Week 3: Simple Visualization Tasks

http://noodle.med.yale.edu/~papad/seminar/
Man Pages etc: http://noodle.med.yale.edu/tcl
and http://noodle.med.yale.edu/vtk/

Xenios Papademetris
papad@noodle.med.yale.edu
BML 325, 5-7294

Revised Schedule for Part 1

1. Introduce VTK (9/3)
2. Introduce Tcl/Tk (9/10)
3. Simple Visualization Tasks (9/17)
4. Image and Surface Manipulation (9/23)
5. Image Display (Volume Rendering) (10/1)
6. Local Extensions (10/8)
7. Tk and GUIs (10/15)
8. [Incr] Tcl and other extensions (10/22)

Flashback
Object Oriented Programming (OOP)

• In OOP data is “intelligent” i.e. data structures encapsulate procedures i.e.

• If we have matrix a to print it we do a print

• Procedures are embedded within the data structure definition (called methods)

OOP and Basic Tcl

• Tcl can be loosely termed as semi object oriented.
  – Tcl cannot be used to define new types of objects
  – Tcl can use objects defined in other languages (e.g. C++) if these are appropriately imported
  – Examples of such objects are the widgets in Tk, and all the VTK classes
  – In Part II of this seminar we will look into how to add our own code to Tcl as objects.

• There are a number of extension packages to make Tcl fully OOP we will discuss [incr Tcl] later in the semester.

Revision -- VTK Pipeline (I)

A source/filter can drive more than one filter/mapper

Revision -- VTK Pipeline II

No mapper/prop should be added to more than one renderer
**Example 1 — `example3_1.tcl`**

```
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10
# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]
# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map
# Renderer
tkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
# Render Window
tkRenderWindow renWin
renWin AddRenderer ren1
renWin SetSize 300 300
```

**Example 1 — `vtkConeSource`**

```
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10
# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]
# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map
# Renderer
tkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
# Render Window
tkRenderWindow renWin
renWin AddRenderer ren1
renWin SetSize 300 300
```

**Example 1 — `vtkPolyDataMapper`**

```
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10
# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]
# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map
# Renderer
tkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
# Render Window
tkRenderWindow renWin
renWin AddRenderer ren1
renWin SetSize 300 300
```

**Example 1 — `vtkActor`**

```
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10
# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]
# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map
# Renderer
tkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
# Render Window
tkRenderWindow renWin
renWin AddRenderer ren1
```

---

**vtkPolyDataSource → vtkConeSource**

- Abstract base class of sources that generate surfaces (`vtkPolyData`)
- Number of derived classes to:
  1. Load a surface from a file (e.g. `vtkPolyDataReader`)
  2. Generate a surface programmatically (e.g. `vtkConeSource`)
  3. Extract a surface as an iso-contour from an image (e.g. `Marching Cubes`)

- Key methods
  - `Update` — explicit execution
  - `GetOutput` — gets the output (often sufficient for implicit execution)

**vtkAbstractMapper → vtkPolyDataMapper**

- Abstract base class specify interface between data and graphics primitives or software rendering techniques
- Number of derived classes to:
  1. Map 3D Surfaces (e.g. `vtkPolyDataMapper`)
  2. Map Generic Data Sets (e.g. `vtkDataSetMapper`)
  3. Map volumes for volume rendering (e.g. `vtkVolumeMapper`)

- Key methods
  - `SetInput` — sets the input data set to map
  - `SetColorMode` — determines how to color the surface
vtkProp → vtkProp3D → vtkActor

- Abstract base class for all actors, volumes, and annotations.
- Represents an object for placement in a rendered scene.
- Number of derived classes to
  1. Represent 3D Geometric Objects (e.g. vtkActor)
  2. Represent Volumes for volume rendering (e.g. vtkVolume)
  3. Generate an x-y plot from input dataset(s) (e.g. vtkXYPlotActor)

  - Key methods
    - SetMapper -- sets the mapper that generates the object
    - GetProperty -- access the property member that defines display properties (e.g. color/ surface vs wireframe)

Example 1 – vtkRenderer

```plaintext
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10

# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]

# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map

# Renderer
vtkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
```

Example 1 – vtkRenderWindow

```plaintext
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10

# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]

# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map

# RenderWindow
vtkRenderWindow renWin
renWin AddRenderer ren1
renWin SetSize 300 300

# Interactor
vtkRenderWindowInteractor iren
iren SetRenderWindow renWin
iren Initialize
iren AddObserver SetExitMethod { exit }

# Start the event loop
wm withdraw .
vwait forever
```

vtkViewport → [] → vtkRenderer

- A renderer is an object that controls the rendering process for objects.
- Rendering is the process of converting geometry, a specification for lights, and a camera view into an image.
- vtkRenderer also performs coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device).
- Controls certain advanced rendering features e.g. two-sided lighting

  - Key methods
    - AddActor/RemoveActor -- adds/removes an actor from the scene
    - AddVolume/RemoveVolume -- adds/removes a volume
    - GetCamera -- gets the camera used for rendering
    - SetBackground -- background color for display
    - SetViewport -- portion of window to occupy (useful when multiple renderers are put into one window)

