Intro to OpenGL

Don Fussell
Computer Science Department
The University of Texas at Austin
Synthetic Camera Model

projector

image plane

projection of $\mathbf{p}$

center of projection

$p$
Use similar triangles to find perspective projection of point at (x,y,z)

\[ x_p = -\frac{x}{z/d} \quad y_p = -\frac{y}{z/d} \quad z_p = d \]
Programmers want to render “objects”
- Say a fire truck or molecule
- Arranged relative to other objects (a scene) & then viewed

Graphics pipeline approach—used by OpenGL and GPUs
- Break objects into geometry batches
  - Batches may be meshes or “patches”
- Batches reduce to polygonal primitives
  - Typically triangles
  - But also lines, points, bitmaps, or images
- Geometric primitives are specified by vertices
  - So vertices are assembled into primitives
- Primitives are rasterized into fragments
- Fragments are shaded
- Raster operations take shaded fragments and update the framebuffer
Advantages

- Separation of objects, viewer, light sources
- Two-dimensional graphics is a special case of three-dimensional graphics
- Leads to simple software API
  - Specify objects, lights, camera, attributes
  - Let implementation determine image
- Leads to fast hardware implementation
What is OpenGL?

- The OpenGL **Graphics System**
  - Not just for 3D graphics; imaging too
  - “GL” standard for “Graphics Library”
  - “Open” means industry standard meant for broad adoption with liberal licensing
- Standardized in 1992
  - By Silicon Graphics
  - And others: Compaq, DEC, Intel, IBM, Microsoft
  - Originally meant for Unix and Windows workstations
- Now *de facto* graphics acceleration standard
  - Now managed by the Khronos industry consortium
  - Available everywhere, from supercomputers to cell phones
  - Alternative: Direct3D provides similar functionality with a very different API for Microsoft Windows platforms
Student’s View of OpenGL

- You can learn OpenGL gradually
  - Lots of its can be ignored for now
  - The “classic” API is particularly nice
    - “Deprecation” has ruined the pedagogical niceness of OpenGL; ignore deprecation
- Plenty of documentation and sample code
- Makes concrete the abstract graphics pipeline for rasterization
OpenGL API Example

```c
// smooth color interpolation
glShadeModel(GL_SMOOTH);

// enable hidden surface removal
glEnable(GL_DEPTH_TEST);

// clear color and depth buffers
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

// draw a triangle
glBegin(GL_TRIANGLES);

// vertex 1
glColor4ub(255, 0, 0, 255); // RGBA=(1,0,0,100%)
glVertex3f(-0.8, 0.8, 0.3); // XYZ=(-8/10,8/10,3/10)

// vertex 2
glColor4ub(0, 255, 0, 255); // RGBA=(0,1,0,100%)
glVertex3f(0.8, 0.8, -0.2); // XYZ=(8/10,8/10,-2/10)

// vertex 3
glColor4ub(0, 0, 255, 255); // RGBA=(0,0,1,100%)
glVertex3f(0.0, -0.8, -0.2); // XYZ=(0,-8/10,-2/10)

glEnd();
```
Think of drawing into a $[-1,+1]^3$ cube

(origin at (0,0))
What does this simple triangle look like with the $[-1,+1]^3$ cube’ s coordinate system?

- We call this coordinate system “Normalize Device Coordinate” or NDC space

From NDC views, you can see triangle isn’ t “flat” in the Z direction

Two vertices have Z of -0.2—third has Z of 0.3
GLUT API Example

```
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
    // << insert code on prior slide here >>
    glutSwapBuffers();
}

void main(int argc, char **argv) {
    // request double-buffered color window with depth buffer
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInit(&argc, argv);
    glutCreateWindow("simple triangle");
    glutDisplayFunc(display);  // function to render window
    glutMainLoop();
}
```
Simplified Graphics Pipeline

Application → Vertex batching & assembly → Clipping → NDC to window space → Rasterization → Fragment shading → Depth testing → Color update → Depth buffer → Framebuffer

OpenGL API

NDC = Normalized Device Coordinates, this is a $[-1, +1]^3$ cube

Really lots more steps than this but these are the non-trivial operations in our simple triangle example
Application

