steps in creating graphics and visualization applications with VTK:
  1. Construct a data pipeline (i.e., visualization network) to process data
  2. Create the necessary graphics objects to display the data

- VTK architecture is based on a demand-driven, pipeline architecture (a visualization network)
  - Applications must first create a network, and then EXECUTE it
  - Typically requires a Render() (sent to the rendering window) or an Update() (sent to a filter in the pipeline).
    - Just instantiating the objects and hooking them together won't do anything – YOU HAVE TO REQUEST DATA to get data
Recall ... VTK Applications
(big picture)

• **Constructing a pipeline means:**
  – Connecting sources (create data), filters (process data), and mappers (map through lookup table and into graphics library)
  – Type checking controls which filters can be connected together

• To create the graphics objects, typically
  – create a rendering win to render into
  – create a renderer
  – create an interactor (allows you to interact with the graphics)
  – create one or more actors (each of which is linked to a mapper)
  – render

• May also wish to
  – transform objects;
  – set material properties; and/or
  – create lights, cameras, texture maps, and lookup tables, and various other graphics objects.
Sphere – VTK Visualization Pipeline …
(big picture, with code)

// geometry for sphere created
vtkSphereSource *sphere = vtkSphereSource::New();
sphere->SetRadius(1.0); sphere->SetThetaResolution(18); sphere->SetPhi...(18);

// map to graphics library
vtkPolyDataMapper *map = vtkPolyDataMapper::New();
map->SetInput(sphere->GetOutput());

// actor coordinates geometry, properties, transformation
vtkActor *aSphere = vtkActor::New();
aSphere->SetMapper(map); aSphere->GetProperty()->SetColor(0,0,1); // sphere color blue

// renderer and render window
vtkRenderer *ren1 = vtkRenderer::New();

// render window
vtkRenderWindow *renWin = vtkRenderWindow::New(); renWin->AddRenderer(ren1);

// interactor
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);

ren1->AddActor(aSphere);
renWin->Render();
iren->Start(); }

// add the actor to the scene
// render an image
// begin mouse interaction
int main( int argc, char *argv[] )
{
    vtkConeSource *cone = vtkConeSource::New();
    cone->SetHeight( 3.0 );
    cone->SetRadius( 1.0 );
    cone->SetResolution( 10 );
    vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    coneMapper->SetInput( cone->GetOutput() );
    vtkActor *coneActor = vtkActor::New();
    coneActor->SetMapper( coneMapper );
    // Create the Renderer and assign actors to it
    vtkRenderer *ren1= vtkRenderer::New();
    ren1->AddActor( coneActor );
    ren1->SetBackground( 0.1, 0.2, 0.4 );
    vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin->AddRenderer( ren1 );
    renWin->SetSize( 300, 300 );
}

Visualization Pipeline - Overview

- Three main elements:
  - Objects to represent data
    - data objects
  - Objects to represent processes
    - process objects
  - Direction of data flow
    - indicates data dependencies
    - reverse direction is dependency tree of later objects in pipeline
    - synchronisation required to keep pipeline up to date

- Also referred to as “visualization network”
  - may contain loops
Example: Visualizing a Quadric

- **Quadric**
  - second order surface function in R3 (more than 2 variables in def.)

\[
f(x,y,z) = a_0 x^2 + a_1 y^2 + a_2 z^2 + a_3 xy + a_4 yz + a_5 xz + a_6 x + a_7 y + a_8 z + a_9
\]

  - Here, \( F(x,y,z) = 0.5x^2 + 1.0y^2 + 0.2z^2 + 0.1yz + 0.2y \)
  - co-efficients: \( a,b,c,d,e,f,g,h \) variables: \( x,y,z \)

- **Task:** Visualize a quadric in the region \(-1 \leq x, y, z \leq 1\)

- **Process:**
  - Evaluate equation on a 50 x 50 x 50 regular grid
  - Extract 5 surfaces corresponding to values of the function \( F(x,y,z) = c \).
  - Generate a 3D outline round the data (bounding box)
Visualising a Quadric: Functional Model

- Visualization flow - process
  - (Text on left)
Visualization Pipeline
Data and Process Objects

