Introduction to OpenGL

OpenGL is a standard library for 2-D & 3-D drawing
- maps fairly directly to graphics hardware
- state-based architecture
- immediate mode drawing
- doesn't address windows or input events (we'll use FLTK)

Why we are using OpenGL
- clean design
- originally by SGI, but enjoys broad cross-platform support
- a free software implementation (www.mesa3d.org)

Decoding OpenGL function names
- general rule: gl*Func*[234][dfis][v]

v = pass pointer rather than values

dimension

type (e.g., i=int)

- example: glVertex3f(x,y,z) — vertex position as 3 floats

Overview of an OpenGL Program

System-dependent initialization
- setup a window on the screen
- bind OpenGL to this window

OpenGL initialization
- setup coordinate system
- setup initial state values

Begin event loop
- wait for system to deliver event (e.g., mouse moved)
- figure out how this will change the scene, if at all
- if redraw event, execute OpenGL commands to draw scene

Event Driven Interactive Programs

This is the model for modern GUI programs
- run some initialization code at startup
- everything after that happens in response to events

Typical events
- window resized
- redraw window
- mouse moved
- button clicked
- key pressed
- …

```
main()
{
  InitializeApp();

  while( !done ) {
    event = WaitForOne();
    handle(event);
  }
}
```

Most frameworks use callbacks
- procedures bound to specific events
- for instance, call function when a button is pressed

OpenGL Uses Immediate Mode Rendering

Execute drawing commands every time window is repainted
- or every time the scene changes (e.g., for animation)
- vs. retained mode where system maintains model of scene

Immediate Mode:
```
void initialize() {}
void on_redraw() {
  for(all triangles t)
  draw_triangle(t);
}
```

Retained Mode:
```
void initialize()
{
  load_triangle_list(T);
}
void on_redraw() {}
```

OpenGL does provide some retained mode features
- display lists — capture & store immediate mode stream
- vertex arrays — pass large array of points in one call

Extensive State Information

Is always a "current" state, initialized with default values

- changes to the state apply to all subsequent operations

```
set_color(Red);
draw_triangle(t1); // red
draw_triangle(t2); // red
draw_triangle(t3); // red
set_color(Blue);
draw_triangle(t4); // blue
```

Examples of state information include

- current drawing color, line width, point size, …
- coordinate system (defined by transformation matrix)
- enabled features: depth tests, alpha blending, lighting, …

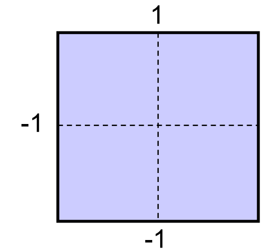
Setting Up the Drawing Canvas

First, we need to define the coordinate system

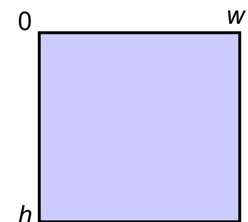
- specifies how to map vertex coordinates onto window
- for now, we're focusing on 2-D — gluOrtho2D(*left*, *right*, *bottom*, *top*)

Example: gluOrtho2D(-1, 1, -1, 1)

OpenGL coordinates



Window Pixel Coordinates



Setting Up the Drawing Canvas

Must always remember to clear canvas before drawing

- glClearColor(r, g, b, α)
  - specify the color to clear the canvas to
  - should generally set α to be 0 (i.e., fully transparent)
  - this is a state variable, and can be done only once
- glClear(GL_COLOR_BUFFER_BIT)
  - actually clears the screen
  - there are other things that can be cleared at the same time
  - such as the depth buffer GL_DEPTH_BUFFER_BIT
  - but we're not using it right now

And always set the current color before you start drawing

- for example, glColor3f(0.8, 0.2, 0.2) for a red shade

Geometric Primitives

Three common modeling primitives

- Points
- Line segments
- Filled polygons

These are very convenient

- the math is simple — just linear equations
- direct hardware support — they're efficient
- universal support — all graphics software understands them

Primitives always bracketed by calls to glBegin() … glEnd()

- argument to glBegin() determines primitive type

Primitive #1: Points

Points are either 2- or 3-dimensional

- By convention, represent them as column vectors

$$v = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad v = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- See `Vec2` and `Vec3` classes in support library

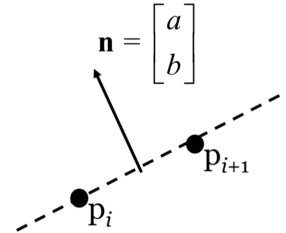
Drawing 2-D Points:

```
Extern Vec2 v[k];
glBegin(GL_POINTS);
  for(int i=0; i<k; i++)
    glVertex2dv(v[i]);
glEnd();
```

Drawing 3-D Points:

```
Extern Vec3 v[k];
glBegin(GL_POINTS);
  for(int i=0; i<k; i++)
    glVertex3dv(v[i]);
glEnd();
```

Primitive #2: Line Segments



Each segment is part of a line

- set of all points satisfying equation

$$ax + by + c = 0 \quad \text{or} \quad \mathbf{n} \bullet \mathbf{p} + c = 0$$

- vector $[a\ b]$ is perpendicular to segment
- it is a normal vector of the segment
- (almost) always want unit normals !