- What’s the app do?
  - Running on the CPU
- Initializes app process
  - Creates graphics resources such as
    - OpenGL context
    - Windows
- Handles events
  - Input events, resize windows, etc.
  - Crucial event for graphics: **Redisplay**
    - Window needs to be drawn — so do it
    - GPU gets involved at this point

```
Application

Vertex batching & assembly

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing

Color update

Depth buffer

Framebuffer
```
App Stuff

- GLUT is doing the heavy lifting
  - Talking to Win32, Cocoa, or Xlib for you
  - Other alternatives: SDL, etc.

```c
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
    // << insert code on prior slide here >>
    glutSwapBuffers();
}

void main(int argc, char **argv) {
    // request double-buffered color window with depth buffer
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInit(&argc, argv);
    glutCreateWindow("simple triangle");
    glutDisplayFunc(display);  // function to render window
    glutMainLoop();
}
```

display function is being registered as a "callback"
Rendering - the *display* Callback

```c
glShadeModel(GL_SMOOTH); // smooth color interpolation
glEnable(GL_DEPTH_TEST); // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

glBegin(GL_TRIANGLES); {
   // every 3 vertexes makes a triangle
   glColor4ub(255, 0, 0, 255); // RGBA=(1,0,0,100%)
   glVertex3f(-0.8, 0.8, 0.3); // XYZ=(-8/10,8/10,3/10)
   glColor4ub(0, 255, 0, 255); // RGBA=(0,1,0,100%)
   glVertex3f(0.8, 0.8, -0.2); // XYZ=(8/10,8/10,-2/10)
   glColor4ub(0, 0, 255, 255); // RGBA=(0,0,1,100%)
   glVertex3f(0.0, -0.8, -0.2); // XYZ=(0,-8/10,-2/10)
} glEnd();
```
Within the draw routine

```c
GLShadeModel(GL_SMOOTH); // smooth color interpolation
Enable(GL_DEPTH_TEST); // enable hidden surface removal

Clear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
Begin(GL_TRIANGLES);
    Color4ub(255, 0, 0, 255); // RGBA=(1,0,0,100%)
    Vertex3f(-0.8, 0.8, 0.3); // XYZ=(-8/10,8/10,3/10)
    Color4ub(0, 255, 0, 255); // RGBA=(0,1,0,100%)
    Vertex3f(0.8, 0.8, -0.2); // XYZ=(8/10,8/10,-2/10)
    Color4ub(0, 0, 255, 255); // RGBA=(0,0,1,100%)
    Vertex3f(0.0, -0.8, -0.2); // XYZ=(0,-8/10,-2/10)
End();
```

graphics context state is “stateful” (sticky) so technically doesn’t need to be done every time display is called
State Updates

- ShadeModel(SMOOTH) requests smooth color interpolation
  - changes fragment shading state
  - alternative is “flat shading”
- Enable(DEPTH_TEST) enables depth buffer-based hidden surface removal algorithm
- State updates happen in command sequence order
- In fact, all OpenGL commands are in a stream that must complete in order

Diagram:

1. Application
2. Vertex batching & assembly
3. Clipping
4. NDC to window space
5. Rasterization
6. Fragment shading
7. Depth testing
8. Color update
9. Depth buffer
10. Framebuffer

University of Texas at Austin    CS354 - Computer Graphics    Don Fussell
Clearing the buffers

- Within the draw routine

```c
glShadeModel(GL_SMOOTH);  // smooth color interpolation
glEnable(GL_DEPTH_TEST);  // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES);  // every 3 vertexes makes a triangle
    glColor4ub(255, 0, 0, 255);  // RGBA=(1,0,0,100%)
    glVertex3f(-0.8, 0.8, 0.3);  // XYZ=(-8/10,8/10,3/10)
    glColor4ub(0, 255, 0, 255);  // RGBA=(0,1,0,100%)
    glVertex3f(0.8, 0.8, -0.2);  // XYZ=(8/10,8/10,-2/10)
    glColor4ub(0, 0, 255, 255);  // RGBA=(0,0,1,100%)
    glVertex3f(0.0, -0.8, -0.2);  // XYZ=(0,-8/10,-2/10)
glEnd();
```
Buffer Clearing

- New frame needs to reset entire color buffer to “background” or “clear” color
  - Avoids having remnants of prior frame persist
    - Needed if can’t guarantee every pixel is touched every frame
- Depth buffer needs to be cleared to “farthest value”
  - More about depth buffering later
- Special operation in OpenGL
  - Hardware wants clears to run at memory-saturating speeds
  - Still in-band with command stream
Clear Values and Operations

