OpenGL, GLUT, GLEW, GLSL
An Overview
Objectives

- Give you an overview of the software that you will be using this semester
  - OpenGL, GLUT, GLEW, GLSL
    - What are they?
    - How do you use them?
    - What does the code look like?
    - Evolution
  - All of them are required to write modern graphics code, although alternatives exist
What is OpenGL?

- An application programming interface (API)
- A (low-level) Graphics rendering API
- Generate high-quality images from geometric and image primitives
Maximal Portability

- Display device independent
- Window system independent
- Operating system independent

Without a standard API (such as OpenGL) - impossible to port

Moveto(100, 50)
Lineto(150, 100)
Line(100, 50, 150, 80) - device/lib 1
Line(100, 50, 150, 80) - device/lib 2
Brief History of OpenGL

- Originated from a proprietary API called Iris GL from Silicon Graphics, Inc.
- Provide access to graphics hardware capabilities at the lowest possible level that still provides hardware independence
- The evolution is controlled by OpenGL Architecture Review Board, or ARB.
- OpenGL 1.0 API finalized in 1992, first implementation in 1993
- In 2006, OpenGL ARB became a workgroup of the Khronos Group
- 10+ revisions since 1992
OpenGL Evolution

- 1.1 (1997): vertex arrays and texture objects
- 1.2 (1998): 3D textures
- 1.3 (2001): cubemap textures, compressed textures, multitextures
- 1.4 (2002): mipmap generation, shadow map textures, etc
- 1.5 (2003): vertex buffer object, shadow comparison functions, occlusion queries, non-power-of-2 textures
OpenGL Evolution

- 2.0 (2004): vertex and fragment shading (GLSL 1.1), multiple render targets, etc
- 2.1 (2006): GLSL 1.2, pixel buffer objects, etc
- 3.0 (2008): GLSL 1.3, deprecation model, etc
- 3.1 (2009): GLSL 1.4, texture buffer objects, move much of deprecated functions to ARB compatible extension
- 3.2 (2009)
- 4.3 (August 2012)
OpenGL Extensions

- New features/functions are marked with prefix
- Supported only by one vendor
  - NV_float_buffer (by nvidia)
- Supported by multiple vendors
  - EXT_framebuffer_object
- Reviewed by ARB
  - ARB_depth_texture
- Promoted to standard OpenGL API
Deprecation Model, Contexts, and Profiles

- Redundant and In-efficient functions are deprecated – to be removed in the future
  - glBegin(), glEnd()
- OpenGL Contexts – data structures where OpenGL stores the state information used for rendering
  - Textures, buffer objects, etc
- Profile – A subset of OpenGL functionality used by an application program
  - Core vs. Compatibility
OpenGL Basics

- OpenGL’s primary function – Rendering
- Rendering? – converting geometric/mathematical object descriptions into frame buffer values
- OpenGL can render:
  - Geometric primitives
  - Bitmaps and Images (Raster primitives)
OpenGL Programming Main Steps

- Initialize OpenGL (using GLUT, discussed later)
- Define the geometry (points, lines, triangles/polygons)
- Define the vertex attributes (color, normal, etc)
- Transform the geometry (translate, rotate, scale)
- Set up the camera (position, direction, angle, etc)
- Set up lighting (light position/color etc)
- Set up textures
- Draw
All geometric primitives are specified by vertices

GL_POINTS  GL_LINES

GL_LINE_STRIP  GL_LINE_LOOP

GL_QUADS

GL_QUAD_STRIP  GL_TRIANGLES

GL_TRIANGLE_STRIP
Specifying Geometric primitives

- OpenGL 1.x
  
  ```
  glBegin(primType); // e.g. GL_TRIANGLES
  glVertex3f(x,y,z);
  ...
  glEnd();
  ```

- OpenGL 3.x and up: use Vertex Buffer Objects (VBO)
  