Can also use a vector-valued function:

$$\mathbf{p}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1) \qquad \text{for} \quad 0 \le t \le 1$$

Drawing 2-D Plane Curves

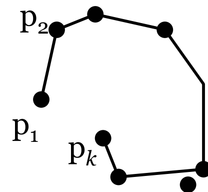
Signal start of primitive with glBegin()

- specifies type of primitive as well
  - GL_LINE_STRIP — open curve
  - GL_LINE_LOOP — closed curve
  - GL_LINES — separate segments

Call glVertex() for each vertex

- specifies position only
- glBegin() type determines how they'll be linked together
- all lines drawn in current color

Call glEnd() after all vertices



Drawing an Open 2-D Curve:

```
extern Vec2 p[k];

glBegin(GL_LINE_STRIP);
  for(int i=0; i<k; i++)
    glVertex2dv(p[i]);
glEnd();
```

Primitive #3: Polygons

A simple polygon

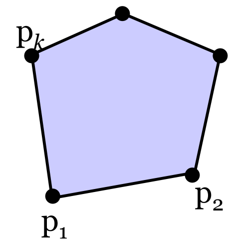
- defined by a sequence of vertices
- edges between consecutive vertices
- no two edges meet except at vertices
  - if this isn't true, polygon is "complex"

Things are a little complicated in 3-D

- all vertices may not lie in same plane

Always maintain consistent ordering

- List vertices (counter-)clockwise



Drawing a 2-D Polygon:

```
extern Vec2 p[k];
glBegin(GL_POLYGON);
  for(int i=0; i<k; i++)
    glVertex2dv(p[i]);
glEnd();
```

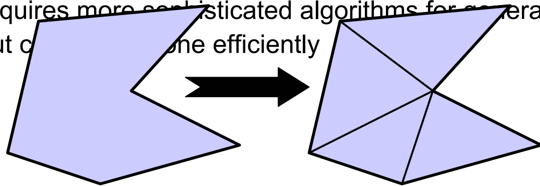
Common Simplification: Triangles

A particular special case of polygon with several advantages

- number of vertices is always fixed
- triangles in 3-D are necessarily planar
- they must also always be convex
- these make it particularly attractive for hardware design

Can convert any planar polygon into exactly equivalent triangles

- trivial if polygon is convex: connect all vertices to a point of interior
- requires more sophisticated algorithms for general polygons
- but c [illegible] ne efficiently



Plane Equations in 3-D

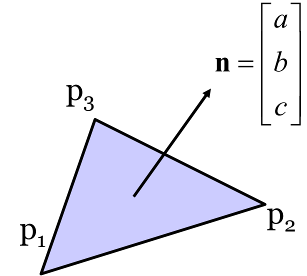
Triangles define a unique plane in 3-D

- set of all points satisfying equation

$$ax + by + cz + d = 0$$

- vector [$a\ b\ c$] is the plane normal
- hence perpendicular to the triangle
- typically use unit normal vector

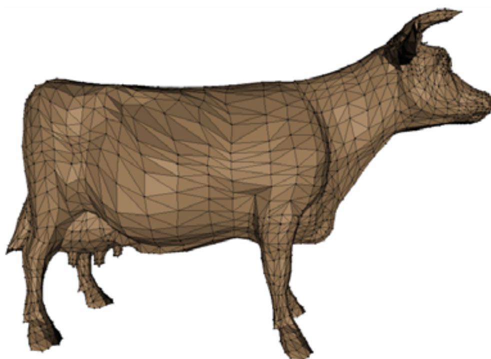
$$a^2 + b^2 + c^2 = 1$$



Normals will show up again and again

- especially in rendering

An Object Modeled with Triangles



Generating Animation

We create animated behavior just like movie projectors

- display a sequence of still images in rapid succession
- creates the illusion of continuous motion
- around 10 Hz is the minimum acceptable rate
- film projectors generally run at 24 Hz
- broadcast and video are usually at 30 Hz

Assuming our redraws are fast, it's easy to program

- set up a system timer to go off every 1/n of a second
- generate a redraw event when the timer goes off

Immersive applications must pay close attention to frame rate

- flight simulators are a good example
- inconsistent frame rates lead to incredible nausea

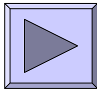
Single vs. Double Buffering

Single Buffering

- draw into same frame buffer being used to generate video signal
- can draw on top of existing image if appropriate
- tends to lead to flickering

Double Buffering

- draw in separate frame buffer
- front buffer — for video signal
- back buffer — drawing target
- must redraw entire scene every frame
- must swap buffers when drawing is finished
- avoids flickering

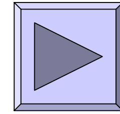


Demo

MP0: Sample Code for Drawing Polygons

You should do the following with this code:

- make sure you can compile it on the CSIL machines
- read and understand the source code
- experiment with modifying it
  - make the polygon purple
  - make it an outline rather than filled



Demo