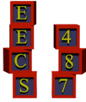



# EECS 487: Interactive Computer Graphics

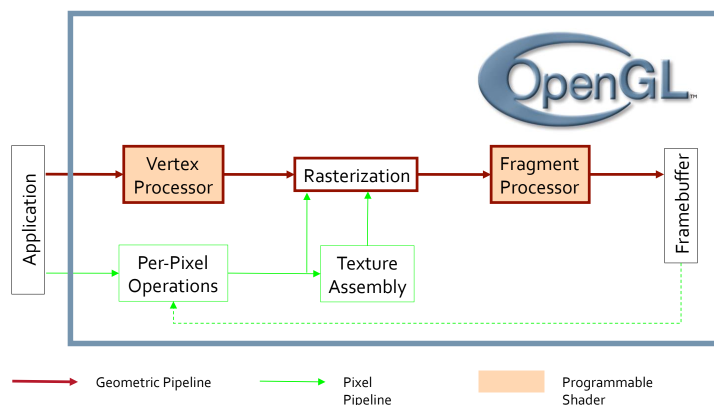
Lecture 3: Introduction to OpenGL and GLUT

## What is OpenGL?

A set of APIs to interface with the graphics hardware

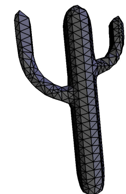
- used to manage the GPU resources to render image
  - sets up the graphics pipeline for rendering
- deals only with low-level geometric primitives: points, lines, polygons
  - no commands for drawing high-level objects, e.g., sphere (GLU)
  - no commands for system, I/O, and windowing or UI tasks (GLUT/GLFW)
- is hardware-independent
- is cross platform, usually bundled with the OS, windowing subsystem, and/or graphics card driver software

## The OpenGL Graphics Pipeline




## What OpenGL Does

Draw primitives into the framebuffer



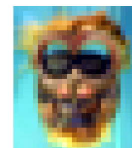
Primitives:

- points
- line segments
- polygons
  - application provides vertex data $(x, y, z)$

} geometric pipeline

- pixel rectangles
- bitmaps
  - application provides pixel data

} pixel pipeline



# OpenGL 2.1 vs. "Modern" OpenGL

We'll learn how to draw using OpenGL 2.1 immediate drawing mode

- deprecated since OpenGL 3.0
- but easier to learn with, similar to using an interpreter vs. a compiler

We'll learn how to use buffers to pass data to the GPU later in the term

# Drawing 101 Immediate Mode

All geometric objects can be represented as a set of vertices in 2D or 3D

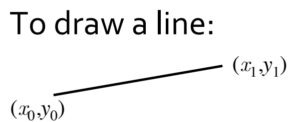
Draw objects by specifying the vertices and how they are to be connected to form primitives:

1. specify how the vertices are to be connected: `glBegin(connection);`
2. specify the vertices: `glVertex*(…);`
3. specify end of primitive: `glEnd();`

# Drawing 101 Examples

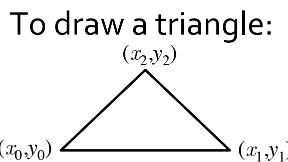
Note the order of vertices!

To draw a line:



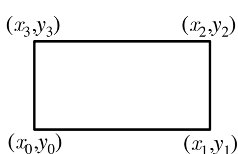

```
glBegin(GL_LINES);
  glVertex2f(x0,y0);
  glVertex2f(x1,y1);
glEnd();
```

To draw a triangle:



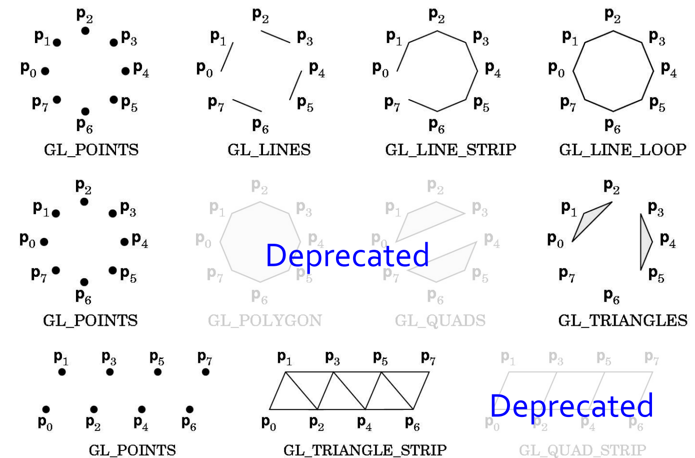

```
glBegin(GL_TRIANGLES);
  glVertex2f(x0,y0);
  glVertex2f(x1,y1);
  glVertex2f(x2,y2);
glEnd();
```

To draw a quadrilateral polygon (quad):




```
glBegin(GL_QUADS);
  glVertex2f(x0,y0);
  glVertex2f(x1,y1);
  glVertex2f(x2,y2);
  glVertex2f(x3,y3);
glEnd();
```

# Connection Types




- note vertex ordering
- GL_POLYGON and GL_QUADS must be simple and convex!

# OpenGL Command Syntax

glVertex3fv( . . . )

| API | base call | argument count | argument data type | | | vector |
|---|---|---|---|---|---|---|
| gl | Color | 2 – $(x, y)$ | b | 8 bit | GLbyte | omit 'v' for scalar form |
| glu | Normal | 3 – $(x, y, z)$ | ub | 8 bit unsigned | GLubyte, GLboolean | glVertex2f(x,y) |
| glut | Flush | 4 – $(x, y, z, w)$ or | s | 16 bit | GLshort | |
| glX | Vertex | 4 – $(r, g, b, a)$ | us | 16 bit unsigned | GLushort | |
| agl | ... | | i | 32 bit | GLint, GLsizei | |
| wgl | | | ui | 32 bit unsigned | GLuint, GLenum, GLbitfield | |
| | | | f | 32 bit | GLfloat, GLclampf | |
| | | | d | 64 bit | GLdouble, GLclampd | |

# Drawing Block

$(x_0,y_0)$ $(x_1,y_1)$

```
glBegin(GL_LINES);
  glVertex2f(x0,y0);
  glVertex2f(x1,y1);
glEnd();
```

Multiple `glBegin() ... glEnd()` blocks allowed

- each block specifies a single type of primitive
- multiple instances of primitive inside each block allowed
- loops, conditions etc. inside each block allowed
- normal C/C++ code and changing attributes like vertex color allowed, but not other OpenGL commands

# Drawing 101

To draw a 3D triangle mesh:

- emit a list of vertices
- every triple makes a face

```
glBegin(GL_TRIANGLES);
  for (int i=0; i < n; i++) {
    glVertex3fv(v[i++]);
    glVertex3fv(v[i++]);
    glVertex3fv(v[i]);
  }
glEnd();
```

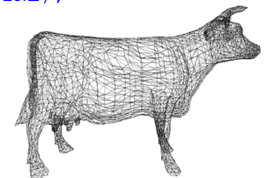
Or more efficiently (fewer calls to `glVertex()`):

- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN

# Drawing Wireframe Meshes

Draw polygon boundary edges only:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_TRIANGLES);
for (int i=0; i < n; i++) {
 glVertex3fv(v[i++]);
 glVertex3fv(v[i++]);
 glVertex3fv(v[i]);
}
glEnd();
```

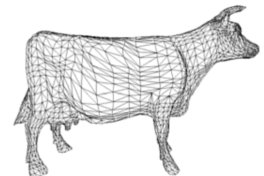


Hidden surface removal:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_FILL);
// fill w/ background color
```

rely on $z$-buffer to remove hidden polygons

- was once an important problem



## Drawing Shaded Polygon

`glVertex*()`

- specifies position only
- drawn in current color

What do these commands draw?

```
glColor3f(0.0,1.0,0.0);
glBegin(GL_TRIANGLES);
 glVertex2f(x0,y0);
 glVertex2f(x1,y1);
 glVertex2f(x2,y2);
glEnd();
```