  Much more efficient since it allows the geometry to be stored in the graphics card and reduce the number of function calls

  ```
  glGenBuffers(1, &vboHandle);  // create a VBO handle
  glBindBuffer(GL_ARRAY_BUFFER, vboHandle);  // bind the handle to the current VBO
  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);  // allocate space and copy the data over
  ...
  ```
OpenGL 1.x Code Example

```c
void Display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor4f(1,1,0,1);
  glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
  glEnd();
  glFlush();
}
```

- While `glBegin`/`glEnd` and `glVertex*/glColor*` have been deprecated, it is still important to know them since there is a large amount of legacy code that uses them.
- I will introduce OpenGL VBO in a separate lecture.
**glVertex* Command Formats**

**glVertex2f**

- **glVertex2f(x, y)**

**Number of Components/Dimensions**

- 2 - (x, y)
- 3 - (x, y, z)
- 4 - (x, y, z, w) or (r, g, b, a)

**Add ‘v’ for vector form**

- **glVertex2fv(v)**

**Data Types**

- B - byte
- ub - unsigned byte
- s - short
- us - unsigned short
- i - int
- ui - unsigned int
- f - float
- d - double
After the geometry is provided, OpenGL will control the GPUs to perform vertex and fragment processing.

- Vertex processing: transformation and lighting (T&L)
- Fragment processing: shading and texture mapping

Your job is to control the processing through OpenGL/GLSL API, the focus of this course.

- OpenGL 1.x: use the fixed function pipeline
- OpenGL 3.x and up: provide more detailed control to the shaders
Window-based programming

- Most of the modern graphics systems are window-based

Non-window based environment

Window based environment
Window system independent

- OpenGL is window system independent
  - No window management functions – create windows, resize windows, event handling, etc
  - This is to ensure the application’s portability
  - Create some headache though – just a pure OpenGL program won’t work anywhere.
More APIs are needed

- Microsoft Windows: WGL
- Apple Macintosh: AGL
- X window system: GLX

These libraries provide complete functionality to create the drawing area as well as graphics interface (GUI) such as sliders, buttons, menus etc.

Problem – you need to learn them all to write a platform independent program
Use GLUT (OpenGL Utility Toolkit)

- For fast prototyping, you can use GLUT to interface with different window systems.

- GLUT is a window independent API - programs written using OpenGL and GLUT can be ported to X windows, MS windows, and Macintosh with no effort.

- GLUT does not contain all the bells and whistles though (no sliders, no dialog boxes, no menu bar, etc).
GLUT Basics

Program Structure

1. Configure OpenGL frame buffer
2. Create a drawing area (window)
3. Register input callback functions
   - Render
   - Resize
   - Input: keyboard, mouse, etc
4. Enter event processing loop
Sample Program

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

Void main(int argc, char** argv) {
    int mode = GLUT_RGB|GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```
Sample Program

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

void main(int argc, char** argv) {
    int mode = GLUT_RGB | GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Simple");
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```
Sample Program

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

Void main(int argc, char** argv) {
    int mode = GLUT_RGB|GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow("Simple");
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```

Create a window Named “simple” with resolution 500 x 500
Sample Program

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

Void main(int argc, char** argv) {
    int mode = GLUT_RGB|GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow("Simple");
    glutDisplayFunc(display);
    glutDisplayFunc(display);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```
```c
#include <GL/glut.h>
#include <GL/gl.h>

Void main(int argc, char** argv) {
    int mode = GLUT_RGB|GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow(“Simple”);
    init();
    glutDisplayFunc(display);
    glutMainLoop();
}

Register your call back functions
```
Callback functions?

- Most of window-based programs are event-driven
  - which means do nothing until an event happens, and then execute some pre-defined functions

- Events – key press, mouse button press and release, window resize, etc.
Void main(int argc, char** argv)
{
    ...
    glutDisplayFunc(display);
    ...
}

void display() – a function written by you. It contains all the OpenGL drawing function calls and will be called when pixels in the window need to be refreshed.
Event Queue

GLUT

MainLoop()

Event queue

Mouse_callback()
{
 ... 
}

Keypress_callback()
{
 ... 
}

window_callback()
{
 ... 
}

Keyboard
...

