# Lecture 3: CS6250 Graphics & Visualization

- Actors
- Raster devices summary
- Software interface to devices
- Primitives
- Rasterization revisited
- Hidden Surface Algorithms
- VTK overview

# Actors

Modeling

Representing the geometry of physical or virtual objects.

Usually we use mathematical models such as equations of lines, points, polygons, curves, and polynomial curves (splines) of various types.

For visualizations, we want to be able to compute these representations from data. Usually we will stick to simple geometric models, to keep this process as simple as possible.

In addition, we want to take advantage of hardware acceleration wherever possible to keep the visualization interactive.

# Actor Location and Orientation

In VTK, an actor can be assigned a location, and orientation. The position for an actor becomes the origin about which rotations of the actor occur.

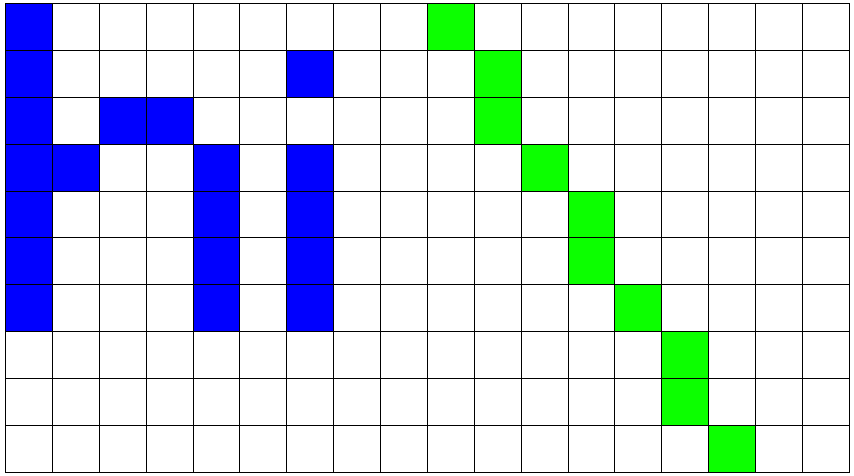
Orientations are specified by a series of rotation matrices in a particular order. First is a rotation about the *y* axis. Next is a rotation about the *x* axis. Finally, a rotation about the *z* axis is performed.

Corresponding to these rotations are three methods `RotateX()`, `RotateY()`, and `RotateZ()`. In addition, there is a method to rotate about an arbitrary axis passing through the origin, `RotateXYZ()`.

# Graphics Hardware

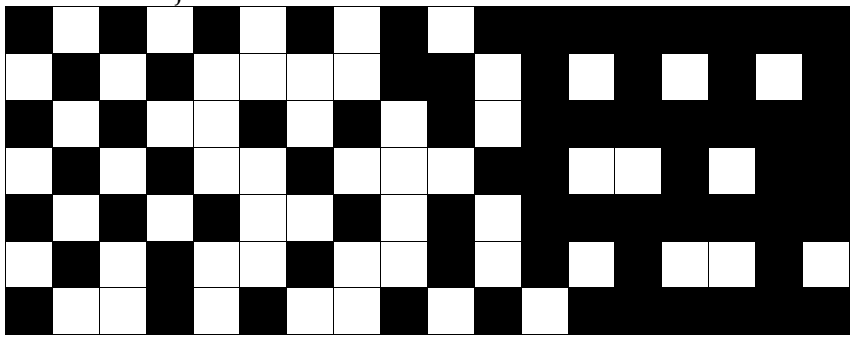
Raster devices



## Dithering

***Dithering*** is a technique that allows you to represent a color or shade with a pattern of colors or shades that are available to a graphics device. For example, a digital half-tone image is created on laser printers to allow them to represent images that are not just black and white.