## Drawing Shaded Polygon

Within a `glBegin()... glEnd()` block

- normal C/C++ code and changing attributes like vertex color allowed, but not other OpenGL commands

What do these commands draw?

```
glColor3f(0.0,1.0,0.0);
glBegin(GL_TRIANGLES);
 glVertex3fv(v[i++]);
 glColor3f(0.0,0.0,1.0);
 glVertex3fv(v[i++]);
 glColor3f(1.0,0.0,0.0);
 glVertex3fv(v[i]);
glEnd();
```

## Setting the Rendering Color

```
void glColor3f(GLfloat red,
               GLfloat green,
               GLfloat blue);
```

`f` for float (single-precision)

Example of other color functions:

```
glColor3d(): rgb-double,
glColor3s(): rgb-short,
glColor4i(): rgba-int
```

```
void glColor3fv(const GLfloat* rgb);
```

`v` for vector: pass values as array

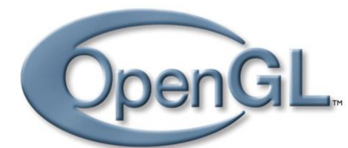
Color value persists until the next `glColor()` call
→ GL state machine

## OpenGL's Utilities

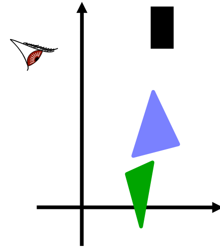
Points, lines, and polygons are such low-level primitives

It would be nice if the graphics API can

- accept some basic higher-level models (GLU)
  (a sphere is a sphere is a sphere, after all)
- handle some basic UI tasks (GLUT)
- but still be:
  - hardware-independent
  - cross platform

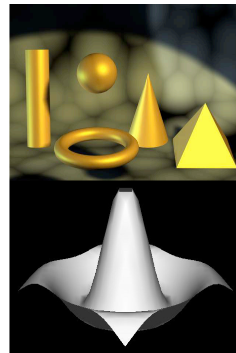


# GLU is …



… a graphics library built on top of OpenGL 2.1 and provides higher-level modelling primitives such as:

- camera and projection controls
- quadrics: sphere, cylinder, disk
- mipmaps
- curves
- surfaces
- tessellation
- NURBS



… deprecated

# Device and Window Abstractions

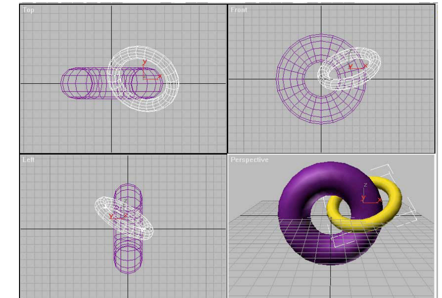
Application must provide a "context" to OpenGL

Device context:

- abstraction of families of output media, e.g., screen, off-line image buffer, printer

GL rendering context:

- window and data structure containing all OpenGL state info
- multiple OpenGL windows need different contexts
  - e.g., different scenes or different views of the same scene



# GL Utility Toolkit

A lightweight library for creating and managing windows and context for OpenGL apps

Provides a single context

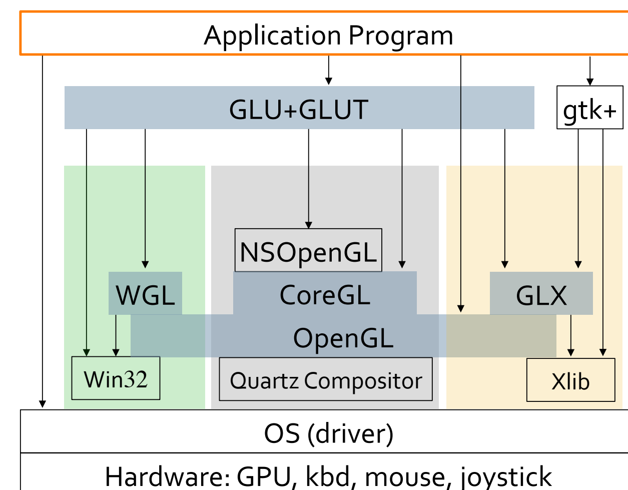
- easy to learn
- but used only for prototyping!

