Rendering Primitives

Ken Martin
Kitware Inc.

Outline

- Introduction to Rendering
- Visualization Data Model
- VTK's Graphics Subsystem
- Rendering a Polygonal Mesh
- Rendering an Image

- In the context of visualization rendering is the process of converting visualization primitives into a 2D image
- more generally it is converting visualization primitives into something that can be visually perceived by the user, this includes physical models (such as from stereo lithography), 3D images, etc
- Typically the resulting image is a series of pixels each containing a Red, Green, and Blue value.
- The physical world is vastly more complicated and many rendering engines support (or approximate) some of this complexity (functions over wavelength, polarity, etc)
- In visualization we can take advantage of this complexity

What are visualization primitives?

- 2D surface elements such as triangles, and polygons
- 3D volumetric elements such as tetrahedra and voxels
  - (Lisa will discuss this in her section of Volume Rendering)
- Higher order elements such as NURBS, etc
- Analytic primitives (less widely supported)
- 0D and 1D elements such as points and lines

- All the above are typically represented in a 3D world coordinate system

What information goes with a visualization primitive?

- Geometry (the 3D positions)
- Topology (the connectivity of the elements)
- Normals (a unit vector normal to the surface)
- Color (RGBA)
  - Emissive (Ambient)
  - Diffuse
  - Specular
  - Specular Power
- Texture Coordinates
- Texture Map
- Other Generalizations (Displacement Maps etc)

What information goes with a visualization primitive?

- Interpolation (Phong, Gouraud, Flat)
- Backface / Frontface properties

Visualization algorithms map from the data being visualized to visualization primitives

- Density scalar values → colors, isosurfaces
- Velocity vector fields → streamlines, glyphs

Visualization Data Model

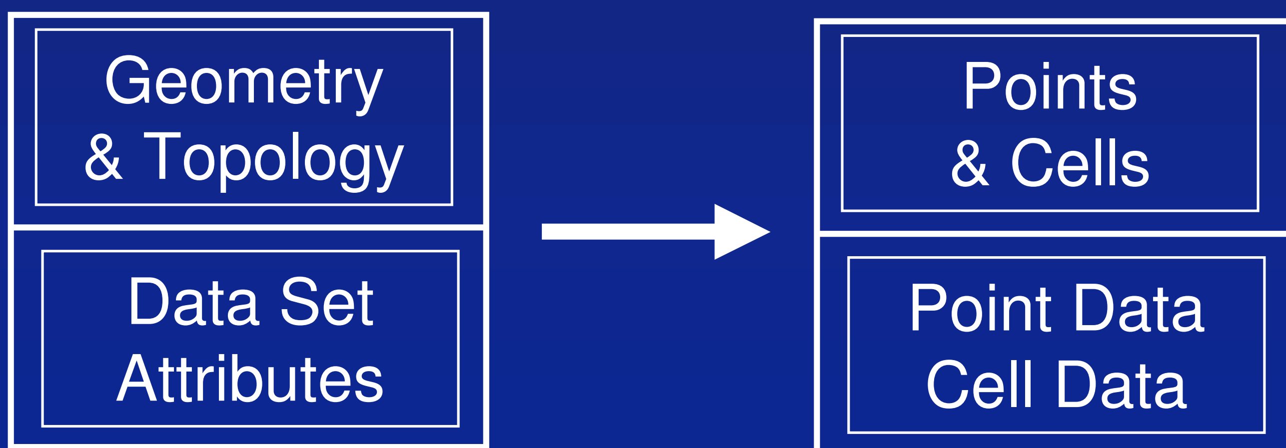
- To understand rendering for visualization we will look at the original data and the rendering process
- We will use VTK as a framework (and examples) for this discussion (other toolkits have similar concepts and names)
- VTK is a visualization toolkit
  - Designed and implemented using object-oriented principles
  - C++ class library (400,000 LOC, <150,000 executable lines)
  - Automated Java, TCL, Python bindings
  - Portable across Unix, Windows9x/NT
  - Supports 3D/2D graphics, visualization, image processing, volume rendering

Visualization Data Model

- Data Objects
  - represent data
  - provide access to data
  - compute information particular to data (e.g., bounding box, derivatives)
- Represent a “blob” of data
  - contain instance of vtkFieldData
  - an array of arrays
  - no geometric/topological structure
  - typically not used in pipelines (but its subclasses such as vtkDataSet are)
- Can be converted to vtkDataSet
  - vtkDataObjectToDataSetFilter

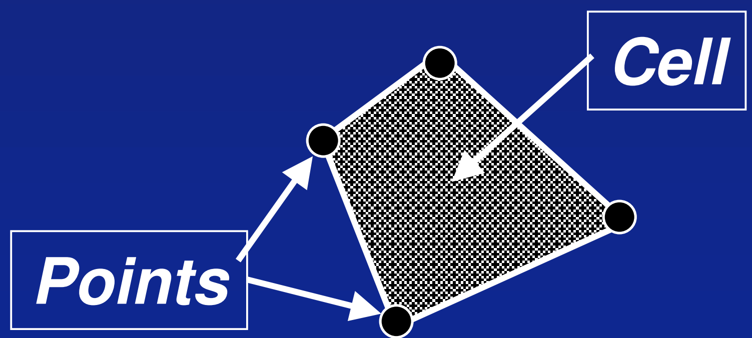
Visualization Data Model

- vtkDataObject is a “blob” of data
  - Contains an instance of vtkFieldData
- vtkDataSet is data with geometric & topological structure; and with <u>attribute</u> data