- OpenGL commands to set clear values
  - `glClearColor` for RGBA color buffers
    - **Example:** `glClearColor(0,0,0,1);`
    - Clear to black with 100% opacity
    - Initial clear value is (0,0,0,0) so black with 0% opacity
  - `glClearDepth` for depth buffers
    - **Example:** `glClearDepth(1.0);`
    - Clear to farthest depth value, for [0,1] range
    - Initial depth clear value is 1.0 so farthest depth value
  - Neither commands does the actual clear operation…

- That’s done by `glClear(mask)`
  - Mask parameter indicates buffers to clear
    - `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`
    - Bitwise-OR (|) them together
    - Also `GL_STENCIL_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT`
  - Allows multiple buffers (e.g. depth & color) to be cleared in single operation, possibly in parallel
Batching and Assembling Vertices

- `glBegin` and `glEnd` designate a batch of primitives
  - Begin mode of `GL_TRIANGLES` means every 3 vertices = triangle
- Various vertex attributes
  - Position attribute sent with `glVertex*` commands
  - Also colors, texture coordinates, normals, etc.
- `glVertex*` assembles a vertex and puts it into the primitive batch
  - Other vertex attribute commands such as `glColor*` have their attributes "latched" when `glVertex*` assembles a vertex

```
Application
  ↓
Vertex batching & assembly
  ↓
Clipping
  ↓
NDC to window space
  ↓
Rasterization
  ↓
Fragment shading
  ↓
Depth testing
  ↓
Color update
  ↓
Depth buffer
  ↓
Framebuffer
```
Assembling a Vertex

*glVertex* command assembles a complete vertex

Assemble a vertex with all its attributes

- `glColor4f`, `glColor3f`, `glColor4ub`, etc.
- `glTexCoord2f`, `glTexCoord3s`, `glTexCoord4i`, etc.
- `glNormal3f`, `glNormal3s`, `glNormal3b`, etc.

`glVertex*` command assembles a complete vertex

- `glVertex2s`
- `glVertex3f`
- `glVertex4d`
Vertex Attribute Commands

- OpenGL vertex attribute commands follow a regular pattern
  - gl-prefix :: common to all OpenGL API calls
  - Vertex, Normal, TexCoord, Color, SecondaryColor, FogCoord, VertexAttrib, etc.
    - Name the semantic meaning of the attribute
    - VertexAttrib is for generic attributes
      - Used by vertex shaders where the shader determines “meaning” of attributes
      - Attribute zero & Vertex are “special”—they latch the assembly of a vertex
  - 1, 2, 3, 4 :: Number of components for the attribute
    - For an attribute with more components than the number, sensible defaults apply
      - For example, 3 for Color means Red, Green, Blue & Alpha assumed 1.0
  - f, i, s, b, d, ub, us, ui
    - Type of components: float, integer, short, byte, double, unsigned byte, unsigned short, unsigned integer
  - v :: means parameters are passed by a pointer
    - Instead of immediate values
Example

Consider `glColor4ub` and `glVertex3fv`
Assemble a Triangle

- Within the draw routine

```c
glBegin(GL_TRIANGLES);

    glColor4ub(255, 0, 0, 255);
    glVertex3f(-0.8, 0.8, 0.3);

    glColor4ub(0, 255, 0, 255);
    glVertex3f(0.8, 0.8, -0.2);

    glColor4ub(0, 0, 255, 255);
    glVertex3f(0.0, -0.8, -0.2);

glEnd();
```
glBegin Primitive Batch Types

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_POLYGON
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_QUADS
- GL_QUAD_STRIP
Assembly State Machines

- Fixed-function hardware performs primitive assembly
  - Based on glBegin’s mode
- State machine for GL_TRIANGLES

```
Begin(TRIANGLES)

no vertex

End

Vertex

one vertex

End

Vertex

two vertexes

End

Vertex / Emit Triangle

initial
```
GL_TRIANGLE_STRIP

Begin(TRIANGLE_STRIP)

no vertex → Vertex

one vertex → Vertex

two vertexes → Vertex

End

End

End

End

Vertex / Emit Reverse Triangle

Vertex / Emit Triangle

initial

no vertex

one vertex

two vertexes
GL_POINTS and GL_LINES

Actual hardware state machine handles all OpenGL begin modes, so rather complex
Triangle Assembly

