CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 3: Computer Graphics Primer

- Computer graphics is the foundation of data visualization.
- Visualization is the process of transforming data into a set of graphics primitives.
- Computer graphics is the tool to convert graphic primitives into pictures and animations.
- Computer graphics principles
- Hardwares of computer graphics
3.1 Rendering

- Computer graphics is the process of generating images using computers.
- Rendering is the process of converting graphical data into images.
- Data visualization is the process of transforming data into graphics primitives, and rendering the graphics primitives.
Rays of light are emitted from a light source in all directions. Some of these rays happen to strike the cube. The cube surface absorbs some of the incident light and reflects the rest. A part of the reflected light may enter our eyes, and as a consequence, we can see the object. Similarly, some of the light from the sun will reach the ground and a small portion of it will be reflected into our eyes.
Problems of vision?

- The chances of a ray of light traveling from the sun through space to hit a small object on a planet are very low.
- The chances of the ray of lights reflecting off the object and into our eyes are also very low.
- The only reason we can see is that the sun produces such an enormous amount of light that it overpowers the low possibility.
Ray-tracing or ray casting simulates the interaction of light with objects by following the path of each light ray. We follow the ray backwards from the viewer’s eyes and into the world to determine what the ray strikes. The direction of the ray is in the direction we are looking.
Ray-tracing model

- When a ray intersects an object, we can determine if that point is being lit by our light source. This is done by tracing a ray from the point of intersection towards the light.
- If the ray intersects the light, then the point is being lit.
- If the ray intersects something else before it gets to the light, then the light will not illuminate the point.
- For multiple light sources, we repeat this process for each light source.
- The total contributions from all light sources, plus any ambient scattered light, will determine the total lighting or shadow for that point.
- Ray-tracing dramatically reduces the number of rays that must be computed by a simulation program since ray-tracing only looks at rays that end up entering the viewer’s eyes.
Image-order and object-order methods

- Two categories of rendering processes: image-order and object-order.
- Ray-tracing is an image-order process. It works by determining what happens to each ray of light.
- An object-order process works by rendering each object, one at a time.
- In the image-order process, we work in a very orderly fashion, left to right, top to bottom.
- In the object-order process, we work by jumping from one part of the canvas to another, depending on what object we are drawing.
- Ray-tracing is a time-consuming process since it does not require any specialized hardware.
**Image-order Processing:** render each pixel in the image, one scan line at a time and one pixel at a time.

- Example: *Ray-tracing.*
  - For each pixel...
  - A *ray of projection* is defined between the pixel and the viewer.
  - The color of the pixel is determined by the *color of the closest object* intercepting the ray of projection.
**Simple Ray Tracing:**

for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray from eye through the pixel;
        for (each object in the scene) {
            if (object is intersected by the ray and
                is closest considered thus far)
                record intersection point and object id.
        }
        set pixel’s color to that
        at closest object intersection point
    }
}

**Recursive Ray Tracing:**

Simulates the interaction of light with objects by following the path of each light ray.
- Backward: from viewer to the light
- Forward: from light to the viewer
Rendering models

- Two rendering models: surface rendering and volume rendering.
- Surface rendering: render the surfaces of an object.
- Volume rendering: render the surface and interior of an object.
- Volume rendering is useful in analyzing CT and MRI images.
Components of rendering process

- Actors: represent graphical data or objects
- Lights: illuminate the actor
- Camera: a camera constructs a picture by projecting the actors onto a view plane.
- The combination of actors, lights, and camera is called the scene.
3.2 Color theory

- The properties of light: wavelength and intensity.
- The electromagnetic spectrum visible to humans contains the wavelengths ranging from about 400 to 700 nanometers.
- There are three types of color receptors in the human eyes: red, green, and blue.
#### Common colors in RGB