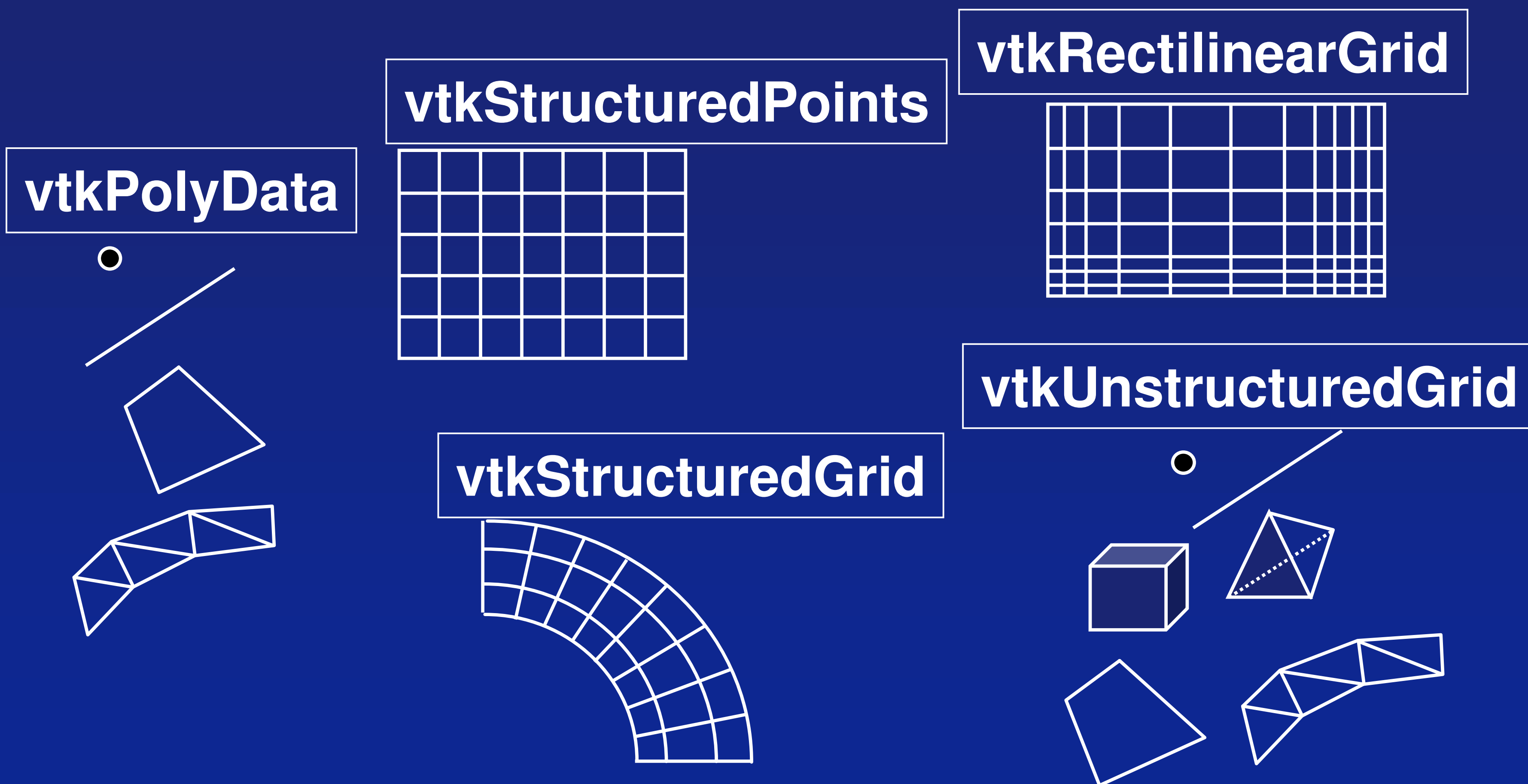


Visualization Data Model

- A dataset is a data object with structure
- Structure consists of
  - cells (e.g., polygons, lines, voxels)
  - points (x-y-z coordinates)
  - cells defined by connectivity list referring to points
  - implicit representations
  - explicit representations



Visualization Data Model



Visualization Data Model

- **vtkDataArray labeled as:**
  - **Scalars** – single value
  - **Vectors** - 3-vector
  - **Tensors** - 3x3 symmetric matrix
  - **Normals** - unit vector
  - **Texture Coordinates** 1-3 values
  - **Field Data** (arbitrary arrays)
- The values in the data arrays must be mapped to values of visualization primitives

VTK’s Graphics Subsystem

- A VTK scene consists of:
- vtkRenderWindow - contains the final image
- vtkRenderer - draws into the render window
- vtkActor - combines properties / geometry
  - vtkProp, vtkProp3D are superclasses
  - vtkProperty
- vtkLights - illuminate actors
- vtkCamera - renders the scene
- vtkMapper - represents geometry
  - vtkPolyDataMapper, vtkDataSetMapper are subclasses
- vtkTransform - position actors

VTK's Graphics Subsystem

```
vtkSphereSource *sphere = vtkSphereSource()::New();

vtkPolyDataMapper *sphereMapper = vtkPolyDataMapper::New();
  sphereMapper→SetInput(sphere→GetOutput());
vtkActor *sphereActor = vtkActor::New();
  sphereActor→SetMapper(sphereMapper);

vtkRenderer *renderer = vtkRenderer::New();
vtkRenderWindow *renWin = vtkRenderWindow::New();
        renWin→AddRenderer(renderer);
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
        iren→SetRenderWindow(renWin);

renderer→AddProp(sphereActor);
renderer→SetBackground(1,1,1);
renWin→SetSize(300,300);

renWin→Render();
iren→Start();
```

- The following is a summary of instance variables & methods
- Remember there is typically a Set__() and Get___() method to set and get the instance variable values.
- Refer to Doxygen man pages, or class header files, for more information.

