Chapter 2
VTK – The Visualization Toolkit

An introduction based on

- *Visualizing with VTK: A Tutorial*, Schroeder, Avila and Hoffman, IEEE Computer Graphics and Applications, Vol. 20, No. 5, pp. 20-27.

- *The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization*, Schroeder, Martin and Lorensen, Proceedings of IEEE Visualization '96, pp. 93-100.

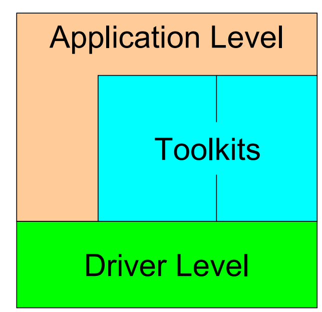
Overview

2.1 Design Goals

- Toolkit Philosophy
- Interpreted Language Interface
- Standards Based
- Portable
- Freely Available
- Simple

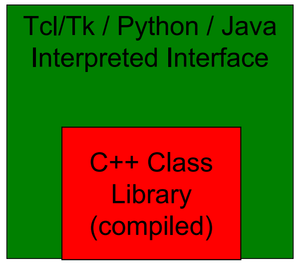
Toolkit Philosophy

- Sharply focused object library
- Easily embedded in applications
- Enables the building of complex systems
  - Pieces well defined
  - Simple interfaces



Interpreted Language Interface

- Compiled languages
  - Faster
  - Low level manipulations
- Interpreted
  - Simpler more compact code
  - Faster application development
  - Higher level
  - Easier to debug



Standards Based

- Use standard components and languages
- Encourages use of the toolkit
- Eases support and maintenance

Portable

- Authors skeptical that any graphics library will ever become a “standard.”
  - Toolkit uses high-level abstraction for 3D graphics
  - System can be easily ported as new standards become available
- Toolkit independent of system
  - Operating system
  - Windowing system

Freely Available

- For software to succeed it must be
  - Widely used (cheap/useful)
  - Well supported (expandable/source code available)
- Benefits
  - Better dissemination of algorithms
  - Collaboration with other researchers
  - Credibility in the Visualization field
  - Used for education and research

Simple

"Everything should be as simple as possible, but no simpler" – Albert Einstein

- Benefits
  - Encourages wider use of 3D graphics and visualization
  - Easier to maintain
  - Easier to interface
  - Easier to extend?
- Avoid cool but complex toolkit features
  - Interesting to programmers … but overwhelming to users

2.2 Object Models

- Graphics Model
  - Abstract model of 3-D graphics
- Visualization Model
  - Data flow model of the visualization process

Graphics Model

- Render Window – manages window
- Renderer – coordinates rendering
- Light – illuminates the scene
- Camera – view of scene
- Actor – object in scene
- Property – appearance of actor
- Mapper – geometry of actor
- Transform – position and orientation of actor, camera, lights

Device Dependent Subclasses

- Portability of the design achieved by using device objects, which extend the functionality of graphics classes in a device dependent way.
  - The VTK toolkit returns a subclass specific to the system
- Example:

```
vtkRenderMaster rm;
renderWindow = rm.MakeRenderWindow();
aRen = renderWindow->MakeRenderer();
```

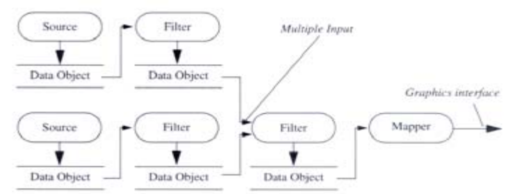
Application running on Sun UNIX creates an X-Windows window and a SUN XGL renderer whereas on a PC it creates a Windows rendering window and an OpenGL renderer.

Visualization Model

- Data flow paradigm
  - Modules connected to form a network.
  - Data flows through network, modules perform operations on the data.
  - Execution demand driven (pulls data from source) or event driven (responds to user input).
- Visualization model consists of
  - Process objects – visualization algorithms
  - Data objects – datasets to be visualized

Process Objects

- Sources
  - Generate output datasets
- Filters
  - Transform datasets into new datasets
- Mappers
  - Map datasets into Actors (graphics objects)



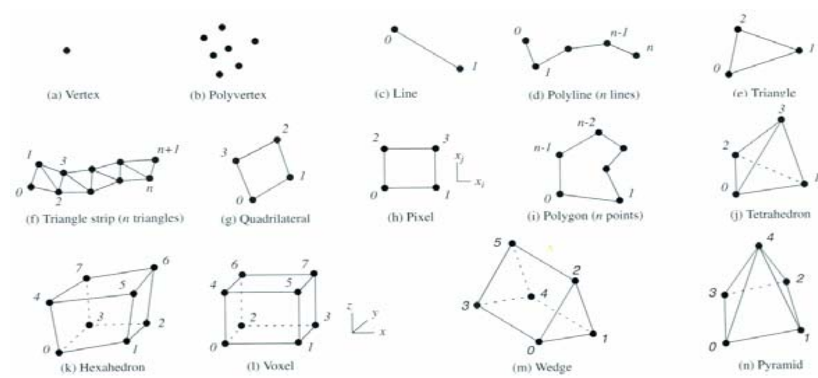
© 2003 VTK User's Guide, Kitware Inc.