- Now we have a triangle assembled
- Later, we’ll generalize how the vertex positions get transformed
  - And other attributes might be processed too
- For now, just assume the XYZ position passed to `glVertex3f` position is in NDC space
Our Newly Assembled Triangle

Think of drawing into a \([-1,+1]^3\) cube

Origin at \((0,0,0)\)

Vertices:
- \((-1.8, 0.8, 0.3)\)
- \((-0.8, 0.8, -0.2)\)
- \((0, -0.8, -0.2)\)
What if any portion of our triangle extended beyond the NDC range of the \([-1,+1]^3\) cube?

- Only regions of the triangle \([-1,+1]^3\) cube should be rasterized!

No clipping for our simple triangle

- This situation is known as “trivial accept”
- Because all 3 vertices in the \([-1,+1]^3\) cube

Triangles are convex, so entire triangle must also be in the cube if the vertexes are 

\((-0.8, 0.8, 0.3)\)  
\((-0.8, 0.8, -0.2)\)  
\((0, -0.8, -0.2)\)

origin at \((0,0,0)\)
Triangle Clipping

- Triangles can straddle the NDC cube
  - Happens with lines too
- In this case, we must "clip" the triangle to the NDC cube
  - This is an involved process but one that must be done
Consider a Different Triangle

- Move left vertex so it’s $X = -1.8$
- Result is a clipped triangle

![Diagram showing a triangle with vertices at (-1.8, 0.8, 0.3), (-0.8, 0.8, -0.2), and (0, -0.8, -0.2), with an origin at (0,0,0).]
Clipped and Rasterized Normally

Visualization of NDC space

Notice triangle is “poking out” of the cube; this is the reason that should be clipped
New triangles out

But how do we find these “new” vertices?

The edge clipping the triangle is the line at $X = -1$
so we know $X = -1$ at these points—but what about $Y$?
Use Ratios to Interpolate Clipped Positions

(-1, 0.8, 0.146153)

(-1.8, 0.8, 0.3)

(-0.8, 0.8, -0.2)

(0, -0.8, -0.2)

(-1, 0.8, 0.146153)

0.8(-1.8)=2.6

0.8(-1)=1.8

(-0.8, 0.8, -0.2)

origin at (0,0,0)

Weights:
1.8/2.6
0.8/2.6, sum to 1

Straightforward because all the edges are orthogonal

X = -1
Y = (1.8/2.6)×0.8 + (0.8/2.6)×0.8
= 0.8
Z = (1.8/2.6)×0.3 + (0.8/2.6)×-0.2
= 0.1461538
Use Ratios to Interpolate Clipped Positions

(-1.8, 0.8, 0.3)

(-0.8, 0.8, -0.2)

origin at (0,0,0)

0-(-1) = 1

X = -1

Y = (1/1.8)×0.8 + (0.8/1.8)×-0.8
   = 0.08888...

Z = (1/1.8)×0.3 + (0.8/1.8)×-0.2
   = 0.07777...

Weights:
1/1.8
0.8/1.8, sum to 1
Given primitive may be clipped by multiple cube faces

- Potentially clipping by all 6 faces!

**Approach**

- Four possibilities
  - Face doesn’t actually result in any clipping of a triangle
    - Triangle is unaffected by this plane then
  - Clipping eliminates a triangle completely
    - All 3 vertices on “wrong” side of the face’s plane
  - Triangle “tip” clipped away
    - Leaving two triangles
  - Triangle “base” is clipped away
    - Leaving a single triangle

**Strategy**: implement recursive clipping process

- “Two triangle” case means resulting two triangles must be clipped by all remaining planes
Attribute Interpolation

- When splitting triangles for clipping, must also interpolate new attributes
  - For example, color
  - Also texture coordinates

- Back to our example
  - BLUE\(\times 0.8/1.8\) + RED\(\times 1/1.8\)
    - (0,0,1,1)\(\times 0.8/1.8\) + (1,0,0,1)\(\times 1/1.8\)
    - (0.444,0,.555,1) or MAGENTA

Weights:
- 1/1.8
- 0.8/1.8, sum to 1
What to do about this?