VTK's Graphics Subsystem

- Converting datasets to visualization primitives is mainly handled by mappers, with some help from properties and actors

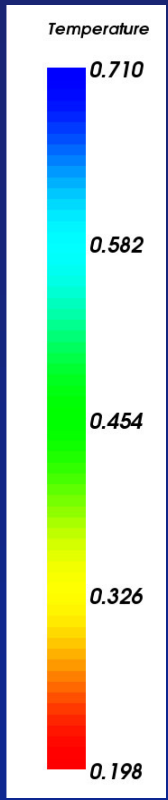
vtkMapper (vtkVolumeMapper, vtkPolyDataMapper, etc

- Controls which scalar array is used for vertex (or cell) colors
- Defines a mapping from scalar values to colors using a lookup table and scalar range
- Defines how the vertex colors are used to control the lighting equations
- Fairly intuitive mapping from geometry and topology to visualization primitives

VTK's Graphics Subsystem

vtkLookupTable

- NumberOfColors – number of colors in the table
- TableRange – the min/max scalar value range to map
- If building a table from linear HSVA ramp:
  - HueRange – min/max hue range
  - SaturationRange – min/max saturation range
  - ValueRange – min/max value range
  - AlphaRange – min/max transparency range
- If manually building a table
  - Build (after setting NumberOfColors)
  - SetTableValue(idx, rgba) for each NumberOfColors entries



vtkProperty

- Interpolation - shading interpolation method
  (Flat, Gouraud, Phong)
- Representation – how to represent itself
  (Points, Wireframe, Surface)
- AmbientColor, DiffuseColor, SpecularColor – a different color for ambient, diffuse, and specular lighting
- Color – sets the three colors above to the same
- Ambient, Diffuse, Specular – coefficients for ambient, diffuse, and specular lighting
- Opacity – control transparency

VTK's Graphics Subsystem

vtkActor (subclass of vtkProp)

- Combines the visualization primitives from the mapper with transformations and properties
- Property – surface lighting properties
- Texture – a texture map associated with the actor
- Position – where it's located
- Origin – the origin of rotation
- Visibility – is the actor visible?
- Pickable – is the actor pickable?
- Dragable – is the actor dragable?
- RotateX, RotateY, RotateZ – rotate around different axes
- RotateWXYZ – rotate around a vector

vtkCamera

- Position – where the camera is located
- FocalPoint – where the camera is pointing
- ViewUp – which direction is "up"
- ClippingRange – data outside of this range is clipped
- ViewAngle – the camera view angle controls perspective effects
- EyeAngle – the angle between eyes (for stereo)
- ViewPlaneNormal – the normal vector to the view plane