Dataset Objects

Data objects have a …

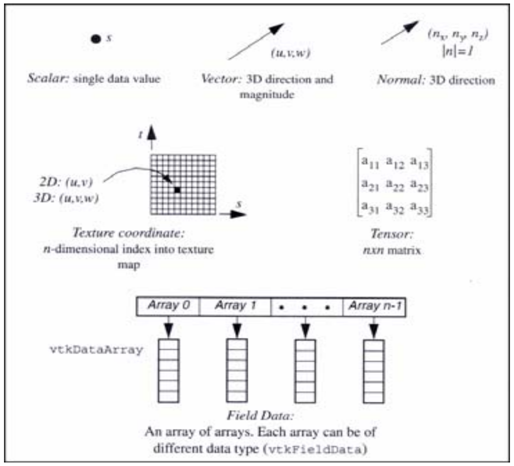
- Structure consisting of
  - Points: specify geometry (position in space)
  - Cells: specify topology (type of shape, allows interpolation between points)
- Associated Data Attributes
  - information associated with topology and/or geometry, e.g. scalars, vectors, normals, tensors, texture coordinates.

Cell Types

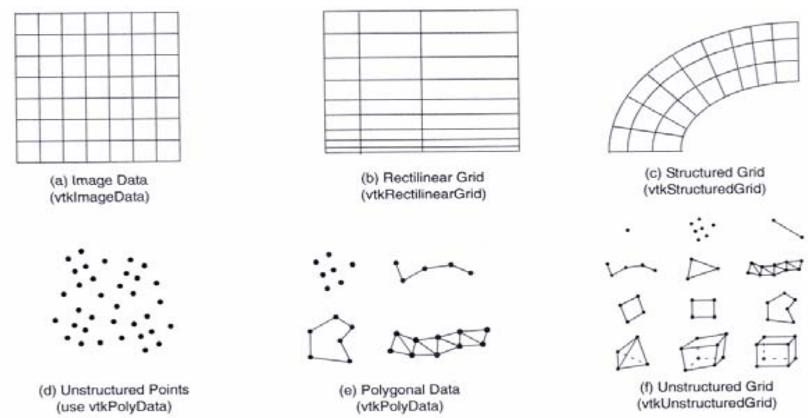


© 2003 The Visualization Toolkit, Schroeder, Martin, Lorensen

Attribute Data



Types of Data



Types of Data (cont'd)

- Image Data (`vtkImageData`)
  - Topology and geometry completely regular.
  - Represented implicitly by data dimension (n_x, n_y, n_z), origin, spacing.
- Rectilinear Grid (`vtkRectilinearGrid`)
  - Collection of points and cells on a regular lattice.
- Structured Grid (`vtkStructuredGrid`)
  - Regular topology and irregular geometry.
  - Geometry represented by array of point coordinates.
- Unstructured Grid (`vtkUnstructuredGrid`)
  - The most general form of a dataset.
  - Topology and geometry completely unstructured.
- Polygonal data (`vtkPolyData`)
  - Bridge between data, algorithms and high-speed computer graphics.

Object Oriented Design

- Generic Filter
  - Operates on any type of data (e.g. contour filter)
- Specific Filter
  - Operates only on one particular type of data (e.g. the decimation filter has been specifically constructed for polygonal data)
- Allows the implementer to trade of generality with efficiency

2.3 Implementation Issues

- Why C++?
  - Efficient *and* object-oriented
  - Strongly typed
- Get/Set macros
  - Uniform access to all object variables
  - Debugging, auditing (tracks modifications)
  - Enforce uniform object behaviour (e.g. maintain internal modification time → network execution)

Memory Management

- Garbage Collection
  - Datasets often shared by multiple processes
  - Dataset objects maintain reference counters
  - When reference count is zero, object commits suicide (deletes itself).
- Resource Management
  - Memory scarce – delete result after use
  - CPU scarce – save result after use

Making OO Fast

- Avoid creating/destroying large numbers of objects
  - Datasets are large but contained in single object
- Minimize data copying
  - Datasets encapsulated in objects
  - Dataset objects passed by reference
- Reduce object function overhead
  - Use inline functions when possible

2.4 Example

```
// Create a cone represented by polygons
vtkConeSource *cone = vtkConeSource::New();
cone->SetHeight( 3.0 );
cone->SetRadius( 1.0 );
cone->SetResolution( 10 );

// map the polygonal data into graphics primitives.
// Connect the output of the cone source to the
// input of this mapper.
vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
coneMapper->SetInput( cone->GetOutput() );
```

Example (cont'd)

```
// Create an actor to represent the cone. The actor
// orchestrates rendering of the mapper's graphics
// primitives using given properties and an
// internal transformation matrix.
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 );
```


Example (cont'd)

```
// Create the render window put renderer into it
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer( ren1 );
renWin->SetSize( 300, 300 );

// Loop over 360 degrees and
// render the cone each time.
int i;
for (i = 0; i < 360; ++i){
  renWin->Render();
  ren1->GetActiveCamera()->Azimuth( 1 );
}
```