<table>
<thead>
<tr>
<th>Color</th>
<th>RGB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0,0,0</td>
</tr>
<tr>
<td>White</td>
<td>1,1,1</td>
</tr>
<tr>
<td>Red</td>
<td>1,0,0</td>
</tr>
<tr>
<td>Green</td>
<td>0,1,0</td>
</tr>
<tr>
<td>Blue</td>
<td>0,0,1</td>
</tr>
<tr>
<td>Yellow</td>
<td>1,1,0</td>
</tr>
<tr>
<td>Cyan</td>
<td>0,1,1</td>
</tr>
<tr>
<td>Magenta</td>
<td>1,0,1</td>
</tr>
<tr>
<td>Sky blue</td>
<td>(\frac{1}{2}, \frac{1}{2}, 1)</td>
</tr>
</tbody>
</table>
3.2 Lights

Local light

Infinite light
3.4 Surface properties

- As rays of light travel through space, some of them intersect actors. The rays of light interact with the surface of the actor to produce a color.

- The resulting color could be from direct light or ambient light.
Ambient lighting

\[ R_c = L_c \cdot O_c \]

Where \( R_c \) is the resulting intensity curve, \( L_c \) is the intensity curve of the light, and \( O_c \) is the color curve of the object.
Diffuse lighting

\[
\text{Light color} = L_c \\
\cos \theta = (O_n \cdot (-L_n)) \\
\text{Object color} = O_c \\
R_c = L_c O_c [O_n \cdot (-L_n)]
\]

\(R_c\) is the resulting intensity curve, \(L_c\) is the intensity curve for the light, and \(O_c\) is the color curve of the object. \(L_n\) is the incident light vector, and \(O_n\) is the surface normal of the object.
Specular lighting

Light color = \( L_c \)

\[
\cos \theta = (O_n \cdot (-L_n))
\]

Object color = \( O_c \)

\[
R_c = L_c O_c [S \cdot (-C_n)]^{O_{sp}}
\]

\[
S = 2 [O_n \cdot (-L_n)] O_n + L_n
\]

\( R_c \) is the resulting intensity curve, \( L_c \) is the intensity curve for the light, and \( O_c \) is the color curve of the object, \( C_n \) is the direction for the camera and \( S \) is the direction of specular reflection. \( L_n \) is the incident light vector, and \( O_n \) is the surface normal of the object. \( O_{sp} \) is the specular power.
The overall lighting model

\[ R_c = O_{ai} O_{ac} L_c - O_{di} O_{dc} L_c (O_n \cdot L_n) + O_{si} O_{sc} L_c [S \cdot (-C_n)]^{O_{sp}} \]

This is the resulting color of a point on the surface of the object. The constants \( O_{ai}, O_{di}, \text{ and } O_{si} \) control the relative amounts of ambient, diffuse and specular Lighting for an object. The constants \( O_{ac}, O_{dc}, \text{ and } O_{sc} \) specify the colors to be used for each type of lighting. These six constants along with the specular Power are part of the surface material properties.
3.5 Cameras

- View angle
- View up
- Position
- Focal point
- Direction of projection
- Front clipping plane
- Back clipping plane
Camera attributes

- Camera attributes include the position, orientation, and focal point of the camera, the method of camera projection, and the location of the camera clipping planes.

- The position and focal point of the camera define the location of the camera and where it points. The vector defined from the camera position to the focal point is called the direction of projection.

- The camera image plane is located at the focal point and is typically perpendicular to the projection vector.

- The camera orientation is controlled by the position and focal point plus the camera view-up vector.

- The methods of projection include parallel projection and perspective projection.

