CS230
Introduction
Intro to OPEN GL

Introduction

Input devices  Output device
Introduction

2D, we will draw directly on the *image plane*

3D, images created through rendering, as if taken by a synthetic camera

Conceptual Model
OpenGL - Software to Hardware

- Silicon Graphics (SGI) revolutionized the graphics workstation by putting graphics pipeline in hardware (1982)
- To use the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

- The success of GL lead to OpenGL (1992), a platform-independent API that was
  - Easy to use
  - Close to the hardware - excellent performance
  - Focus on rendering
  - Omitted windowing and input to avoid window system dependencies
OpenGL Libraries

• OpenGL core library (gl.h)
  - OpenGL on Windows
  - GL on most unix/linux systems

• OpenGL Utility Library -GLU (glu.h)
  - avoids having to rewrite code

• OpenGL Utility Library -GLUT (glut.h)
  - Provides functionality such as:
    • Open a window
    • Get input from mouse and keyboard
    • Menus

Software Organization
Polygons

- Multi-sided planar element composed of edges and vertices.
Storing polygons

- Types in GL for single polygons
  - GL_POLYGON, GL_QUAD, GL_TRIANGLES

- Types in GL for groups of triangles and quads
  - GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN

- Strips and fans reduce the number of vertices specified for the same number of basic elements

Basic transforms in OpenGL

- `glTranslatef(a, b, c);`
- `glTranslated(a, b, c);`

- `glScalef(a, b, c);`
- `glScaled(a, b, c);`

- `glRotatef(angle, x, y, z);`
- `glRotated(angle, x, y, z);`
Material/Shading

- A simple model that can be computed rapidly
- Includes three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source, I
  - To viewer, v
  - Normal, n
  - Perfect reflector, r

Steps in OpenGL shading

1. Enable shading and select model
2. Specify lights
3. Specify material properties
4. Specify normals
Enabling Shading

- Shading calculations are enabled by
  - `glEnable(GL_LIGHTING)`
  - Once lighting is enabled, `glColor()` ignored

- Polygon shading is turned on as:
  - `glShadeModel(GL_SMOOTH)` or
  - `glShadeModel(GL_FLAT)`

- Must enable light sources individually
  - `glEnable(GL_LIGHTi)` i=0,1.....

- Choose lighting parameters

---

Defining a Light Source

- For each light source, we can set an RGB for the
diffuse, specular, and ambient parts, and the
position

```c
GLfloat diffuse0[]={1.0, 0.0, 0.0, 1.0};
GLfloat ambient0[]={1.0, 0.0, 0.0, 1.0};
GLfloat specular0[]={1.0, 0.0, 0.0, 1.0};
GLfloat light0_pos[]={1.0, 2.0, 3.0, 1.0};
```

```c
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
gllightv(GL_LIGHT0, GL_POSITION, light0_pos);
gllightv(GL_LIGHT0, GL_AMBIENT, ambient0);
gllightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
gllightv(GL_LIGHT0, GL_SPECULAR, specular0);
```
Material Properties

- Material properties are also part of the OpenGL state and match the terms in the Phong model
- Set by `glMaterialv()`

```c
GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};

glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialf(GL_FRONT, GL_SPECULAR, specular);
```
Z-buffer Rendering

- Z-buffering is a very common approach, also often accelerated with hardware.
- OpenGL is based on this approach.

OpenGL Orthogonal Viewing

\[ \text{glOrtho}(x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, \text{near}, \text{far}) \]
\[ \text{glOrtho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) \]
OpenGL Perspective

\texttt{glFrustum(xmin, xmax, ymin, ymax, near, far)}

Using Field of View

- With \texttt{glFrustum} it is often difficult to get the desired view
- \texttt{gluPerspective(fovy, aspect, near, far)} often provides a better interface

\[ \text{aspect} = \frac{w}{h} \]
Introduction

2D, we will draw directly on the *image plane*

3D, images created through rendering, as if taken by a synthetic camera

---

Introduction

- Image Plane is a N x M grid of pixels
- This grid is similar to a coordinate system only it is discrete.
Raster Graphics

2D – Area filling primitives

- Polygons
- Image “Textures” - alpha
- Sprites - animation
2D – Area filling primitives

Typeface Text

Bitmap fonts:

Stroke-based:

Stroke-based fonts require “rasterization”

Rasterization

Rasterization is taking the graphic from the continuous (thin) line to turning on pixels

Also known as “Scan conversion”
Rasterization

- Lines, circles, curves, etc...

Circle centered at origin
\[ x^2 + y^2 = r^2 \]
where R is the radius

We can plot this as:
\[ y = \pm \sqrt{r^2 - x^2} \]
Rasterizing lines

0 < \( M < 1 \)  
\[
y_{i+1} = m x_{i+1} + B = m(x_i + \Delta x) + B = y_i + m(\Delta x)
\]

Increment by 1 pixel in X  
“Turn on” (render) \([x_i, \text{Round}(y_i)]\)

...Similar for \( y \) if \( M > 1 \) ...

Rasterizing Lines: the DDA algorithm for \( 0 < M < 1 \)  
(digital differential analyzer)
Fast rasterization

Compute integer coordinates for pixels near the 2D primitives

Algorithms are invoked many, many times and so must be efficient

Output should be visually pleasing, for example, lines should have constant density

Obviously, they should be able to draw all possible 2D primitives, curves, lines and polygons

Fast rasterization

DDA includes round(); and this is fairly slow

For faster rasterization, we want to avoid all slow operations, if possible, only do only integer math +,-

We do this using the Midpoint Algorithm

(In essence, it removes the division associated with the slope by treating it as fraction of two integers)
Fast rasterization

For lines in the first octant, the next pixel is always either to the right or to the right and up one pixel.

By looking at the difference between the line and the midpoint, \( e \), we only need to look at the sign.

\[ \text{i.e.) no} \ \text{round( ) needed} \]

Rasterizing Circles

- What problems arise when we think about scan conversion for circles (i.e. when we increment one pixel in \( x \), say)?
Rasterizing Circles

- What problems arise when we think about scan conversion for circles (ie. when we increment one pixel in x, say)

- Only consider the first octant of the circle of radius r centered at origin

- Start from (r,0) and go until x < y (at 45 degrees)

- The decision is at each step to go up, or go over to the left and up
Rasterizing Circles

• Only consider the first octant of the circle of radius r centered at origin

• Start from (r,0) and go until x < y (at 45 degrees)

• The decision is at each step to go up, or go over to the left and up

Rasterizing Circles

• Better approach takes into account the 8 way symmetry of each octant in a circle

Begin

Plot(x,y);  
Plot(y,x);  
Plot(y,-x);  
Plot(x,-y);  
Plot(-x,-y);  
Plot(-y,-x);  
Plot(-y,x);  
Plot(-x,y)

End;
**Conclusion**

2D, we will draw directly on the *image plane*.

3D, images created through rendering, as if taken by a synthetic camera.