Cross platform

- alternative: GLFW, freeglut (open-source GLUT clone with extensions)
- GUI widget toolkits: Fltk, gtk+, Qt, wxWidgets, etc.

# OpenGL and Related APIs

(simplified view)



# Event-Driven Programming

GLUT programs are event-driven
(most modern GUI programs are)
- example events: window resized, covered, exposed, mouse moved, button clicked, key pressed, etc.

You need to register a callback function (handler) for each event you want to handle on your own

GLUT's main loop runs without an exit
- if an event occur, its handler is called
- upon handler exit, control returns to the main loop

# Typical OpenGL Program Structure

See the *Installing and Using GLUT and OpenGL* course note for sample code (http://www.eecs.umich.edu/~sugih/courses/eecs487/glut-howto/sample.c)

1. Create a window and bind OpenGL to this window
   - OpenGL API calls draw/render within this window
     - what to use to create the window?
     - what to use to interact with the graphics?

2. Register event handlers (call-back functions)

# A Simple OpenGL/GLUT Program

see `sample.c`

```
#include <GL/glut.h>

int main(int argc, char *argv[])
{
  glutInit(&argc, argv);

  /* Create the window first before drawing! */
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
                 /* single buffered, RGBA color (more later) */
  glutInitWindowSize((int)width, (int)height);
  wd = glutCreateWindow("Title"); /* wd is the window handle */

  /* register callback functions/event handlers */
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutIdleFunc(refresh);
  glutKeyboardFunc(kbd);
  glutPassiveMotionFunc(cursor);
  glutMouseFunc(mouse);
  glutMotionFunc(drag);

  glutMainLoop();
  return 0;
}
```

name of callback functions (optional)

# Typical OpenGL Program Structure

3. Set up drawing canvas and coordinate system

4. Prepare the canvas: set up OpenGL states

5. Loop:
   - clear framebuffer
   - perhaps change the screen mapping
   - or change the coordinate system or projection matrix
   - set up lights, camera
   - draw primitives
   - complete drawing

## Setting Up the Drawing Canvas

Define the screen mapping and coordinate system (e.g., in `reshape()`):

- tell OpenGL to use the whole window for drawing:

```
void glViewport(0, 0, window_width, window_height);
```