- The method of projection controls how actors are mapped to the image plane.
Camera attributes

- In parallel projection, all rays of light entering the camera are parallel to the projection vector.
- In perspective projection, all light rays go through a common point, the viewpoint or center of projection.
- The front and back clipping planes intersect and are perpendicular to the projection vector.
- The clipping planes are used to eliminate data either too close to the camera or too far away from the camera.
- Only actors or portions of actors within the clipping planes are visible.
- The locations of the planes are measured from the camera’s position along the direction of projection.
- The front clipping plane is at the minimum range value, and the back clipping plane is at the maximum range value.
Camera movement

1. **Camera moves about Focal Point**…
   - **Azimuth**: rotation about View-up\(_{fp}\)
   - **Roll**: rotation about Dir. Proj\(^n\).
   - **Elevation**: rotation about 
     \[\text{Dir. Proj}\(^n\) \times \text{View-up}\(_{fp}\).\]

2. **Focal Point moves about camera**…
   - **Yaw**: rotates FP about View-up\(_{cam}\)
   - **Pitch**: rotates FP about 
     \[\text{Dir. Proj}\(^n\) \times \text{View-up}\(_{cam}\).\]

3. **Other camera motion**…
   - **Dollying**: translate along Dir. Proj\(^n\).
   - **Zooming**: change to viewing angle.
3.6 Coordinate systems

- There are four coordinate systems in computer graphics, the model coordinate system, the world coordinate system, the view coordinate system, and the display coordinates system.

- The model coordinate system is a local Cartesian coordinate system, in which the model is defined.

- The world coordinate system is the 3D space where the actors are positioned.
Model A

Model B

Actor A’s transform

Actor B’s transform

Model coordinates

World coordinates
- Light positions
- Camera positions
- Actor positions

Camera’s transform

View coordinates

Display coordinates

Displaying images

Display transform

Model, world, view, and display coordinate systems
Coordinate systems

- The world coordinate system is also the system in which the position and orientation of cameras and lights are specified.
- The view coordinate system represents what is visible to the camera. The x, y values specify location in the image plane, while the z coordinate represents the distance, or range, from the camera. The camera's properties are represented by a four-by-four transformation matrix, which is used to convert from world coordinates into view coordinates.
- The display coordinate system uses the same basis as the view coordinate system, but instead of using negative one-to-one as the range, the coordinates are actual x, y pixel locations on the image plane. Factors such as the window's size on the display determine how the view coordinate range (-1, 1) is mapped into pixel locations.
- Similar to the view coordinate system, the z-value in the display coordinate system also represents depth into the window.
3.7 Coordinate transformation

When we create images with computer graphics, we project objects defined in three dimensions onto a two-dimensional image plane. We do this by projection, using homogeneous coordinates.

- A point in 3D is represented by a vector \((x, y, z)\) in Cartesian coordinates. A point in homogeneous coordinates are represented by a four element vector \((x_h, y_h, z_h, w_h)\).
- We use homogeneous coordinates to represent an infinite point by setting \(w_h\) to zero.
- The conversion between Cartesian coordinates and homogeneous coordinates is given by:

\[
\begin{align*}
x &= \frac{x_h}{w_h} \\
y &= \frac{y_h}{w_h} \\
z &= \frac{z_h}{w_h}
\end{align*}
\]
Coordinate transformation

- The transformation from Cartesian coordinates to homogeneous coordinates is implemented by using a $4 \times 4$ transformation matrix.

- Transformation matrices are widely used in computer graphics because we can perform translation, scaling, and rotation of objects by repeated matrix multiplication.
Transformation matrices

$$T_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To translate a point $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ in Cartesian space by the vector $(t_x, t_y, t_z)$
$T_s = \begin{bmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$

Scaling matrix

Change the scale in x-direction
$$T_{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_{R_y} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{R_z} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_R = \begin{bmatrix} \cos \theta_{x'x} & \cos \theta_{x'y} & \cos \theta_{x'z} & 0 \\ \cos \theta_{y'x} & \cos \theta_{y'y} & \cos \theta_{y'z} & 0 \\ \cos \theta_{z'x} & \cos \theta_{z'y} & \cos \theta_{z'z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate an object around x axis by angle $\Theta$  
Rotate an object around y axis by angle $\Theta$  
Rotate an object around z axis by angle $\Theta$  
To transform one coordinate axes x-y-z to another coordinate axes $x'-y'-z'$

**Note:** To rotate around the center of the object $O_c$, we must first translate the object from $O_c$ to the origin, apply rotation, and then translate the object back to $O_c$. 
3.8 Actor geometry

The location of the actor is specified by orientation, position, scaling, and the origin of rotation. The orientation of an actor is determined by rotations stored in an orientation vector \((O_x, O_y, O_z)\). This vector defines a series of rotational transformation matrices.

