Chapter 2
Graphics Programming

Rendering Pipeline: Vertex Processor
- Transforms object representation (vertices) from one coordinate system to another
  - World Coordinate System / Frame
  - Camera Coordinate System / Frame
  - Screen Coordinate System / Frame
- Each coordinate frame can be represented as a 4x4 matrix

Rendering Pipeline: Clipping
- [Transformed] vertices are assembled into corresponding geometric primitives
  - Lines, polylines, polygons, ....
- Clipping Volume / View Volume
- Algorithms
  - Line clipping algorithms
  - Polygon clipping algorithms

Rendering Pipeline: Rasterization
- Visible geometric primitives are converted into fragments (“pixel candidates”)
  - Fill polygons, draw lines, ...
- A fragment is a pixel + additional attributes
  - Color, location, depth (distance from image plane), material property, ....

Rendering Pipeline: Fragment Processor
- Assign actual pixels in frame buffer
- Hidden surface removal
- Texture mapping
- Lighting effect
- Blending function (alpha channel)

SGI, GL, and OpenGL
- Silicon Graphics Inc. (SGI) implementation of pipeline in hardware (1982)
- Graphics Library to interact with the hardware
- OpenGL platform independent API (1992)
  - OpenGL Architecture Review Board: SGI, IBM, MS, HP, ....
OpenGL

- Previous Specification: 2.1 (Aug 2006)
  - http://www.opengl.org
- Current Spec: 3.0 (Aug 2008)
  - http://www.khronos.org
- Commands for specifying objects and operations for 3D interactive applications
- Hardware independent

OpenGL as a Finite State Machine

- OpenGL states can be set by user program
- OpenGL system “remembers” these states
  - Current color, current line thickness, current texture, ...
- User program may query these states

OpenGL Libraries

- Core Library (GL)
- OpenGL Utility Library (GLU)
- Bindings for various window system
  - Unix GLX
  - MacOS: AGL
  - Win32: WGL

GLUT

- Additional Libraries used by OpenGL programs
- OpenGL Utility Toolkit (GLUT)
- Commands for
  - Window management
  - User input
  - GUI: button, radio button, ....

Library Organization (in Unix)

OpenGL

GL

GLU

Frame Buffer

X11 Lib

GLX

OpenGL Geometric Primitives

- Used for building 3D object models
- A model is made of its vertices
- OpenGL provides some standard options how vertices can be “connected” together
  - POINTS, LINES, LINE_STRIP, LINE_LOOP
  - POLYGON, QUADS, TRIANGLES, TRIANGLE_STRIP, QUAD_STRIP, TRIANGLE_FAN
glVertex*()

```
void glVertex3d (double x, double y, double z);
void glVertex3f (float x, float y, float z);
void glVertex3i (int x, int y, int z);
void glVertex2f (float x, float y);
void glVertex2iv (int *v);
void glVertex3f (float x, float y, float z);
void glVertex3iv (int *v);
void glVertex4if (int x, int y, int z, int w);
void glVertex4fv (float *v);
void glVertex4iv (int *v);
void glVertex4sv (short *v);
void glVertex4sv (int *v);
void glNormal3fv (float *v);
```

**Isolated Points**

// draw 6 *isolated* points
```
glBegin (GL_POINTS);
glVertex3f (0.0, 0.0, 0.0); // A
glVertex3f (1.0, 0.0, 0.0); // B
glVertex3f (0.0, 1.0, 0.0); // C
glVertex3f (1.0, 1.0, 0.0); // D
glVertex3f (0.0, 0.0, 1.0); // E
glVertex3f (1.0, 0.0, 1.0); // F
```

**Lines**

- Isolated Lines
  - glBegin(GL_LINES)
- Connected Lines
  - glBegin(GL_LINE_STRIP)
  - glEnd

**Polygons**

- Single Polygon: GL_POLYGON
- Connected Polygons
  - Isolated triangles: GL_TRIANGLES
  - Isolated Quads: GL_QUADS
- Strips
  - Triangle Strips (“T-strips”): GL_TRIANGLE_STRIP
  - Triangle Fan: GL_TRIANGLE_FAN
  - Quadrilateral Strips (“Q-Strip”): GL_QUAD_STRIP

**Polygon Front / Back Side?**

- When drawing polygons (GL_POLYGON, GL_QUAD*, GL_TRIANGLE*), vertices must be carefully ordered to maintain Counter Clockwise order
  - Polygon front side = Counter Clockwise order
- Front and back side of a polygon may be rendered differently

**Vertex Order in Q-strip**

- GL_QUAD_STRIP
  - 1st quad: v0, v1, v3, v2
  - 2nd quad: v2, v3, v5, v4
  - 3rd quad: v4, v5, v7, v6
- Even-numbered vertices for “top” edges, odd-numbered vertices for “bottom” edges
  - Illustration (in lecture)
Vertex Order in T-strip
- GL_TRIANGLE_STRIP
  - 1st triangle: v0, v1, v2
  - 2nd triangle: v2, v1, v3,
  - 3rd triangle: v2, v3, v4,
  - 4th triangle: v4, v3, v5
- Even-numbered vertices for “top” edges, odd-numbered vertices for “bottom” edges
- Zigzag trace: v0, v1, v2, v3, v4, ....