- set up orthographic projection:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, window_width, 0.0, window_height, -1.f, 1.f);
```

  - we'll see more of this when we discuss transformations; for now, we use a simple 2D orthographic parallel projection, looking straight into the scene

## Setting Up the Drawing Canvas

If you don't plan on changing your coordinate system or perspective often, set up the canvas outside the GLUT main loop

More conveniently, change them every time the window is reshaped

- GLUT calls the `reshape()` function when a window is first created, before the first call to the `display()` function

## Typical OpenGL Program Structure

3. Set up drawing canvas and coordinate system

4. Prepare the canvas: set up OpenGL states

5. Loop:
   - clear framebuffer
   - perhaps change the screen mapping
   - or change the coordinate system or projection matrix
   - set up lights, camera
   - draw primitives
   - complete drawing

## OpenGL State Machine

OpenGL is a state machine

Majority of OpenGL functions do not cause anything to be drawn; instead, they modify OpenGL state

The few calls that actually draw a primitive all use the current state in drawing the primitive

- example draw calls: `glVertex`, `glDrawElements`

Application sets and changes state variables by issuing OpenGL API calls prior to sending down primitives

- the state variables indicate where and how an application wants a primitive to be drawn

# Example State Variables

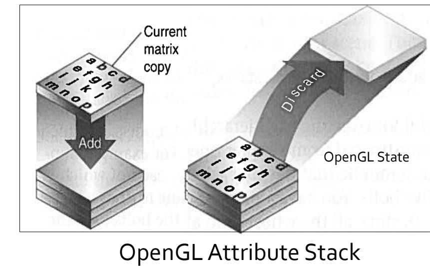
- background color
- vertex color
- polygon drawing mode: points, lines, filled
- camera location, orientation, field of view
- light source: number, color, location
- normal vectors
- material properties
- texture coordinates
- whether to enable depth, transparency, fog
- current viewing and projection transformations (matrices)

# OpenGL Attribute Stack

State variables can be saved on an attribute stack, with elements pushed to and popped from the stack

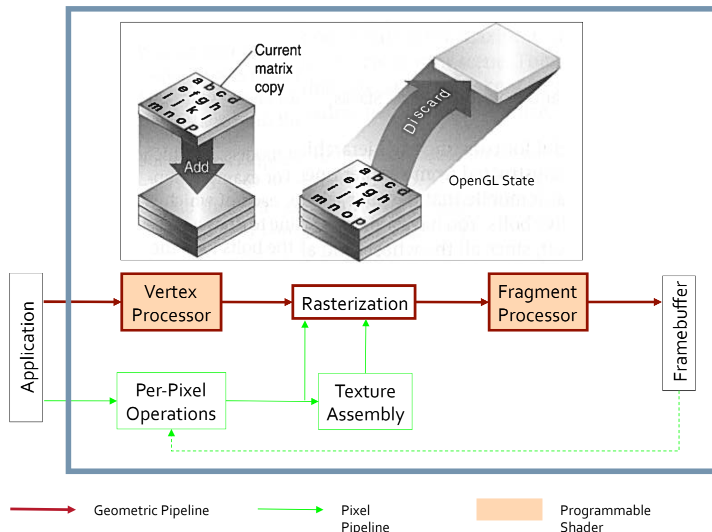
- example stack manipulation calls: `glPushAttrib()`, `glPopAttrib()`

Primitives drawn will reflect the current state (top of the stack)




OpenGL Attribute Stack

# The OpenGL Graphics Pipeline




# Typical OpenGL Program Structure

3. Set up drawing canvas and coordinate system

4. Prepare the canvas: initialize OpenGL state variables on attribute stack

5. Loop:
   - clear framebuffer
   - perhaps change the screen mapping
   - or change the coordinate system or projection matrix
   - set up lights, camera
   - draw primitives
   - complete drawing

# Clear Framebuffer

Always clear the color buffer of the framebuffer before rendering a new frame (what happens otherwise?)

- set clearing color (could be done outside main loop):
  `void glClearColor(red, green, blue, alpha);`
- perform the actual clear operation
  `void glClear(GL_COLOR_BUFFER_BIT);`
- can also clear the depth buffer (`GL_DEPTH_BUFFER_BIT`) and stencil buffer (`GL_STENCIL_BUFFER_BIT`)

# Typical OpenGL Program Structure

3. Sets up drawing canvas and coordinate system
4. Prepare the canvas: sets up state variables on attribute stack
5. Loop:
   - clear framebuffer
   - perhaps change the screen mapping
   - or change the coordinate system or projection matrix
   - set up lights, camera
   - draw primitives
   - complete drawing

# Rendering a Line in OpenGL

All GL rendering happens inside the `display()` handler

For GL window to render a line:

```
void display(void)
{
  /* clear the screen to white */
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);

  /* draw line */
  glBegin(GL_LINES);
    glVertex2f(x0,y0);
    glVertex2f(x1,y1);
  glEnd();
  glFlush(); /* force rendering to start */
  /* or glFinish() which returns only upon
     completion of rendering */
}
```

# Completing the Drawing

Issued GL commands may be stuck in buffers along the pipeline, e.g., waiting for more commands to be issued before sending them in batch

You need to flush all these buffers if you have no more commands to issue, to start rendering

- `void glFlush(void);`
  flushes the buffers and starts execution of commands
- `void glFinish(void);`
  waits for commands to finish executing before returning
- `void glutSwapBuffers(void);`
  if double buffered