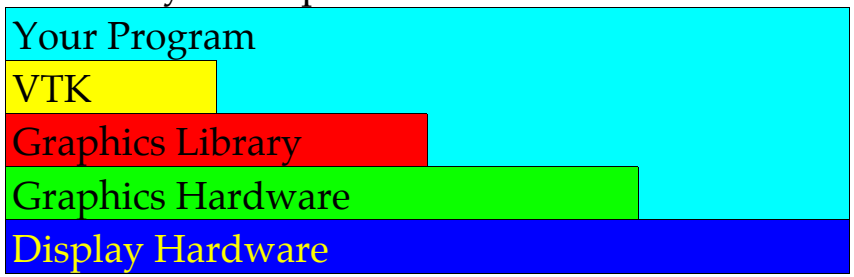
 

## Dithering Continued

Dithering can also be used to display more colors than are available on your output device. A pattern of colors that are available can give rise to the impression of other colors.

## Hierarchy of Graphics Code



## Graphics Primitives

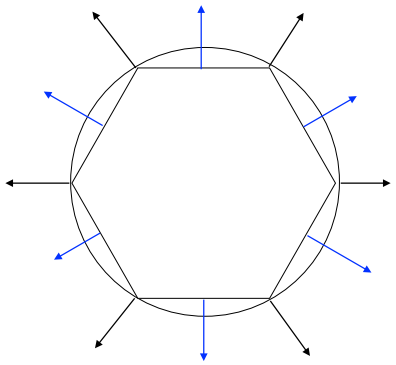
- Point
- Line
- Polyline
- Triangle
- Polygon
- Triangle Strip

What does each primitive need to represent?

Simplest case point?

## Case for Vertex Normals



## Other Styles of Displaying Primitives

- Wireframe
- Surface
- Points
- Depth Cueing
- Edge Color
- …

## Details of the Rasterization Process

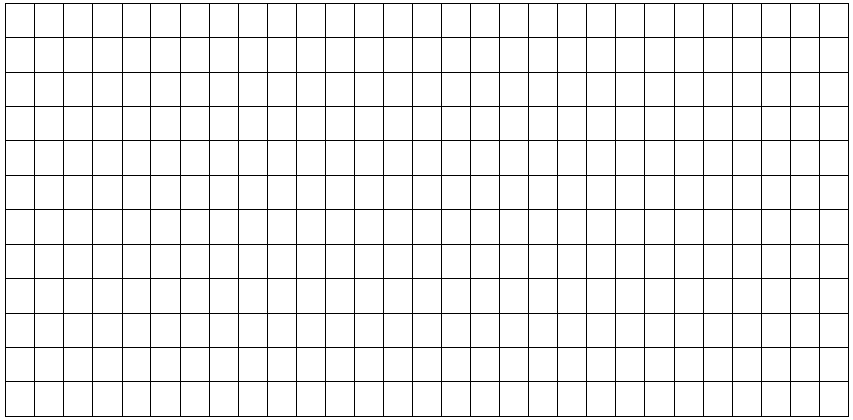
Also called scan-conversion. Let's look at what happens to a typical polygon.

Steps:

- Apply the 3D polygon's transformation matrix to its coordinates.
- Project the polygon to the image plane.
- Clip (clipping can be in both 3D and 2D).
- Identify initial scan-line for polygon. (sort *y*-coords)
  - Compute which line segments are currently "active."
  - For all active segments, interpolate values into interior.

## Rasterization



## Hidden Surface Algorithms

- Painter's Algorithm
- Polygon splitting methods
- Z-buffer
- Ray Tracing

## Z-buffer Issues

- Memory for 1K x 1K need as much as 4 Meg of extra memory
- Precision (is polygon really in front of other?)
- Efficiency

## VTK Objects for Graphics

Here are the basic objects used most frequently in VTK:

- `vtkRenderWindow`
- `vtkRenderer`
- `vtkLight`
- `vtkCamera`
- `vtkActor`
- `vtkProperty`
- `vtkMapper`

## Structure of VTK Classes

`vtkRenderWindow` ties the whole process together, managing a window on whatever machine vtk is running on.

It stores information about the window (size, position, title, etc.). In addition, it stores information related to rendering, e.g. window depth, whether double buffering is on, etc.

Modern graphics cards support 72 or more bits per pixel. 24 for the buffer you see, 24 for the second buffer, and 24 for the z-buffer.