*The order of transformation is a rotation by \(O_y\) around y axis, then by \(O_x\) around the x axis, and finally by \(O_z\) around the z axis.*

Every actor has its own *transformation matrix* that controls

- orientation
- location
- scaling

in *world space.*
3.9 Graphics hardware

- Hardware issues include:
  - Raster devices
  - Program communication to graphics hardware
  - Widely used algorithms in graphics hardware
Raster devices

- Printed pictures, monitor display, images on TV or in a movie are all raster displays.

- A raster device represents an image using a two dimensional array of picture elements called pixels.

- For black and white display, each pixel stores one bit of information, black or white.

- The display of raster devices is slightly blurred and overlapping.

- The resolution of a raster device is measured by dpi (dots per inch).
A graphics system

Input devices:
- Keyboard
- Mouse
- Tablet

Processor

Memory

Frame buffer

Output devices:
- Video monitor

Video monitor

Electron gun

Vertical deflection

Electron filament

Cathode

Control grid

Focusing system (electron lens)

Phosphor coating

Horizontal deflection

Interior metallic coating

(at high positive voltage)
A simple two-color raster-scan system
**Image storage system (frame buffer, bitmap):**
- Refresh memory arranged as a 2D array; each entry corresponds to a screen pixel
- Each entry is composed of a number of bits; brightness and/or color value of each pixel of the screen is stored in corresponding entry in frame buffer
- Implemented with solid state RAM

**Image display system (video/image controller):**
- Cycle through frame buffer row by row, 30 or 60 times/second
- Memory reference addresses are generated in synchronism with the raster scan; contents of the memory are used to control monitor beam’s intensity
- Changes in frame buffer is done during the 1.3 millisecond flyback (or, vertical retrace) time
- Interlaced raster scan (to produce a picture whose effective refresh rate is closer to 60 than to 30 Hz.

**Image creation system:**
- Scan convert abstract representation of an image into appropriate pixel values in the frame buffer
Shadow mask color monitor

- Phospher dots (red, green, blue) are arranged in triangular pattern called triad (or, pixel)
- Three electron guns are used
- A shadow mask, behind the view surface, is equipped so that each small hole for each triad (holes are aligned so that each electron gun excites its corresponding phospher dot
- Resolution of these monitor is limited (high resolution: triads are on about 0.21 mm centers; home TV: triads are on about 0.60 mm centers)
Display with lookup table (LUT)

Frame buffer

X address
Y address

Scan controller

12 bits

R G B

Lookup table (LUT)

Shadow mask CRT

CRT screen

Note:
(1) Each number stored in frame buffer is an index (address) into a lookup table (color Table or color map)
(2) Lookup table provides significant saving on memory while gives the ability to Change colors from picture to picture
Memory cost of raster devices

$1000 \times 1000 \times 24 \text{ bytes} = 24 \text{ Mbytes}$ (based on 80 pixels/inch)
Liquid-crystal display (LCD)

LCD consists of six layers, and is made of long crystalline molecules arranged in a Spiral fashion.
1. Direction of polarization of polarized light passing through is rotated 90°
2. The crystals line up in the same direction when in an electric field, therefore no polarizing effect.
3. The light passing through the liquid-crystal layer will be absorbed by the rear polarizer, so the viewer sees a dark spot on the display.

To create a dark spot at $(x_1,y_1)$, use **matrix addressing**: applying a negative Voltage $-V$ to the vertical grid wire $x_1$ and a positive voltage $+V$ to the horizontal grid wire $y_1$ to create an electric field at $(x_1,y_1)$.