Example 1 – vtkRenderWindow

```plaintext
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10

# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]

# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map

# Render
vtkRenderer ren1
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
```

vtkWindow → vtkRenderWindow

- A rendering window is a window in a graphical user interface where renderers draw their images. Methods are provided to synchronize the rendering process, set window size, and control double buffering.
- It can contain a number of renderers (vtkRenderer)
- It controls the display update rate

  - Key methods
    - AddRenderer/RemoveRenderer
    - SetDesiredUpdateRate -- controls frame/seconds for display
    - SetSize -- Size of the window
    - Render -- forces a rendering operation

  - vtkRenderWindow is also an abstract base class

Example 1 – vtkRenderWindow

```plaintext
# Source
vtkConeSource cone
cone SetHeight 3.0
cone SetRadius 1.0
cone SetResolution 10

# Mapper
vtkPolyDataMapper map
map SetInput [cone GetOutput]

# Prop -- Actor
vtkActor coneActor
coneActor SetMapper map

# RenderWindow
vtkRenderWindow renWin
renWin AddRenderer ren1
renWin SetSize 300 300

# Interactor
vtkRenderWindowInteractor iren
iren SetRenderWindow renWin
iren Initialize
iren AddObserver SetExitMethod { exit }

# Start the event loop
wm withdraw .
vwait forever
```

vtkWindow → vtkRenderWindow

- Some Functionality is Platform Specific
- Platform independent alternative vtkTkRenderWindow which is a Tk widget that you can render into. It has a GetRenderWindow method that returns an appropriate vtkRenderWindow, but does not allow for Interactors (see TkInteractor.tcl for how to do this)
**vtkRenderWindowInteractor**

- vtkRenderWindowInteractor provides a platform-independent interaction mechanism for mouse/key/time events.
- It serves as a base class for platform-dependent implementations that handle routing of mouse/key/timer messages to vtkInterActorStyle and its subclasses.
- vtkRenderWindowInteractor also provides controls for picking, rendering frame rate, and headlights.
- Does not always work well if using vtkTkRenderWidget (Best Avoided)

**Playing with vtkProperty**

*(example3_2.tcl)*

- The appearance of each prop (e.g. vtkActor) is controlled by its property member (of type vtkProperty)
- To modify the appearance of the actor we use methods associated with the property member e.g.

  **Step 1** – get a reference to the property member
  ```
  set property [ coneActor GetProperty ]
  ```

  **Step 2** – do something e.g. change the color etc.
  ```
  $property SetColor 1 0 0
  $property SetAmbient 1.0 (Default for surface 0.0)
  $property SetDiffuse 0.0 (Default for surface 1.0)
  $property SetSpecular 0.0
  $property SetRepresentationToWireframe
  ```

**Adding a Filter**

*(example3_3.tcl)*

- We can modify the filter produced by cone source by inserting a filter in the pipeline e.g. replace

  ```
  # Mapper
  vtkPolyDataMapper map
  map SetInput [cone GetOutput]
  with
  ```

  ```
  #Filter
  vtkLinearSubdivisionFilter subd
  subd SetInput [ cone GetOutput ]
  ```

**vtkPolyDataToPolyDataFilter → vtkLinearSubdivisionFilter**

- Filters are Sources with Inputs
- The filter generally does something to an input to produce an output
- Filters DO NOT modify their inputs
- `vtkPolyDataToPolyDataFilter` is the parent class of a number of filters that operate on surfaces to produce surfaces
- Large Number of derived classes to
  1. Smooth of surface (e.g. `vtkSmoothPolyDataFilter`)
  2. Decimate a surface (e.g. `vtkDecimate`) (see vtkTestSmoothing.tcl)
  3. Compute Surface Normals (e.g. `vtkPolyDataNormals`)
  4. Transform a Surface (e.g. `vtkTransformPolyDataFilter`)
  5. Increase number of polygons (e.g. `vtkLinearSubdivisionFilter`)

- Key methods
  - `Update` -- explicit execution
  - `GetOutput` -- gets the output
  - `SetInput` -- sets the input to the filter

**A Closer Look at vtkPolyData**

- vtkPolyData is a complex class which has many members. The key ones are:
  - Points of type `vtkPoints` -- represents the geometry of the surface (i.e. the points)
  - Polys of type `vtkCellArray` -- represents part of the topology of the surface (i.e. the polygons)
  - PointData of type `vtkPointData` -- represents data associated with the points (e.g. normals, colors etc)
  - Lots of other members .....
ShallowCopy vs DeepCopy

- `vtkPolyData` (Points, Polys, PointData)
- Consider the following piece of code:
  ```c
  vtkPolyData a
  vtkPolyData b
  b DeepCopy a;
  ```
  - In this example object b has its own independent copies of Points, Polys and PointData. Modifying object a does not affect object b.
  - Replace last line with:
  ```c
  b ShallowCopy a;
  ```
  - Here b simply has pointers to the elements of a, no copying is done. Modifying the points of a will modify b as it only has a pointer to the points, not its own copy.