Mouse

Window
And many more …

- **glutKeyboardFunc()** – register the callback that will be called when a key is pressed
- **glutMouseFunc()** – register the callback that will be called when a mouse button is pressed
- **glutMotionFunc()** – register the callback that will be called when the mouse is in motion while a button is pressed
- **glutIdleFunc()** – register the callback that will be called when nothing is going on (no event)
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_SINGLE;
grutInitDisplayMode(mode);
grutInitWindowSize(500,500);
grutCreateWindow("Simple");
grutDisplayFunc(display);
grutDisplayFunc(display);
grutReshapeFunc(resize);
grutKeyboardFunc(key);
glutMainLoop();
}

The program goes into a infinite loop waiting for events
GLEW: The OpenGL Extension Wrangler Library

- A cross-platform open-source C/C++ extension loading library (windows, OS X, Linux, FreeBSD)

- Why GLEW is needed?
  - It is not possible to link directly to functions that are provided in newer version of OpenGL. In windows, this means OpenGL 1.2 and up
  - GLEW does the tedious work to helps you find the function pointers (addresses of the functions) for OpenGL extensions
  - GLEW can also help you to check if a particular OpenGL extension is available on your machine
GLEW usage

- Make sure you have GLEW on your machine
- Include the glew header file
  ```
  #include <GL/glew.h>
  ```
- Initialize glew before calling any opengl functions
  ```
  GLenum err = glewInit();
  if ( err != GLEW_OK) printf(" Error initializing GLEW! 
");
  else printf("Initializing GLEW succeeded!
");
  ```
- Check OpenGL features, for example, shaders
  ```
  if (! GLEW_ARB_vertex_program)
      printf(" ARB vertex program is not supported!!
");
  else printf(" ARB vertex program is supported!!
");
  ```
OpenGL Shading Language (GLSL)

- A C-like language and incorporated into OpenGL 2.0
- Used to write vertex program and fragment program
- No distinction in the syntax between a vertex program and a fragment program
- Platform independent compared to Cg
Shader Objects

- Shaders are defined as an array of strings
- Four steps to using a shader
  - Send shader source to OpenGL
  - Compile the shader
  - Create an executable (i.e., link compiled shaders together)
  - Install the executable as part of current state
- Goal was to mimic C/C++ source code development model
Sequence
GPU = \textbf{vertex shader} (vertex program) + \textbf{fragment shader} (fragment program, pixel program)

- Vertex shader replaces per-vertex transform & lighting
- Fragment shader replaces texture stages
- Fragment testing after the fragment shader
- Flexibility to do framebuffer pixel blending
GPU programming model

“Stream programming”

- Process each vertex or fragment independently

\[ V_1 \to \text{Vertex Shader} \to V_n \]

\[ f_1 \to \text{Fragment Shader} \to f_n \]
The idea

- You specify vertices as usual
  - Vertex positions, texture coordinates, etc.
  - And some user variables if you want
- The vertex shader modifies/calculates these variables.
- Each fragment gets the interpolated values, which might have been modified.
- The fragment shader can now work on the interpolated values, including the user defined variables.
Vertex Program

- Replace the fixed-function operations performed by the vertex processor
- A vertex program is executed on each vertex
- Each vertex program must output the information that the rasterizer needs
  - At a minimum - transforms the vertex position
- The program can access all OpenGL states
  - Current color, texture coordinates, material properties, transformation matrices, etc
- The application can also supply additional input variables to the vertex program
void main(void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}

- Just a passing-through shader: convert a vertex from local space to clip space
- No color is assigned here, so the fragment program will need to decide the fragment colors
- All variables starts with ‘gl_’ are part of OpenGL state so no need to declare
GLSL Data Types