To display dots at $(x_1,y_1)$ and $(x_2,y_2)$, we cannot simply apply negative voltage to $x_1$ and $x_2$ and positive voltage to $y_1$ and $y_2$: that would cause dots to appear at $(x_1,y_1)$, $(x_1,y_2)$, $(x_2,y_1)$ and $(x_2,y_2)$. We have to activate them one at a time. The display is refreshed one row at a time.
(1) Lights emanated from a fluorescent panel spreads out in waves in all directions.
(2) The polarizing filter lets horizontal light waves pass.
(3) In the liquid-crystal layer, long rod-shaped molecules react to the electrical charges by forming a spiral. The molecules at one end of the cell wind up at an angle of 90 degrees at the other end of the cell.
(4) Polarized light entering the cell from the rear is twisted along the spiral path of the molecules. Lights are twisted to a angle between 0 and 90 degrees.
(5) Light emerged from the liquid-crystal cells passes through color filters.
(6) The second polarizing filter lets vertical light waves pass. Perfect pass, partial pass, entirely blocking.
Interfacing the hardware

Most graphics Programming is done using higher-level primitives than Individual pixels.
Graphics primitives

- Polygon: a set of edges usually in a plane to define a closed region.
- Triangle strip: a series of triangles where each triangle shares its edges with its neighbors.
- Line: connects two points.
- Polyline: a series of connected lines.
- Point: a 3D position in space.
A vertex has a position, normal, and color, each of which is a three element vector. The position specifies where the vertex is located, its normal specifies which direction the vertex is facing, and its color specifies the vertex’s red, green, and blue components.

Why each vertex has a normal instead of having just one normal for the entire polygon? The reason is that sometimes a polygon is used as an approximation of something else, like a curve.
Vertex and polygon normals

[Diagram showing a circle with a polygon inscribed within it, illustrating vertex and polygon normals.]
Rasterization

- Rasterization is the process of converting a geometric representation into a raster image. It is also called scan conversion.
- The actors are represented by polygon primitives. The first step is to transform the polygon using the appropriate transformation matrix. We then need to project the polygon to the image plane using parallel or perspective projection.
- The projected polygon needs to be clipped against the front and back clipping planes, as well as the boundaries of the image plane. Then we do scan-line processing.
Scan-line processing (scan conversion)

1. Sorting the vertex’s y values.
2. Finding the two edges joining the vertex.
3. Start with the first scan line that intersects with the polygon.
4. For each scan line, the data values are interpolated from the edges on either side of the span to compute the internal pixel values.
5. For each scan line color all the pixels that are inside the polygon.
6. Loop until the whole polygon is filled.

\[ d_i = d_{i-1} + \Delta d_i \]
Z-buffer

- We render actors using polygon methods.
- We use z-buffer to determine if a polygon is hidden or not.
- Z-buffering stores the z-value (the depth value along the direction of projection) for each pixel.
- Before a new pixel is drawn, its z-value is compared against the current z-value for that pixel location.
- If the new pixel would be in front of the current pixel, then it is drawn and the z-value for that pixel location is updated. Otherwise the current pixel remains and the new pixel is ignored.
While the front-buffer updates the monitor, clear and render to the back-buffer.

During the vertical retrace, swap the buffers.
- **Rendering** (Shading, 3D): Rasterization (scan-conversion) with lighting.

- **Flat shading**: apply lighting equations to surface normal of the polygon

- **Gouraud shading**:
  1. apply lighting equations to vertices’ normals, then
  2. interpolate the colors to each point in the polygon.

- **Phong shading** (most realistic):
  1. at every pixel on the polygon interpolate the vertices normals, then
  2. apply lighting equations inside the polygon using the interpolated normal
3.10 Client and server

- **Server**: the remote machine supporting client workstations.