Shallow Copy and DeepCopy

- In general, Vtk filters do not modify their inputs directly; they do a DeepCopy and modify their output.
- Use DeepCopy when you want to modify an object directly.
- Use ShallowCopy to ensure that an object does not get deleted – this is useful for partial pipeline execution (next slide).

Breaking the Pipeline

```c
set objcount 0
proc TriangulateInterpolateAndSmooth { surface_in iterations } {     
global objcount
incr objcount
set triF [ vtkTriangleFilter tri$objcount ]
$triF SetInput $surface_in
set subd [ vtkLinearSubdivisionFilter subd$objcount ]
$subd SetInput [ $triF GetOutput ]
$triF Delete
set smooth [ vtkSmoothPolyDataFilter smooth$objcount ]
$smooth SetInput [ $subd GetOutput ]
$smooth SetNumberOfIterations $iterations
$smooth BoundarySmoothingOn
$subd Delete, $smooth Update
set surface_out [ vtkPolyData surface$objcount ]
$surface_out ShallowCopy [ $smooth GetOutput ]
$smooth Delete
return $surface_out
}
```

Breaking the Pipeline – a procedure

```
set objcount 0
proc TriangulateInterpolateAndSmooth { surface_in iterations } {     
global objcount
incr objcount
set triF [ vtkTriangleFilter tri$objcount ]
$triF SetInput $surface_in
set subd [ vtkLinearSubdivisionFilter subd$objcount ]
$subd SetInput [ $triF GetOutput ]
$triF Delete
set smooth [ vtkSmoothPolyDataFilter smooth$objcount ]
$smooth SetInput [ $subd GetOutput ]
$smooth SetNumberOfIterations $iterations
$smooth BoundarySmoothingOn
$subd Delete, $smooth Update
set surface_out [ vtkPolyData surface$objcount ]
$surface_out ShallowCopy [ $smooth GetOutput ]
$smooth Delete
return $surface_out
}
```

Breaking the Pipeline – Calling the procedure

```
vtkCylinderSource cylinder
  cylinder SetResolution 10
  cylinder CappingOn
  cylinder Update
# Filter
set sur_out \       [ TriangulateInterpolateAndSmooth [ cylinder GetOutput ] 100 ]
cylinder Delete
# Mapper
vtkPolyDataMapper cylinderMapper
cylinderMapper SetInput $sur_out
```

ShallowCopy II

- Consider VtkPolyData object `a` with members (pt, pl, pd)
- When `a` is created the reference counts are `a=1, pl=1, pd=1`
- When we execute the ShallowCopy command the ref counts become `a=1, pl=2, pd=2` as `b` adds reference to `pt, pl, pd`
- If we then execute:
  ```c
  a delete
  ```
  - Which calls (`pt delete, pl delete, pd delete`) the reference counts become `a=0, pl=1, pd=1`. In this case `a` gets deleted put `pt, pl` and `pd` are still around, since `b` has a reference to them, they will get deleted once `b` gets deleted.

Breaking the Pipeline

```
set objcount 0
proc TriangulateInterpolateAndSmooth { surface_in iterations } {     
global objcount
incr objcount
set triF [ vtkTriangleFilter tri$objcount ]
$triF SetInput $surface_in
set subd [ vtkLinearSubdivisionFilter subd$objcount ]
$subd SetInput [ $triF GetOutput ]
$triF Delete
set smooth [ vtkSmoothPolyDataFilter smooth$objcount ]
$smooth SetInput [ $subd GetOutput ]
$smooth SetNumberOfIterations $iterations
$smooth BoundarySmoothingOn
$subd Delete, $smooth Update
set surface_out [ vtkPolyData surface$objcount ]
$surface_out ShallowCopy [ $smooth GetOutput ]
$smooth Delete
return $surface_out
}
```

Breaking the Pipeline – a procedure

```
set objcount 0
proc TriangulateInterpolateAndSmooth { surface_in iterations } {     
global objcount
incr objcount
set triF [ vtkTriangleFilter tri$objcount ]
$triF SetInput $surface_in
set subd [ vtkLinearSubdivisionFilter subd$objcount ]
$subd SetInput [ $triF GetOutput ]
$triF Delete
set smooth [ vtkSmoothPolyDataFilter smooth$objcount ]
$smooth SetInput [ $subd GetOutput ]
$smooth SetNumberOfIterations $iterations
$smooth BoundarySmoothingOn
$subd Delete, $smooth Update
set surface_out [ vtkPolyData surface$objcount ]
$surface_out ShallowCopy [ $smooth GetOutput ]
$smooth Delete
return $surface_out
}
```

Breaking the Pipeline – Calling the procedure

```
vtkCylinderSource cylinder
  cylinder SetResolution 10
  cylinder CappingOn
  cylinder Update
# Filter
set sur_out \       [ TriangulateInterpolateAndSmooth [ cylinder GetOutput ] 100 ]
cylinder Delete
# Mapper
vtkPolyDataMapper cylinderMapper
cylinderMapper SetInput $sur_out
```