• Data Objects
  – represent data (internally) + methods to access it
  – data modification only via formal object methods
    • follow abstract data type methodology
    • allows differing internal representations with common access semantics
  – additional methods for data properties / characteristics
    – Example: mesh
      • vertices, connectivity (basic)
      • polygons, normals at vertices or faces (additional)

• Process Objects
  – objects that operate on input data to generate output data
  – data transformation
  – source objects
    • generate data from local parameters (e.g. quadric) or external source (e.g. file)
  – filter objects
    • transform data from single/multiple inputs to single/multiple outputs
  – mapper objects
    • transform data into graphic primitives (for display or file output)
Visualizing a Quadric: Process objects

- **Source object**
  - procedural generation of quadric
  - `vtkQuadric`

- **Filter Objects**
  - `vtkContourFilter`
  - `vtkOutlineFilter`
  - although graphics representation still an internal representation

- **Mapper objects**
  - conversion to graphics primitives
  - `vtkPolyDataMapper`
F(x,y,z) = 0.5*x^2 + 1.0*y^2 + 0.2*z^2 + 0.1*y*z + 0.2*y

To visualize sample it over a regular array of points (i.e., a volume or structured point dataset), and then create iso-contours of the quadric F(x,y,z) = c, where c is a constant (i.e., the contour value).

Visualization network three filters:
1. vtkSampleFunction, samples the quadric equation
2. vtkContourFilter, creates iso-contours for 0, 1, 2, or 3D data
3. vtkPolyMapper, maps the data to the graphics system

(Also create an outline around the data for context).

```cpp
#include "vtkQuadric.h"
#include "vtkSampleFunction.h"
#include "vtkContourFilter.h"
#include "vtkOutlineFilter.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkRenderWindow.h"
#include "vtkRenderer.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkImageData.h"

void main ()
{
    // -- create the quadric function object --

    // create the quadric function definition
    vtkQuadric *quadric = vtkQuadric::New(); //...vtkImplicitFunction
    quadric->SetCoefficients(.5,1,.2,0,.1,0,0,.2,0,0);

    // sample the quadric function
    vtkSampleFunction *sample = vtkSampleFunction::New();
    sample->SetSampleDimensions(50,50,50);
    sample->SetImplicitFunction(quadric);

    // Create five surfaces F(x,y,z) = constant between range specified
    vtkContourFilter *contours = vtkContourFilter::New();
    contours->SetInput(sample->GetOutput());
    contours->GenerateValues(5, 0.0, 1.2);

    // map the contours to graphical primitives
    vtkPolyDataMapper *contMapper = vtkPolyDataMapper::New();
    contMapper->SetInput(contours->GetOutput());
    contMapper->SetScalarRange(0.0, 1.2);

    // create an actor for the contours
    vtkActor *contActor = vtkActor::New();
    contActor->SetMapper(contMapper);
```
// -- create a box around the function to indicate the sampling volume --

// create outline
vtkOutlineFilter *outline = vtkOutlineFilter::New();
outline->SetInput(sample->GetOutput());

// map it to graphics primitives
vtkPolyDataMapper *outlineMapper = vtkPolyDataMapper::New();
outlineMapper->SetInput(outline->GetOutput());

// create an actor for it
vtkActor *outlineActor = vtkActor::New();
outlineActor->SetMapper(outlineMapper);
outlineActor->GetProperty()->SetColor(0,0,0);

// -- render both of the objects -- ........ which we know how to do

// a renderer and render window
vtkRenderer *ren1 = vtkRenderer::New();
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(ren1);

// an interactor
vtkRenderWindowInteractor *iren =
  vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);

// add the actors to the scene
ren1->AddActor(contActor);
ren1->AddActor(outlineActor);
ren1->SetBackground(1,1,1); // Background color white

renWin->Render(); // render an image (lights and cameras are created automatically)
iren->Start(); // begin mouse interaction

F(x,y,z) = 0.5*x^2 + 1.0*y^2 + 0.2*z^2 + 0.1*y*z + 0.2*y
Visualization Pipeline Connections
(this is pretty abstract stuff … still need to use …)

