OpenGL and Project 0
Summary

- OpenGL Overview
- Primitives and Drawing
- Transformations
- Lighting and Materials
- Project 0
OpenGL Overview
The Red Book

- Your best resource for OpenGL is this book.
- But you don’t have to buy it because there’s an online version!
- Not to be confused with the Orange Book (or the Blue Book or the Green Book...)

[Image of the book cover]
What is OpenGL?

- OpenGL is a software API consisting of many functions that allow you to talk to your graphics hardware. It is cross-platform and is most commonly used in professional graphics applications.
OpenGL was designed to produce reasonably looking 3D images quickly and simply. Therefore, a lot of its design is a rough approximation of how visual phenomena behave in the real world. Keep this in mind as we discuss how it works.
The OpenGL Pipeline
Few OpenGL functions cause anything to be drawn. Instead, they modify state of the OpenGL driver.

- Draw calls use the current state. Example draw calls are
  - glVertex, glDrawElements
Buffers

- OpenGL stores per-pixel information in different buffers.
- The frame buffer is the final output to the screen.
- A collection of intermediate buffers store information about pixels.
  - If we are rendering multiple frames, we need to clear the buffers before we draw a new frame.
OpenGL allows us to modify the following buffers:

- Color buffers (multiple)
  - contain information about the color of pixel
- Depth (z) buffer
  - stores depth information of each pixel, allowing closer objects to be drawn over those farther away
- Stencil buffer
  - extra buffer used for advanced techniques
- Accumulation buffer
  - extra buffer used for special effects
Primitives and Drawing
Primitives

- 3D drawing in OpenGL uses primitives.
- Primitives are points, lines, and various polygons that make up larger objects.
- Every primitive is defined by some number of points, or vertices, which are specified during draw calls.
Types of Primitives

- **GL_POINTS**
  - Simply draws single vertices in the order you pass them in.
- **GL_LINES**
  - Takes pairs of vertices and draws lines between them.
- **GL_LINE_STRIP**
  - Takes any number of vertices and draws a series of connected line segments.
- **GL_LINE_LOOP**
  - Same as above, but with the first and last endpoints connected.

![Diagrams](image-url)
Types of Primitives

- **GL_TRIANGLES**
  - Takes vertices in triples and draws them as triangles.

- **GL_QUADS**
  - Takes vertices in quadruples and draws them as four-sided polygons.

- **GL_POLYGON**
  - Takes any number of vertices and draws the boundaries of the convex polygon that they form.
  - Note: Order of vertices here is important. All polygons must be convex and their edges cannot intersect.
Types of Primitives

- Most objects can be drawn using only GL_TRIANGLES.
- Note: There are other primitives (GL_TRIANGLE_STRIP, GL_QUAD_STRIP, and GL_TRIANGLE_FAN). You can find more information about these in the red book.
Vertex Attributes

- Each vertex must be specified by at least a position (glVertex).
- OpenGL has other built in attributes:
  - Normals
  - Color
  - Texture coordinates
The simplest way to draw primitives is to use the OpenGL begin and end calls.

- Declare our primitive:
  - `glBegin( ... )`

- Specify our vertices (and attributes)
  - `glNormal*( ... )`
  - `glVertex*( ... )`

- Declare that we are finished:
  - `glEnd()`
// Sample drawing function
void display_square() {
    ...  
    glBegin(GL_QUADS);
    glColor3f( 0.0, 0.0, 1.0 ); // sets color to blue
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 0.0, 1.0 );
    glColor3f( 1.0, 0.0, 0.0 ); // sets color to red
    glVertex2f( 1.0, 1.0 );
    glVertex2f( 1.0, 0.0 );
    glEnd();
}
The downside of using glVertex is that for anything but simple models, we must make an enormous number of function calls. Consider a cube. Each vertex needs to be declared three times, once for each face it is a part of, resulting in 24 glVertex calls.
OpenGL provides vertex array routines that allow you to specify vertex data using arrays and few function calls.