Practical Use
- Cylinder from QUAD_STRIP & POLYGON
- Cone from TRIANGLE_FAN
- Sphere from
  - QUAD_STRIP (except at north / south poles)
  - TRIANGLE_FAN (at north & south poles)
- Tetrahedron approximation of sphere (TRIANGLES)

Attributes
- Visual properties that affect how a primitive is rendered
  - Color (of a vertex)
  - Size (of a point / line)
  - Stipple Pattern (lines / polygon fill)
  - Display mode (polygons)
- When a primitive is being rendered the current value of attribute is used

OpenGL Command Name: Number and Suffix
function in the GL lib
number of arguments
// 100% green, 50% blue
gColor3b (0, 127, 64);
command name
type of arg (b=byte)
Gubyte[ ] clr = {0, 127, 64}
gColor3b( ) (clr);

The “b” suffix means that clr is an array of bytes

Specifying Color Values (1)
- Float / Double: must be in the range [0.0, 1.0]
- Unsigned Integer:
  - Largest value of the type is mapped to 1.0
  - Smallest value of the type is mapped to 0.0
    - Unsigned byte: [0-255] => [0.0, 1.0]
    - Unsigned short: [0-65535] => [0.0, 1.0]

Specifying Color Values (2)
- Signed Integer
  - Smallest value of the type is mapped to -1.0
  - Largest value of the type is mapped to +1.0
    - Signed byte: [-128, 127] => [-1.0, +1.0]
    - Signed integer: [-32,768, 32,767] => [-1.0, +1.0]
Specifying RGB Colors

- The following commands are equivalent:
  `glColor3f (0.25, 1.0, 0.5);`
  `glColor3b (32, 127, 64);`
  `glColor3s (8192, 32767, 16384);`
  `glColor3ub (63, 255, 127);`
  `glColor3ub (0x3F, 0xFF, 0x7F);`

RGB + Alpha value

- Colors can be specified in 4-value RGBA system
  - 3 values RGB
  - 1 value for Alpha (transparency)
- Alpha = 0 for fully transparent color
- Alpha = 1 for fully opaque color
  - `glColor4f (1.0, 0.0, 0.5);` // 50% opaque red

glClearColor()

- Specify RGBA value to be used when clearing the frame buffer (the background color)
- `glClearColor (0.0, 1.0, 0.0, 1.0);` use opaque green for clearing the buffer
- More common to use (solid white / black)
  - `glClearColor (1.0, 1.0, 1.0, 1.0)`
  - `glClearColor (0.0, 0.0, 0.0, 1.0)`

Coordinate Frames

- Vertices of objects / models are built in their own object coordinate frame
- OpenGL converts them into camera / eye coordinate frame
  - Clipping is performed in camera coordinate
- Objects and camera are placed within the world coordinate frame
- The final image is rendered on the screen coordinate frame

The Digital Camera Analogy

- Place the objects into the desired arrangement
  - Modeling transformation
- Place the camera into the desired position
  - Viewing transformation
- Zoom the lens (in / out)
  - Projection transformation
- Select image resolution (or print size)
  - Viewport transformation

Series of Coordinate Transformations

- From Object / Model Coordinate (3D) to
  - World Coordinate (3D) to
  - Eye / Camera Coordinate (3D) to
  - Eye / Retina / Film Coordinate (2D) to
  - Screen Coordinate (2D)
Transformations

- **Modeling**: 3D Object Coord => 3D World Coord.
- **Viewing**: 3D World Coord => 3D Eye Coord
- **Clipping/Projection**: 3D Eye Coord => 3D Clip Coord
- **Perspective Div.**: 3D Eye Coord => 2D Eye Coord
- **Window/Viewport**: 2D Eye Coord => Screen

OpenGL Camera

- OpenGL initially places the camera at the world origin pointing in the negative Z axis
- Default viewing volume is a box centered at origin with a side of length 2

Orthographic Viewing / Parallel Projection

- Objects are projected using rays parallel to the z-axis (of the camera)

Camera Coordinate System

- Default OpenGL view is orthographic (parallel) projection
- All parameters of glOrtho(2D) are measured in the camera coordinate system
- gluOrtho2D() uses default values
  - Near Plane: z = -1
  - Far Plane: z = +1
  - Center of camera at z = 0
- An orthographic camera can see objects behind it

Transformation Matrices

- **M** = ModelView Matrix (4x4)
  - a composite transformation that describes the relative placement between the object/model and the camera/viewer
- **P** = Projection Matrix (4x4)
  - transformation from 3D eye coordinate to 3D clip coordinate

ModelView & Projection

- Any vertex $v$ specified by glVertex* ($x_w$, $y_w$, $z_w$) is automatically transformed by the *current* ModelView and Projection matrices (in that order)
  - $v = M v$
    - $v$ is a coordinate as measured in the camera CF
  - $v = P v$
    - $v$ is the clipped coordinate
**glMatrixMode()**

- Choose between manipulating the ModelView or Projection matrix
- Common practice: keep the current matrix mode to ModelView mode!
  - We don’t change camera too often
  - We do frequently move the camera / objects to make an interesting image composition

**Viewport / Aspect Ratio**

- Setting the viewport is like choosing between printing your photo as 4x6, 5x7, panorama, ...
  - Changing viewport does not change the image composition
  - Changing the relative placement between camera and the object does change the image composition
- To avoid image distortion, aspect ratio of Projection should match the aspect ratio of the viewport

**glViewport (x, y, w, h)**

- x, y, w, h are defined in screen coordinate (pixels)
- Define which part of graphics window to be used for rendering image

**Using GLUT (Optional)**

- glutInit()
- glutCreateWindow()
- glutInitDisplayMode()
- glutDisplayFunc()
- glutMainLoop()
- ....
  - When OpenGL window is embedded in a GUI framework (like Qt), these functions are not needed