## vtkRenderer

Maintains a list of actors, lights, and cameras for a scene.

A light and a camera will be created automatically if not specifically defined.

Keeps track of the background color, and ambient lighting information.

Draws into a viewport of a `vtkRenderWindow` – this allows more than one renderer to draw into the same `vtkRenderWindow`.

## vtkActor

Contains

- object properties (color, shading type, etc.)
- geometric definition
- orientation and position

These objects are instances of `vtkProperty`, `vtkMapper`, and `vtkTransform`.

## vtkProperty

This class keeps track of the things that can affect the rendered appearance of an object. It is possible for several actors to share the same property.

## vtkRenderWindowInteractor

This class captures events (mouse presses, key presses, etc.) and processes them according to standard conventions to allow interactive viewing of vtk objects.

It allows the you to manipulate your view of a scene.

# Examples

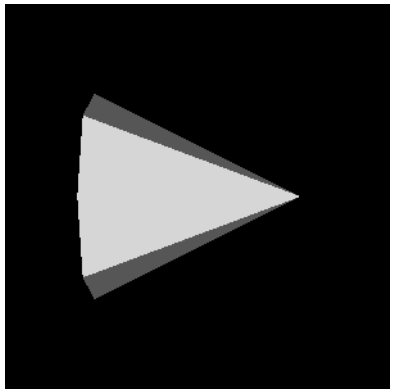
```
#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkConeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"

/* alternately you could #include "vtk.h" */

void main( int argc, char *argv[] )
{
  char a;

  // create a rendering window and renderer
  vtkRenderer *ren = vtkRenderer::New();
  vtkRenderWindow *renWindow = vtkRenderWindow::New();
    renWindow->AddRenderer(ren);
  renWindow->SetSize( 300, 300 );
```

```
  // create an actor and give it cone geometry
  vtkConeSource *cone = vtkConeSource::New();
    cone->SetResolution(8);
  vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    coneMapper->SetInput(cone->GetOutput());
  vtkActor *coneActor = vtkActor::New();
    coneActor->SetMapper(coneMapper);

  // assign our actor to the renderer
  ren->AddActor(coneActor);

  // draw the resulting scene
  renWindow->Render();

// Wait until key is pressed
  cout << "Press any key followed by <Enter> to exit>> ";
  cin >> a;

  // Clean up
  ren->Delete();
  renWindow->Delete();
  cone->Delete();
  coneMapper->Delete();
  coneActor->Delete();
}
```

# Output



# More complex Example

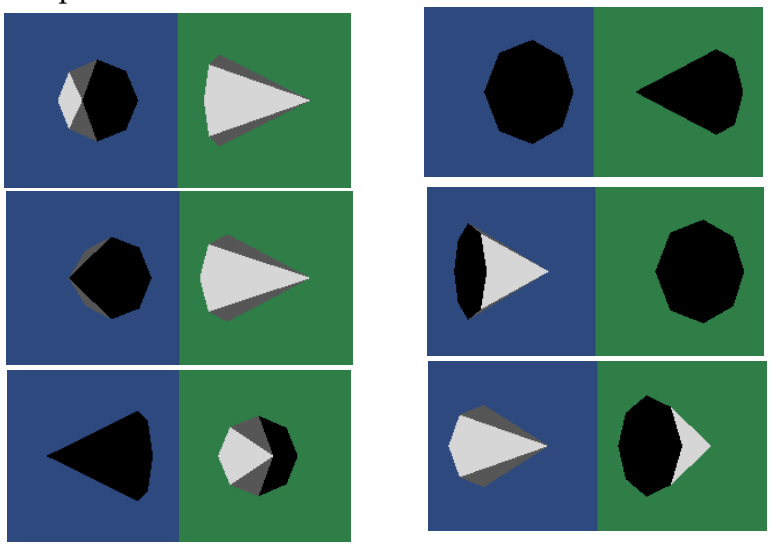
```
// create a rendering window and both renderers
  vtkRenderer *ren1 = vtkRenderer::New();
  vtkRenderWindow *renWindow = vtkRenderWindow::New();
    renWindow->AddRenderer(ren1);
  vtkRenderer *ren2 = vtkRenderer::New();
    renWindow->AddRenderer(ren2);

  // create an actor and give it cone geometry
  vtkConeSource *cone = vtkConeSource::New();
     cone->SetResolution(8);
  vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    coneMapper->SetInput(cone->GetOutput());
  vtkActor *coneActor = vtkActor::New();
    coneActor->SetMapper(coneMapper);

  // assign our actor to both renderers
  ren1->AddActor(coneActor);
  ren2->AddActor(coneActor);

  // set the size of our window
  renWindow->SetSize(300,150);
```

```
// set the viewports and background of the renderers
ren1->SetViewport(0,0,0.5,1);
ren1->SetBackground(0.2,0.3,0.5);
ren2->SetViewport(0.5,0,1,1);
ren2->SetBackground(0.2,0.5,0.3);

// draw the resulting scene
renWindow->Render();

// make one view 90 degrees from other
ren1->GetActiveCamera()->Azimuth(90);

// do a azimuth of the cameras 9 degrees per
// iteration
for (i = 0; i < 360; i += 9)
  {
  ren1->GetActiveCamera()->Azimuth(9);
  ren2->GetActiveCamera()->Azimuth(9);
  renWindow->Render();
  }
```