You must enable your array:
- `glEnableClientState( ... )`
  - Specify the type of array (GL_VERTEX_ARRAY)

You must specify data for your array
- `glVertexPointer( ... )`
  - Specify the number of coordinates per vertex, type, byte offset between vertices, and a pointer to the first vertex in input array
  - All other attributes (e.g. normal) have their own arrays.
There are three ways to call access elements of the vertex array:

- **glArrayElement( ... )**
  - Draws a single vertex
- **glDrawArrays( ... )**
  - Draws a sequence of vertices
- **glDrawElements( ... )**
  - Draws a sequence of vertices based on an indexed array.
  - Generally, you want to use this one.
// Sample code using vertex arrays
void init_array() {
    float vertices[] = { 1.0, 0.0, 0.0, 
                        0.0, 1.0, 0.0, 
                        0.0, 0.0, 1.0 };
    glEnableClientState( GL_VERTEX_ARRAY );
    glVertexPointer( 3, GL_FLOAT, 0, vertices );
}

void display_square() {
    unsigned int indices[] = { 0, 1, 2 };
    glDrawElements( GL_TRIANGLES, 3, GL_UNSIGNED_INT, 
                    indices );
}
Display Lists

- Display lists provide a way for OpenGL to redraw arbitrary primitives with a single draw call.
- Display lists compile a set of commands that draw a particular object.
- Use them only for static (unchanging) geometry.
Display Lists

- Generate a new list
  - glGenLists( ... )
- Declare a new list
  - glNewList( ... )
- Draw an object in between
- Declare the end of the list
  - glEndList()
- Anytime you want to draw your object, call:
  - glCallList( ... )
int list;

// Some method called during initialization
void initialize_triangle(){
    list = glGenLists( 1 );
    glNewList( list, GL_COMPILE );
    glBegin( GL_TRIANGLE );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 0.0, 1.0 );
    glVertex2f( 1.0, 1.0 );
    glEnd();
    glEndList();
}

// Sample drawing function
void display_callback(){
    ...
    glCallList( list );
}
Transformations
Right now, we can draw simple primitives and build scenes.

However, transformations of the original vertices may be required in order to properly view our objects.
Transformations in OpenGL are accomplished using matrices.

OpenGL tracks two different vertices for vertex transformations:
- **ModelView Matrix (GL_MODELVIEW)**
  - These concern model-related operations such as translation, rotation, and scaling, as well as viewing transformations.
- **Projection Matrix (GL_PROJECTION)**
  - Setup camera projection.
// Sample code showing matrix transformations
define display_object()
{
  ...
  glMatrixMode(GL_MODELVIEW); // Loads matrix
  glLoadIdentity(); // Clears the matrix
  glTranslatef(0.0, 0.0, -2.0); // Some transformations
  glRotatef(45.0, 0.0, 0.0, 1.0);
  glScalef(2.0, 2.0, 2.0);
  draw_object();
}
OpenGL allows for 3 basic model transformations:

- **glTranslate***(TYPE x, TYPE y, TYPE z)*
  - Translates an object by given x, y, and z.

- **glRotate***(TYPE angle, TYPE x, TYPE y, TYPE z)*
  - Rotates an object by given angle counterclockwise around the vector(x, y, z)

- **glScale***(TYPE x, TYPE y, TYPE z)*
  - Scales an object by multiplying vertex by given
In OpenGL, you want to call the transformations in the reverse order that you want them applied.
Each transform multiplies the vertices by a matrix (and the transform closest to the vertices gets multiplied first).
Order Matters!

- Since matrix multiplication is not commutative, the order in which you apply transformations can change the final result.
- Generally, you want to apply scaling first, then rotation, and finally translation (see the previous code example for how this is done).
Pushing and Popping

- You may wish to save and reload the current matrix using a stack.
  - `glPushMatrix()`
    - Pushes a matrix onto the stack for later use.
  - `glPopMatrix()`
    - Returns to the most recently-pushed matrix.
- This is particularly useful when thinking about object hierarchies.
// Simple code for object transformation (cube and robot)
void some_method() {
    glPushMatrix(); // Saves current matrix
        transform_cube();
        draw_cube();
    glPopMatrix(); // Returns current matrix
    glPushMatrix();
        transform_body();
    glPushMatrix();
        transform_left_arm();
        draw_left_arm();
    glPopMatrix();
    glPushMatrix();
        transform_right_arm();
        draw_right_arm();
    glPopMatrix();
    draw_body();
    glPopMatrix();
}
Viewing