- **However, for X window system:**
  - **Server**: the device that displays the graphics (machine in front of the user)
  - **Client**: the device that does computing (whatever machine running the application)
Concept of X server

- Client
- X protocol
- CPU
- Input devices
- Display file
- X protocol
- CRT
- Server
- File server
3.11 VTK-graphics models

- **vtkRenderWindow**: `vtkRenderWindow` objects are used to manage renders and store graphics specific characteristics such as size, position, window title, window depth, and the double buffering flag.

- **vtkRender**: `vtkRender` is responsible for coordinating its lights, camera, and actors to produce an image.

- **vtkLight**: `vtkLight` illuminates the scene.

- **VtkCamera**: `vtkCamera` specifies camera position, focal point, location of front and back clipping planes, view up vector, and field of view.
VTK – graphics models

- **vtkActor**: `vtkActor` represents an object rendered in the scene, both its properties and position in the world coordinate system.

- **vtkProperty**: `vtkProperty` defines the appearance properties of an actor including color, transparency, and lighting properties such as specular and diffuse, also representational properties like wireframe and solid surface.

- **vtkMapper**: `vtkMapper` is the geometric representation of an actor. More than one actors may refer to the same mapper.

- **VtkRenderWindowInteractor**: `vtkRenderWindowInteractor` captures events for a render in the rendering window.
Double buffering

- In **double buffering**, a display window is logically divided into two buffers. At anytime, one buffer is visible to users (front buffer), and the second buffer (back buffer) is used to draw the next image in an animation. Once the rendering is complete, the two buffers can be swapped so that the new image is also visible.

- Double buffering allow animations to be displayed without the user seeing the actual rendering of the primitives.

- High-end graphics systems implement double buffering in hardware.
Double buffering

- A typical system would have a rendering window with a depth of 72 bits. The first 24 bits are used to store the red, green, and blue (RGB) pixel components for the front buffer, the next 24 bits store the RGB values for the back buffer, and the last 24 bits are used as a z-buffer.

- What is the storage requirement for a 1024×1024 display?
Device independence means that computer code that runs on one operating system with a particular software/hardware configuration runs unchanged on a different operating system and software/hardware configuration.

VTK handles this transparently by a combination of inheritance and a technique known as object factories.
Achieving device independence: inheritance

The `vtkActor` class is broken into two parts: a device independent superclass, and a device dependent subclass. The user creates a device dependent Subclass by invoking the special constructor `New()` in the device Independent superclass.
Achieving device independence: object factories

vtkActor *vtkActor::New()
{
    char *temp = vtkRenderWindow::GetRenderLibrary();
    ...
    if( !strcmp("OpenGL", temp) ) return vtkOpenGLActor::New();
    ...
}

If a new graphics library became available, we would only have to create a new device dependent subclass, and then modify the New() method from the device independent superclass to instantiate the subclass based on environment variables or other system information.
Transformation in VTK

\[ T = T_T(p_x + o_x, p_y + o_y, p_z + o_z)T_{R_z}T_{R_x}T_{R_y}T_S T_T(-o_x, -o_y, -o_z) \]

The order of transformation:

- Translate the actor to its origin.
- Scale the geometry.
- Rotate the actor about y, x, and then z axes.
- Do translation to move the actor back to its final location.

Methods: RotateX(), RotateY(), RotateZ();
Origin\((o_x, o_y, o_z)\) specifies the point that is the center of rotation and scaling;
Position\((p_x, p_y, p_z)\) specifies a final translation of the object;
Orientation\((o_x, o_y, o_z)\) defines the rotations about x, y, and z axes;
Scale\((s_x, s_y, s_z)\) defines scale factors for the x, y, and z axes.
Two transformation methods: \texttt{vtkTransform} and \texttt{vtkActor}

\begin{verbatim}
vtkTransform *walk = vtkTransform::New();
walk->RotateY(0,20,0);
walk->Translate(0,0,5);
vtkActor *cow = vtkActor::New();
cow->SetUserMatrix(walk->GetMatrix());
\end{verbatim}

\begin{verbatim}
vtkActor *cow = vtkActor::New();
cow->SetOrigin(0,0,-5);
cow->Rotate(20);
cow->SetPosition(0,0,5);
\end{verbatim}

\[
T = T_R y T_S T_T (0,0,5)
\]

\[
T = T_T (0,0,5 + (-5))T_R y T_S T_T (0,0,(-5))
\]