VTK's Graphics Subsystem

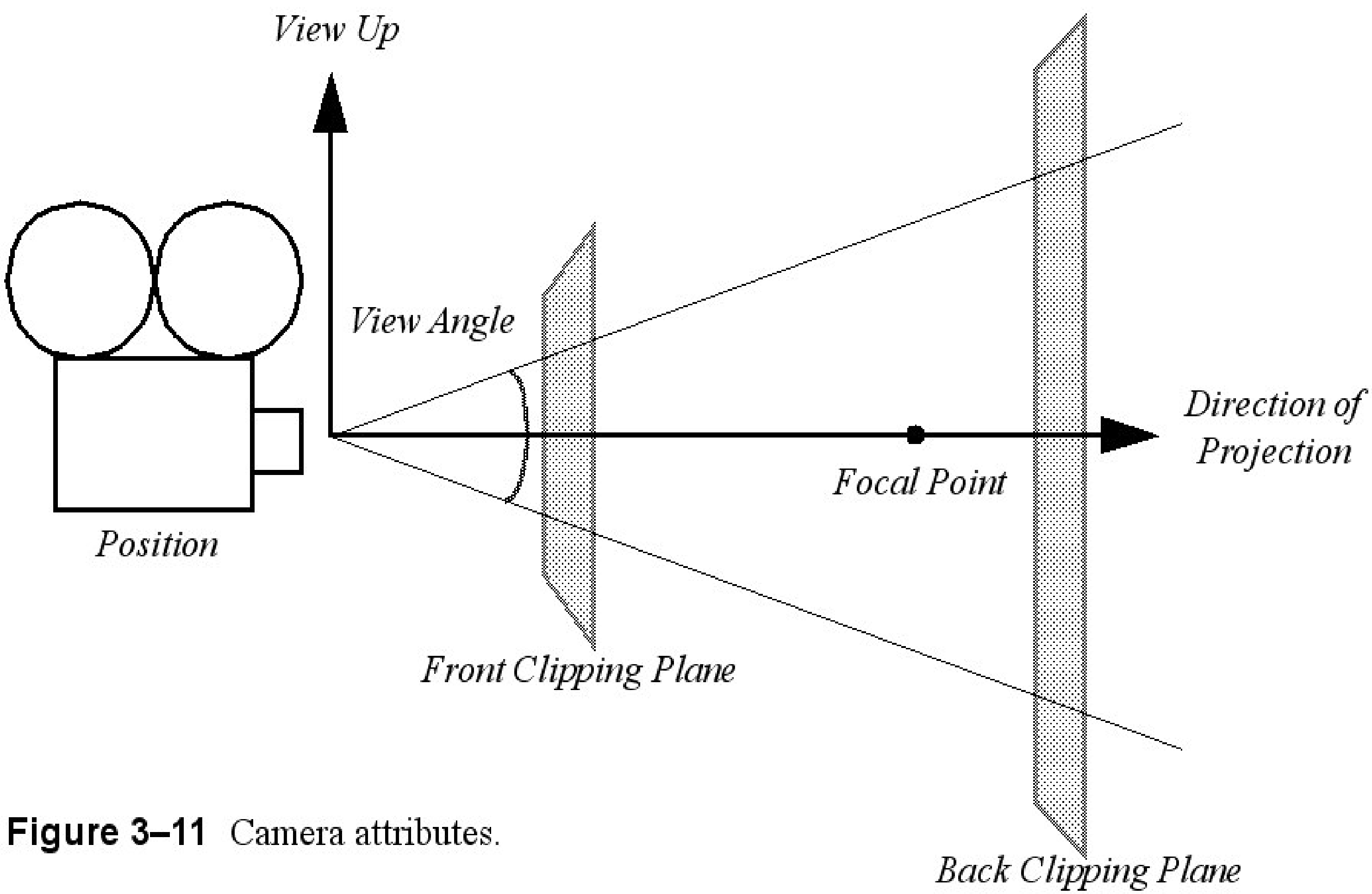
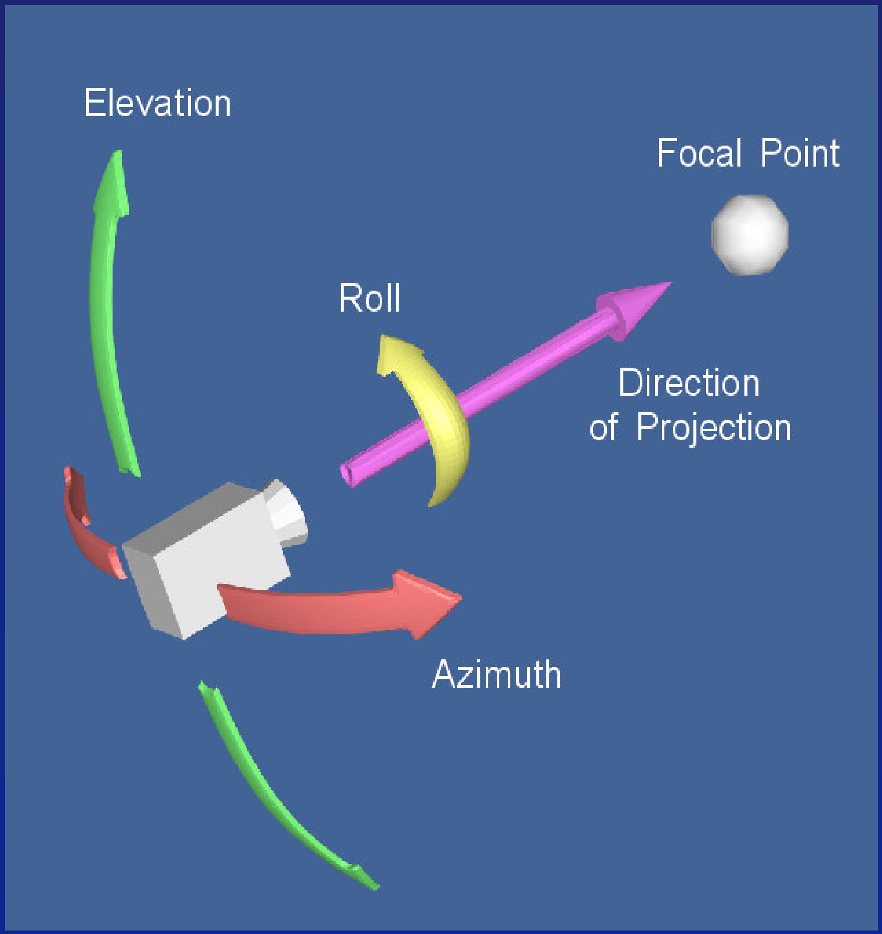


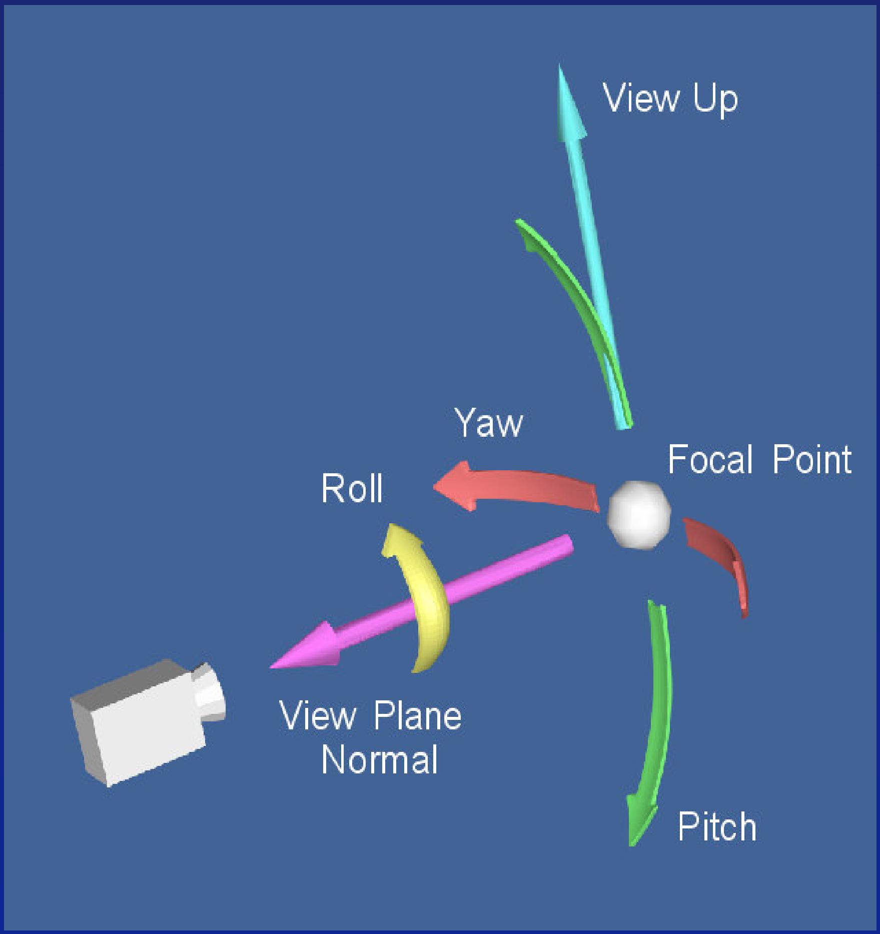
Figure 3–11 Camera attributes.

vtkCamera (cont.)