- **Sources, filters, and mappers** can be connected in variety of ways

- Connectivity Issues:
  - 1. **Type**: restrictions on types of data that a module can handle (as input / for output)
  - 2. **Multiplicity**: number of inputs / outputs supported

- **Untyped**:
  - all data objects treated the same - any process object can connect to any other
  - generic data access methods
  - **disadvantages**
    - generic data representation (i.e. single type) **cannot take advantage of optimised representations**
      - inefficient data representation for certain data types
      - e.g. 2D images, 3D volume, unstructured 3D point cloud all have different optimal data types for access
    - extra burden of data transformation onto process objects

- **Typed**
  - only objects of compatible types can be connected
  - **bad design can lead to explosion of types** and thus type converters
  - data can be represented in a **specific representation optimised towards requirements** for {access, transformation}

- VTK is strongly typed
Multiplicity of Connections

- Two special multi-connection cases:
  - **Fan out**
    - one module supplies the same data to many other modules
      - 1 output : N module connectivity
  - **Multiple outputs**
    - one module producing a number of different outputs that connect to different modules
      - N outputs : N module connectivity

- Multiplicity allows the consideration of parallel processing in the visualisation pipeline
  - useful for “real-time” type demands on large data sets

- **Sequential processing** with multiplicity is a more common alternative

```
Source  Filter  Mapper
    ↓     ↓       ↓
    Filter Filter Filter
        ↓  ↓       ↓
        Data Data Data

Single Output  Fan Out  Multiple Output
```
Visualization Pipeline : Execution Control

• **Problem**: ensuring all parts of pipeline are up to date if a parameter is modified by user, and ensure synchronisation is maintained?

• **Solutions**:
  – 1. **Event-Driven**:
    • centralised executive (i.e. controller) notes change occurrences and reexecutes effected modules
  – 2. **Demand-Driven**
    • when output is requested by a mapper object, the network is reexecuted starting with source objects
**Event-driven Execution Control**

- **Explicit Control** of Execution by executive
  - **Advantages:** only update required modules (i.e. objects)
  - **Disadvantages:** complexity of control, updates even without user demand, update called for every trivial modification, even if it’s not needed, making a series of modifications re-executes the network multiple times.

![Diagram of the execution control system](image)
Demand-driven Execution Control

- **Implicit Control** of Execution by pipeline dependency tree
- **Advantages**: simplicity, no global knowledge required
- **Disadvantages**: inefficient to re-execute module if nothing has changed

Mapper object E requests output
- Chain E-D-B-A back propagates via Update() method
- Chain A-B-D-E executes via Execute() method.
Execution Control - Methods

• **Event-driven**
  – control of modules is *explicit* by executive
  – pipeline is always up to date (even if not required)
  – distribute modules (tasks) across computers

• **Demand-driven**
  – control of modules is *implicit* by user/module requests
  – simple, no central point of control/failure
  – do not re-execute a module unless required for output
Memory Recomputation Trade-off

- **Problem**: do we store intermediate results in the pipeline?

- **Yes =>** keep memory allocated => **static memory model**
  - memory intensive, **beware** of large datasets
  - **saves** computation

- **No =>** release memory allocation => **dynamic memory model**
  - module may need to be re-executed
  - computation intensive, **beware** of slow processors
    - (or large data sets too!)
  - **saves** memory

- Modules are dependent on results from previous modules

- **A & B execute twice with a dynamic memory model** if C and D execute and **once with a static memory model**

- Best solution: Dependency analysis
Visualization Pipeline in VTK

• Each module is a VTK object (C++/TCL/Java/Python)
  – Connect modules together by using:
    • SetInput()
    • GetOutput()
    • e.g. to connect modules A and B so B takes as input the output of B
      – TCL: A setInput [ B GetOutput]
      – Java: A.SetInput( B.GetOutput() );

• VTK pipeline
  – demand-driven execution control maintained implicitly
  – memory for intermediate results can be explicitly controlled
    • by default (static model)
End
Exercise

• Execute quadratic function program

• Modify it, e.g., different sampling, different coefficients, different function, etc.

• Try to learn as much as you can in a reasonable time

• Describe what you have learned