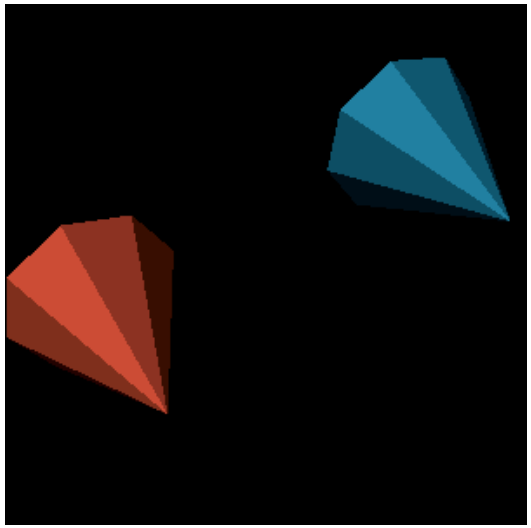



## Output for Cone2:






## Cone4






## Code for Cone4

```
// create a rendering window and renderer
  vtkRenderer *ren = vtkRenderer::New();
  vtkRenderWindow *renWindow = vtkRenderWindow::New();
    renWindow->AddRenderer(ren);
  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWindow);
  renWindow->SetSize( 300, 300 );

  // create an actor and give it cone geometry
  vtkConeSource *cone = vtkConeSource::New();
    cone->SetResolution(8);
  vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    coneMapper->SetInput(cone->GetOutput());
```

```
vtkActor *cone1 = vtkActor::New();
  cone1->SetMapper(coneMapper);
  cone1->GetProperty()->SetColor(0.2000,0.6300,0.7900);
  cone1->GetProperty()->SetDiffuse(0.7);
  cone1->GetProperty()->SetSpecular(0.4);
  cone1->GetProperty()->SetSpecularPower(20);

vtkProperty *prop = vtkProperty::New();
  prop->SetColor(1.0000, 0.3882, 0.2784);
  prop->SetDiffuse(0.7);
  prop->SetSpecular(0.4);
  prop->SetSpecularPower(20);

vtkActor *cone2 = vtkActor::New();
  cone2->SetMapper(coneMapper);
  cone2->SetProperty(prop);
  cone2->SetPosition(0,2,0);

// assign our actors to the renderer
ren->AddActor(cone1);
ren->AddActor(cone2);
```

```
  // draw the resulting scene
  renWindow->Render();

  //  Begin mouse interaction
  iren->Start();

  // Clean up
  ren->Delete();
  renWindow->Delete();
  iren->Delete();
  cone->Delete();
  coneMapper->Delete();
  cone1->Delete();
  prop->Delete();
  cone2->Delete();
}
```