- ParallelProjection – turn parallel projection on/off (no perspective effects)
- ParallelScale – used to shrink or enlarge an image
- Roll, Pitch, Yaw, Elevation, Azimuth – move the camera in a variety of ways
- Zoom, Dolly – changes view angle (Zoom); move camera closer (Dolly)
- OrthogonalizeViewUp – make the view up vector perpendicular to the view plane normal

VTK's Graphics Subsystem





VTK’s Graphics Subsystem

vtkLight

- Color – the light color
- Position – where the light is
- FocalPoint – where the light is pointing
- Intensity – the brightness of the light
- Switch – turn the light on or off
- Positional – is it an infinite or local (positional) light
- ConeAngle – the cone of rays leaving the light

VTK’s Graphics Subsystem

vtkRenderer

- AddProp (preferred), AddActor, AddVolume, AddActor2D – add objects to be rendered
- AddLight – add a light to illuminate the scene
- SetAmbient – set the intensity of the ambient lighting
- SetViewport – specify where to draw in the render window
- SetActiveCamera – specify the camera to use render the scene
- ResetCamera – reset the camera so that all actors are visible

VTK's Graphics Subsystem

vtkRenderWindow

- AddRenderer() – add another renderer which draws into this vtkRenderWindow
- SetSize() – set the size of the window
- SetPosition() – set the position of the window
- SetWindowName() – set the name (in the titlebar)
- AAFrames, FDFrames, SubFrames – used for anti-aliasing and focal depth
- StereoType, StereoRenderOn/Off – control stereo
- AbortRender, AbortCheckMethod – methods to interrupt the rendering process

vtkRenderWindow (cont.)

- DesiredUpdateRate – a frame rate which is used to control LOD (level-of-detail) actors
- DoubleBuffer – turn double buffering on/off
- PixelData, RGBAPixelData, ZbufferData – set/get the color buffer and depth buffer for the window

VTK's Graphics Subsystem

Example: Initial Camera View

```
vtkCamera *cam1 = vtkCamera::New();
  cam1->SetFocalPoint( 0, 0, 0 );
  cam1->SetPosition( 1, 1, 1 );
  cam1->SetViewUp( 1, 0, 0 );
  cam1->OrthogonalizeViewUp();

ren1->SetActiveCamera( cam1 );
ren1->ResetCamera();
```

VTK's Graphics Subsystem

```
// work the the actor's property. One is created by
// default if a property has not been specified

vtkProperty *prop = actor1->GetProperty();

prop->SetDiffuseColor(0,0,1.0);
prop->SetSpecularColor(0.0,1.0,0.0);
prop->SetSpecular(1);
prop->SetSpecularPower(10);
prop->SetAmbientColor(1,0,0);
prop->SetAmbient(0.3);
```

VTK's Graphics Subsystem

Important vtkProp Subclasses

- vtkLODActor - automated LOD creation
- vtkLODProp3D - manual control of LOD's including mixed volumes/surfaces
- vtkFollower - always face a camera
- vtkAssembly - groups of vtkProp3D's, transformed together.

VTK's Graphics Subsystem

vtkLODActor -- Changes resolution based on desired response

```
vtkLODActor *actor = vtkLODActor::New();
actor->SetMapper( mapper );
actor->SetNumberOfCloudPoints( 1000 );

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->SetDesiredUpdateRate( 5.0 );
```

vtkLODProp3D

- ```
  vtkLODProp3D *lod = vtkLODProp3D::New();
  lod->AddLOD ( volumeMapper, volumeProperty2, 0.0);
  lod->AddLOD ( volumeMapper, volumeProperty, 0.0);
  lod->AddLOD ( probeMapper_hres, probeProperty, 0.0);
  lod->AddLOD ( probeMapper_lres, probeProperty, 0.0);
  lod->AddLOD ( outlineMapper, outlineProperty, 0.0);
  ```
- *From* *`Examples/VolumeRendering/Tcl/volSimpleLOD.tcl`*
  ```

vtkFollower – an actor always faces a specified camera

```
vtkFollower *textActor = vtkFollower::New();
    textActor->SetMapper( textMapper );
    textActor->SetScale( 0.2, 0.2, 0.2 );
    textActor->AddPosition( 0, -0.1, 0 );
    textActor->SetCamera(aCamera);
```

VTK's Graphics Subsystem

vtkAssembly -- Create hierarchies of vtkProp3D's:

```
vtkAssembly *cylinderActor = vtkAssembly::New();
    cylinderActor->AddPart( sphereActor );
    cylinderActor->AddPart( cubeActor );
    cylinderActor->AddPart( coneActor );
    cylinderActor->SetOrigin( 5, 10, 15 );
    cylinderActor->AddPosition( 5, 0, 0 );
    cylinderActor->RotateX( 15 );
```

vtkRenderWindowInteractor -- Key features:

- SetRenderWindow – the single render window to interact with
- Key and mouse bindings (Interactor Style)
- Light Follow Camera (a headlight)
- Picking interaction

Rendering a Polygonal Mesh

```
vtkLookupTable *lut = vtkLookupTable::New();
lut->SetHueRange(0.6, 0);
lut->SetSaturationRange(1.0, 0);
lut->SetValueRange(0.5, 1.0);

vtkDEMReader *demModel = vtkDEMReader::New();
demModel->SetFileName("C:/SainteHelens.dem");
demModel->Update();

double lo = Scale * demModel->GetElevationBounds()[0];
double hi = Scale * demModel->GetElevationBounds()[1];
```

Rendering a Polygonal Mesh