We always specify transformation in the reverse order of their application!
Implementation in vtk
VTKRenderWindow
specify part of screen where to draw.
manage window in Display Coordinates.

VTKRenderer:
in World Coordinates, manage the scene, light, cameras and actors.

VTKLight: specify a light.

VTKCamera: specify a camera.

VTKActor: define an object.

VTKProperty: define the appearance of an actor.

VTKMapper: define the geometry of an object.
Example Program

**Main()**

- create a window;
- create a renderer; give the renderer to the window;
- create procedural geometry;
- create a mapper; give the geometry to the mapper;
- create an actor; give the mapper to the actor;

  give the actor to the renderer;
  window->render();

```cpp
Example Program

Main() {  
  create a window;  
  create a renderer; give the renderer to the window;  
  create procedural geometry;  
  create a mapper; give the geometry to the mapper;  
  create an actor; give the mapper to the actor;  

  give the actor to the renderer;  
  window->render();  
}
```
User interaction

- vtkRenderWindowInteractor – allow the user to interact with the graphics objects
- Try the following keypresses:
  - w: wireframe mode
  - s: surface mode
  - r: reset the transformation
  - 3: toggle stereo
  - button 3: zoom; button 2: pan; button1: rotate;
  - c/o: camera mode or object mode
  - j/t: joy stick or tracer ball mode
  - e: exit
#include "vtk.h"
Main()
{
    char a;
    //create a rendering window and renderer
    vtkRender *ren = vtkRender::New();
    vtkRenderWindow *renWindow = vtkRenderWindow::New();
    renWindow->AddRenderer(ren);
    //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();
    //loop 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();
}
#include "vtk.h"
Main()
{
    int i;
    //create a rendering window and both renderers
    vtkRender *ren1 = vtkRender::New();
    vtkRenderWindow *renWindow = vtkRenderWindow::New();
    vtkRender *ren2 = vtkRenderer::New();
    renWindow->AddRenderer(ren1);
    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(400,200);
    //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();
    }
    //clean up
    ren->Delete();
    renWindow->Delete();
    cone->Delete();
    coneMapper->Delete();
    coneActor->Delete();
}
#include "vtk.h"
Main()
{
    //create a rendering window and renderer
    vtkRender *ren = vtkRender::New();
    vtkRenderWindow *renWindow = vtkRenderWindow::New();
        renWindow->AddRenderer(ren);
    vtkRenderWindowInteractor *iren =
        vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWindow);
    //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();
    //begin mouse interaction
    iren->Start();
    //clean up
    ren->Delete();
    renWindow->Delete();
    cone->Delete();
    coneMapper->Delete();
    coneActor->Delete();
}
#include "vtk.h"
Main()
{
    //create a rendering window and renderer
    vtkRender *ren = vtkRender::New();
    vtkRenderWindow *renWindow = vtkRenderWindow::New();
    renWindow->AddRenderer(ren);

    //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(0.2,0.63,0.79);
    cone1->GetProperty()->SetDiffuse(0.70);
    cone1->GetProperty()->SetSpecular(0.4);
    cone1->GetProperty()->SetSpecularPower(20);
    vtkProperty *prop = vtkProperty::New();
    prop->SetColor(1.0,0.3882,0.2784);
    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 actor to the renderer
    ren->AddActor(cone1);
    ren->AddActor(cone2);

    //draw the resulting scene
    renWindow->Render();
    //begin mouse interaction
    iren->Start();
}