- Several possibilities
  - Require applications to **never** send primitives that require clipping
    - Wishful thinking
    - And a cop-out—makes clipping their problem
  - Rasterize into larger space than normal and discard pixels outsize the NDC cube
    - Increases useless rasterizer work
      - Requires additional math precision in the rasterizer
    - Worse, creates problems when rendering into a projective clip space (needed for perspective)
      - Something for a future lecture
  - Break clipped triangles into smaller triangles that tessellate the clipped region…
Triangle clipped by Two Planes

Recursive process can make 4 triangles
And it gets worse with more non-trivial clipping
NDC to Window Space

- NDC is “normalized” to the \([-1,+1]^3\) cube
  - Nice for clipping
  - But doesn’t yet map to pixels on the screen

- **Next:** a transform from NDC space to window space
Viewport and Depth Range

- OpenGL has 2 commands to configure the state to map NDC space to window space

  - `glViewport(GLint vx, GLint vy, GLsizei w, GLsizei h);`
    - Typically programmed to the window’s width and height for \( w \) & \( h \) and zero for both \( vx \) & \( vy \)
    - **Example:** `glViewport(0, 0, window_width, window_height);`

  - `glDepthRange(GLclampd n, GLclampd f);`
    - \( n \) for near depth value, \( f \) for far depth value
    - Normally set to `glDepthRange(0,1)`
      - Which is an OpenGL context’s initial depth range state

- The mapping from NDC space to window space depends on \( vx, vy, w, h, n, \) and \( d \)
The OpenGL specification allow an implementation to specify how language data types map to OpenGL API data types
- GLfloat is usually typedef'd to float but this isn’t necessarily true
  - Same for GLint, GLshort, GLdouble
  - But is true in practice
- GLbyte is byte-sized so expected it to be a char
- GLubyte, GLushort, and GLuint are unsigned versions of GLbyte, GLshort, and GLint

Certain names clue you into their parameter usage
- GLsizei is an integer parameter that is not allowed to be negative
  - An GL_INVALID_VALUE is generated if a GLsizei parameter is ever negative
- GLclampd and GLclampf are the same as GLfloat and GLdouble, but indicate the parameter will be clamped automatically to the [0,1] range

Notice
- glViewport uses GLsizei for width and height
- glDepthRange uses GLclampd for near and far
OpenGL Errors

- OpenGL reports asynchronously from your commands
  - Effectively, you must explicitly call glGetError to find if any prior command generated an error or was otherwise used incorrectly
  - glGetError returns GL_NO_ERROR if there is no error
  - Otherwise an error such as GL_INVALID_VALUE is returned

- Rationale
  - OpenGL commands are meant to be executed in a pipeline so the error might not be identified until after the command’s function has returned
    - Errors might be detected by hardware that isn’t actually the CPU
    - Also forcing applications to check return codes of functions is slow
      - It’s inappropriate for a high-performance API such as OpenGL
  - So if you suspect errors, you have to poll for them
    - Learn to do this while you are debugging your code
    - If something fails to happen, suspect there’s an OpenGL errors
  - Also commands that generated an error are ignored
    - The only exception is GL_OUT_OF_MEMORY which results in undefined state
Mapping NDC to Window Space

- Assume \((x, y, z)\) is the NDC coordinate that’s passed to `glVertex3f` in our `simple_triangle` example.

- Then window-space \((w_x, w_y, w_z)\) location is:
  - \(w_x = (w/2) \times x + v_x + w/2\)
  - \(w_y = (h/2) \times y + v_y + h/2\)
  - \(w_z = [(f-n)/2] \times z + (n+f)/2\)  \(\times\) means scalar multiplication here.


Where is glViewport set?

- The simple_triangle program never calls glViewport
  - That’s OK because GLUT will call glViewport for you if you don’t register your own per-window callback to handle when a window is reshaped (resized)
  - Without a reshape callback registered, GLUT will simply call glViewport(0, 0, window_width, window_height);
- Alternatively, you can use glReshapeFunc to register a callback
  - Then calling glViewport or otherwise tracking the window height becomes your application’s responsibility
  - Example reshape callback:
    ```c
    void reshape(int w, int h) {
        glViewport(0, 0, w, h);
    }
    ```
  - Example registering a reshape callback:
    ```c
    glReshapeFunc(reshape);
    ```
- FYI: OpenGL maintains a lower-left window-space origin
  - Whereas most 2D graphics APIs use upper-left
What about `glDepthRange`?