```
vtkImageDataGeometryFilter *geom =
    vtkImageDataGeometryFilter::New();
geom->SetInput(demModel->GetOutput());

vtkWarpScalar *warp = vtkWarpScalar::New();
warp->SetInput(geom->GetOutput());

vtkElevationFilter *elevation = vtkElevationFilter::New();
elevation->SetInput(warp->GetOutput());
elevation->SetScalarRange(lo, hi);

vtkDataSetMapper *dsMapper = vtkDataSetMapper::New();
dsMapper->SetInput(elevation->GetOutput());
dsMapper->SetScalarRange(lo, hi);
dsMapper->SetLookupTable( lut );
```

Rendering an Image

- There are multiple ways to render an image
  - Direct mapping to pixels
  - Texture mapped onto a plane
  - Converted into polygons as in the prior example
  - Use to modify (texture, etc) a different geometry

- Direct mapping to pixels has the advantage of straight forward, advanced interpolation or scaling can be done algorithmically

- Texture mapping leverages graphics hardware to perform interpolation and scaling, this is very fast

- Other approach depend on the specific of the visualization

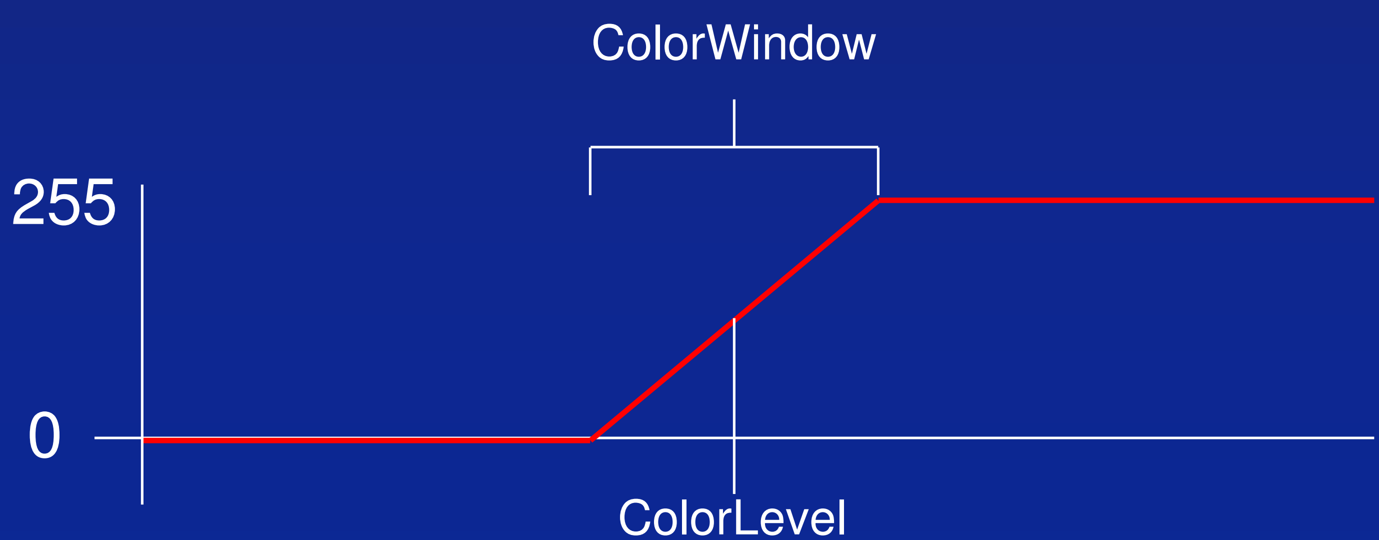
Rendering an Image

- ImageViewer2 - simple one step solution
  - RenderWindow
  - Renderer
  - vtkImageActor
  - vtkImageMapToWindowLevelColors
- vtkImageActor can make use of hardware interpolation and scaling
- Mipmaps, etc, can be used (in hardware) to address aliasing issues

Rendering an Image

Image Display Methods

- SetInput
- Set/GetZSlice
- GetWholeZMin/Max
- SetColorWindow – width that determines which data values are displayed
- SetColorLevel – data value that centers the window



Rendering an Image

- Coordinate Systems

Viewport	Pixels (0 to size – 1)
Normalized Viewport	0, 1
Display	Pixels (0 to size – 1)
Normalized Display	0, 1
View	-1, 1
World	-inf, inf

Texture Mapping

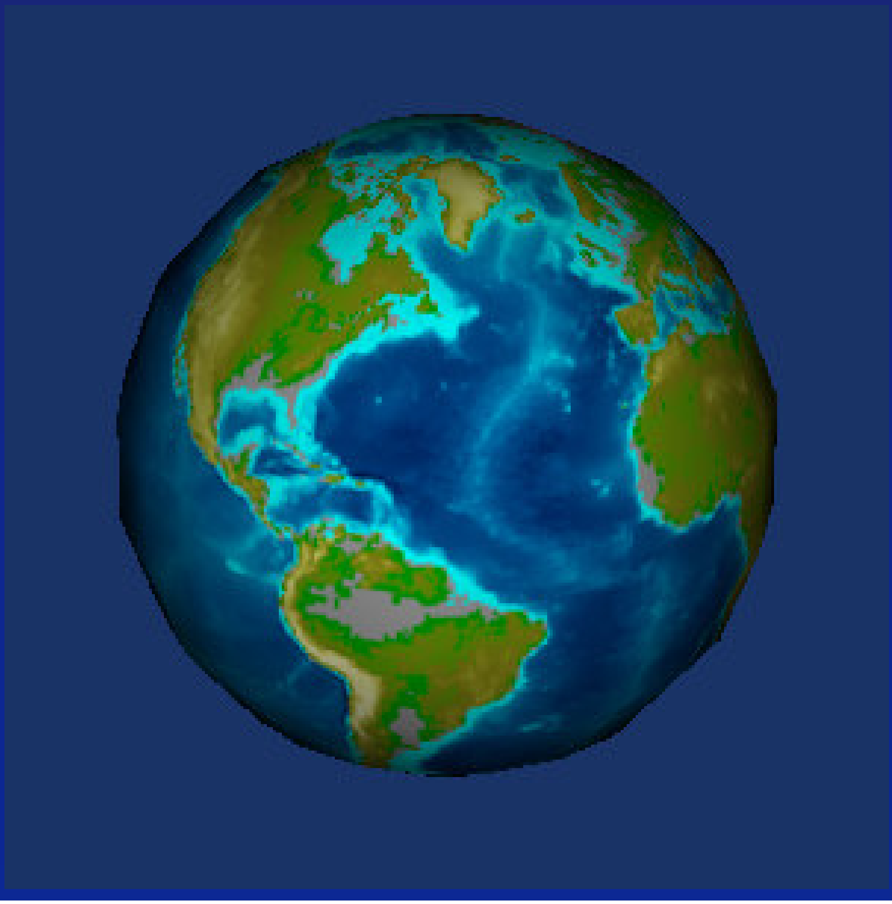
Ken Martin
Kitware Inc.

Texture Mapping

- How to use texture mapping for visualization
- Static texture maps
  - Satellite (or photo etc) imagery mapped onto geometry
  - Texture maps used to illustrate geometry
  - Texture maps used for scalar coloring
  - Texture maps used to modulate a visualization through opacity
- Dynamic texture maps
  - Used in vector field visualization to denote flow direction and velocity
  - Used in 4D visualization to show imagery over time

Texture Mapping

- Static texture maps
  - Satellite (or photo etc) imagery mapped onto geometry



Texture Mapping

- Satellite (or photo etc) imagery mapped onto geometry