- Supported data types are very similar to C/C++: float, int, bool, etc
- Additional types examples: vec2, vec3, vec4, mat2, mat3, mat4
- Can use C++ style constructor
  - vec3 a = vec3(1.0, 2.1, -1.2);
  - vec3 b = vec2(a); //conversion
GLSL Qualifiers

Three types of variables: *Attributes*, *Uniform*, *Varying*

- **Attribute**: used by vertex shaders for variables that can change once per vertex
  - Build-in attributes: `gl_Vertex`, `gl_FrontColor`
  - User-defined attributes (example): temperature, velocity

- **Uniform**: variables set for the entire primitive, i.e., assigned outside `glBegin()`/`glEnd()`;
  - Also include build-in and user-defined
GLSL Qualifiers (cont’d)

- **Varying variables**: the mechanism for conveying data from a vertex program to a fragment program.
  - Defined on a per vertex basis but interpolated over the primitive for the fragment program.
  - Include build-in and user defined varying variables.
Vertex Processor Input

- Vertex shader is executed once each time a vertex position is specified
  - Via glVertex or glDrawArrays or other vertex array calls
- Per-vertex input values are called “attributes”
  - Change every vertex
  - Passed through normal OpenGL mechanisms (per-vertex API or vertex arrays)
- More persistent input values are called “uniforms”
  - Can come from OpenGL state or from the application
  - Constant across at least one primitive, typically constant for many primitives
  - Passed through new OpenGL API calls
Vertex Processor Output

- Vertex shader uses input values to compute output values
- Vertex shader **must** compute `gl_Position`
  - Mandatory, needed by the rasterizer
  - Can use built-in function `ftransform()` to get invariance with fixed functionality
Vertex Processor Output

- Other output values are called “varying” variables
  - E.g., color, texture coordinates, arbitrary data
  - Will be interpolated in a perspective-correct fashion across the primitives
  - Defined by the vertex shader
  - Can be of type float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays of these

- Output of vertex processor feeds into OpenGL fixed functionality
  - If a fragment shader is active, output of vertex shader must match input of fragment shader
  - If no fragment shader is active, output of vertex shader must match the needs of fixed functionality fragment processing
Fragment Program

- The fragment program is executed after rasterizer and operate on each fragment.
- Vertex attributes (colors, positions, texture coordinates, etc) are interpolated across a primitive automatically as the input to the fragment program.
- Fragment program can access OpenGL state, (interpolated) output from vertex program, and user defined variables.
A very simple fragment program

```c
void main(void)
{
    gl_FragColor = gl_FrontColor;
}
```

- Just a passing-through fragment shader
**Fragment Program Input**

- Output of vertex shader is the input to the fragment shader
  - Compatibility is checked when linking occurs
  - Compatibility between the two is based on *varying variables* that are defined in both shaders and that match in type and name
- Fragment shader is executed for each fragment produced by rasterization
- For each fragment, fragment shader has access to the interpolated value for each *varying* variable
  - Color, normal, texture coordinates, arbitrary values
Fragment Processor Input

- Fragment shader may access:
  - gl_FrontFacing - contains “facingness” of primitive that produced the fragment
  - gl_FragCoord - contains computed window relative coordinates x, y, z, 1/w

- Uniform variables are also available
  - OpenGL state or supplied by the application, same as for vertex shader

- If no vertex shader is active, fragment shader get the results of OpenGL fixed functionality
Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables:

- `gl_FragColor` – computed R, G, B, A for the fragment
- `gl_FragDepth` – computed depth value for the fragment
- `gl_FragData[n]` – arbitrary data per fragment, stored in multiple render targets
- Values are destined for writing into the frame buffer if all back end tests (stencil, depth etc.) pass

Clamping or format conversion to the target buffer is done automatically outside of the fragment shader.