- Simple applications don’t normally need to call `glDepthRange`
  - Notice the simple_triangle program never calls `glDepthRange`

- Rationale
  - The initial depth range of [0,1] is fine for most application
  - It says the entire available depth buffer range should be used
  - When the depth range is [0,1] the equation for window-space z simplifies to \( wz = \frac{1}{2} \times z + \frac{1}{2} \)
Triangle Vertices in Window Space

- Assume the window is 500x500 pixels
- So `glViewport(0,0,500,500)` has been called

\[-0.8, 0.8, 0.3\]  \([-0.8, 0.8, -0.2]\]
\[(0, -0.8, -0.2)\]  \[\text{origin at } (0,0,0)\]
Apply the Transforms

- **First vertex :: (-0.8, 0.8, 0.3)**
  - \(w_x = (w/2)x + v_x + w/2 = 250 \times (-0.8) + 250 = 50\)
  - \(w_y = (h/2)y + v_y + h/2 = 250 \times (0.8) + 250 = 450\)
  - \(w_z = [(f-n)/2]z + (n+f)/2 = 0.65\)

- **Second vertex :: (0.8, 0.8, -0.2)**
  - \(w_x = (w/2)x + v_x + w/2 = 250 \times (-0.8) + 250 = 50\)
  - \(w_y = (h/2)y + v_y + h/2 = 250 \times (0.8) + 250 = 450\)
  - \(w_z = [(f-n)/2]z + (n+f)/2 = 0.4\)

- **Third vertex :: (0, -0.8, -0.2)**
  - \(w_x = (w/2)x + v_x + w/2 = 250 \times 0 + 250 = 250\)
  - \(w_y = (h/2)y + v_y + h/2 = 250 \times (-0.8) + 250 = 50\)
  - \(w_z = [(f-n)/2]z + (n+f)/2 = 0.4\)
Still Left to Do

- Rasterize the clipped triangle
  - But our triangle’s vertexes are in window space so we are ready
- Interpolate color values over the triangle
- Depth test the triangle
- Update pixel locations
- Swap buffers
- Next lecture!
Next Lecture

- Graphics Pipeline
  - *What are the operations in the so-called “graphics pipeline”?*
  - As usual, expect a short quiz on today’s lecture
    - Know how to map clip space to NDC space to window space

- Assignments
  - Reading from “Interactive Computer Graphics” (Angel)
    - Chapter 2, pages 43-107
  - Homework (a.k.a. Project Zero), **deadline** January 25th
    - Get the ZIP for the “simple triangle” and “clip space” example programs
    - Learn how to compile and run them on your CS account
    - Modify either program to
      - Change the clear color to burnt orange
      - Change the title of the window to your name
      - Instead of drawing a single triangle, make a simple arrangement of polygons forming a letter from your name
    - Use the **turnin** system to submit your modified source code and a screenshot image of your modified example

**Purpose**
Gain familiarity with OpenGL programming and submitting projects
Programming tips

- 3D graphics, whether OpenGL or Direct3D or any other API, can be frustrating
- You write a bunch of code and the result is

Nothing but black window; where did your rendering go??
Things to Try

- Set your clear color to something other than black!
  - It is easy to draw things black accidentally so don’t make black the clear color
  - But black is the initial clear color
- Did you draw something for one frame, but the next frame draws nothing?
  - Are you using depth buffering? Did you forget to clear the depth buffer?
- Remember there are near and far clip planes so clipping in Z, not just X & Y
- Have you checked for glGetError?
  - Call glGetError once per frame while debugging so you can see errors that occur
  - For release code, take out the glGetError calls
- Not sure what state you are in?
  - Use glGetIntegerv or glGetFloatv or other query functions to make sure that OpenGL’s state is what you think it is
- Use glutSwapBuffers to flush your rendering and show to the visible window
  - Likewise glFinish makes sure all pending commands have finished
- Try reading
  - [http://www.slideshare.net/Mark_Kilgard/avoiding-19-common-opengl-pitfalls](http://www.slideshare.net/Mark_Kilgard/avoiding-19-common-opengl-pitfalls)
  - This is well worth the time wasted debugging a problem that could be avoided