```
vtkTexturedSphereSource *tss = vtkTexturedSphereSource::New();
  tss->SetThetaResolution ( 18 );
  tss->SetPhiResolution ( 9 );

vtkPolyDataMapper   *earthMapper = vtkPolyDataMapper::New();
  earthMapper->SetInput ( tss->GetOutput() );

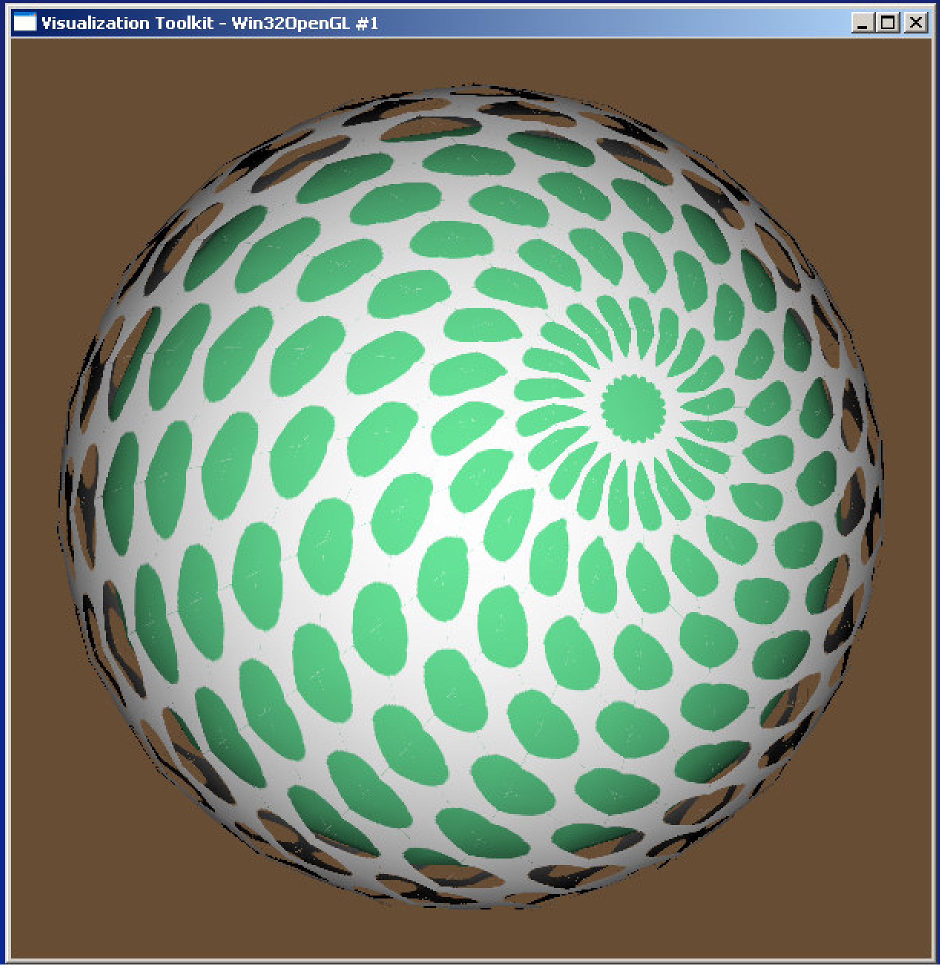
vtkActor *earthActor = vtkActor::New();
  earthActor->SetMapper ( earthMapper );

vtkTexture *atext = vtkTexture::New();
vtkPNMReader *pnmReader = vtkPNMReader::New();
  pnmReader->SetFileName (“C:/Data/earth.ppm“ );

atext->SetInput ( pnmReader->GetOutput() );
atext->InterpolateOn ();
earthActor->SetTexture ( atext );
```

Texture Mapping

- Static texture maps
  - Texture maps used to illustrate geometry



Texture Mapping

Static texture maps - Texture maps used to illustrate geometry

```
vtkTriangularTexture *aTriangularTexture = vtkTriangularTexture::New();
aTriangularTexture->SetTexturePattern( 2 );
aTriangularTexture->SetScaleFactor( 1.3 );

vtkSphereSource *aSphere = vtkSphereSource::New();

vtkTriangularTCoords *tCoords = vtkTriangularTCoords::New();
tCoords->SetInput( aSphere->GetOutput() );

vtkPolyDataMapper *dsMapper = vtkPolyDataMapper::New();
dsMapper->SetInput(tCoords->GetOutput());

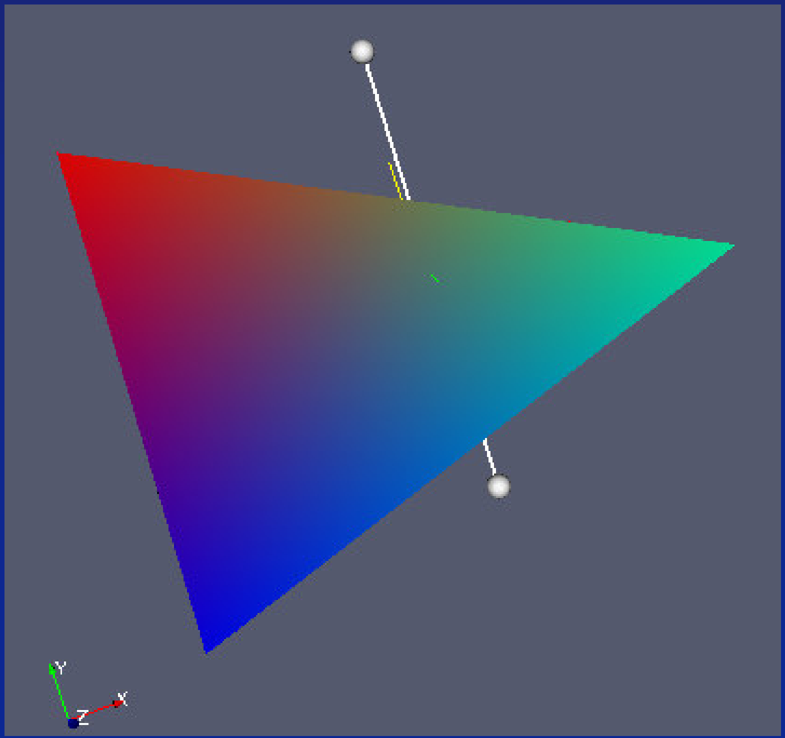
vtkTexture *aTexture = vtkTexture::New();
aTexture->SetInput( aTriangularTexture->GetOutput() );

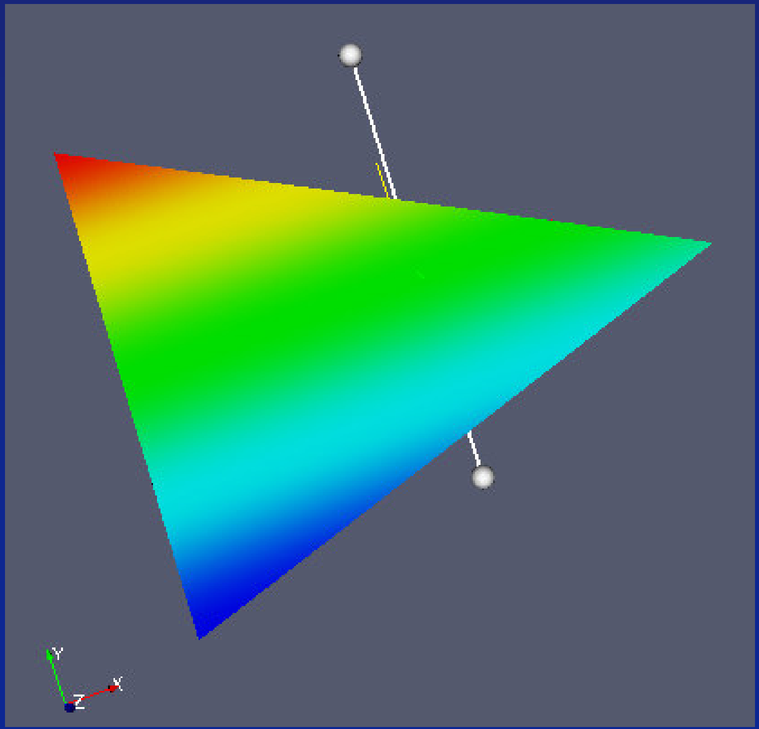
vtkActor *anActor = vtkActor::New();
anActor->SetMapper(dsMapper);
anActor->SetTexture( aTexture );
```

Texture Mapping

Static texture maps -- Texture maps used for scalar coloring

- OpenGL interpolates colors from the vertices, can instead use texture coordinates and then use a texture map to perform per pixel coloring





Texture Mapping

- Static texture maps
  - Texture maps used to modulate a visualization through opacity
- For example, generate texture coordinates based on scalar values (can be 1D or higher)
- then create a RGBA or IA texture map that defines some texture coordinate ranges to be transparent, etc.
- Apply this to any visualization streamlines, isosurfaces of one value textured by another etc.

Texture Mapping

- Dynamic texture maps
  - Used in vector field visualization to denote flow direction and velocity

- Create a series of texture maps that can be cycled
- Create a vector field visualization such as with hedgehogs
- Apply the texture maps to the hedgehogs and then animate through the texture maps