- We also handle viewing transformations here, which specify the viewpoint of our scene.
- You *could* position all of your objects perfectly using lots and lots of transformations.
- It’s like looking through a camera and moving all of the objects into position. Except why move all of the objects around when we could just move the camera?
Fortunately, there’s a handy function already:

- `gluLookAt(GLdouble e_x, GLdouble e_y, GLdouble e_z, GLdouble c_x, GLdouble c_y, GLdouble c_z, GLdouble u_x, GLdouble u_y, GLdouble u_z)`
  - `e_x, e_y, and e_z` specify the desired viewpoint
  - `c_x, c_y, c_z` specify some point along the desired line of sight
  - `u_x, u_y, and u_z` define the up vector of our camera

Since we want to transform to our viewpoint before transforming our objects, we want this to be the last transformation applied (which means we should call it first, right after we load the identity).
Matrix mode use is similar:
- glMatrixMode(GL_PROJECTION)

We are really concerned with only two types of projection transformations:
- Orthographic projection
  - Our viewing volume is rectangular and all objects appear the same size no matter the distance
- Perspective projection
  - Uses perspective to give a sense of depth. Our viewing volume is conical or pyramidal in shape.
OpenGL provides a call to setup orthographic projection:

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
             GLdouble top, GLdouble near, GLdouble far);
```

- `left, right, top, and bottom` define the boundaries of the near/far clipping planes.
- `near` and `far` specify how far from the viewpoint the near and far clipping planes are.
We have two calls to use:

```c
gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far )
```

- `fovy` is the angle in the field of view (in range from [0.0, 180]
- `aspect` is the aspect ratio of the frustrum (width of window over height of window)
- `near` and `far` are the values between viewpoint and the near/far clipping planes
glFrustum( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far )

- left, right, top, and bottom define the boundaries of the near clipping plane
- near and far specify how far from the viewpoint the near and far clipping planes are
Lighting and Materials
By default, OpenGL’s fixed-function pipeline implements the Blinn-Phong Shading Model.
- The model provides an approximation for how to represent color and lighting for both lights and object materials.
Blinn-Phong

\[ I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^\alpha i_s) \]

- \( k_a, k_d, \) and \( k_s \) are the ambient, diffuse, and specular terms of the materials (objects)
- \( i_a, i_d, \) and \( i_s \) are the ambient, diffuse, and specular intensities of the lights
- Dot products: provide the dependence on the light-surface and reflection-viewer angles (this will be discussed during the lighting lecture)
The ambient term approximates the low level of light that is normally present everywhere in a scene (scattered by many objects before reaching the eye).

Constant term; is applied equally to all points on the object

\[ I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V) \alpha i_s) \]
The diffuse term approximations light scattered by objects with rough surfaces. Its intensity depends on the angle between the light source and the surface normal (not the direction to the viewer).

\[ I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^\alpha i_s) \]
The specular term approximates light reflected by “shiny” objects with smooth surfaces.

Its intensity depends on the angle between the viewer and the direction of a ray reflected from the light source.

\[ I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^a i_s) \]
OpenGL solves the lighting equation for you, but you have to specify properties of the light. You must enable lighting and lights:

- `glEnable(GL_LIGHTING)`
- `glEnable(GL_LIGHT*)`

You also must set properties of each light:

- `glLight*( ... )`
  - Specify the particular light (e.g. `GL_LIGHT0`), the parameter of the light to set (`GL_AMBIENT`, `GL_POSITION`, etc), and the value of the parameter.
Similarly, you must set properties about the materials:

- `glMaterial*( ... )`
  - Specify what face to apply the material to (GL_FRONT, GL_BACK, GL_FRONT_AND_BACK), the parameter to set (GL_DIFFUSE, GL_AMBIENT), and the value for that parameter.

- Also, don’t forget to define normals for every vertex!
// Sample code to with lights and materials
void set_lights_materials() {
    GLfloat red = (1.0, 0.0, 0.0, 1.0);
    glEnable(GL_LIGHTING); // Enables lighting
    glEnable(GL_LIGHT0); // Enables light 0

    glMaterialfv(GL_FRONT, GL_DIFFUSE, red); // Sets diffuse component of material to red
    glLightfv(GL_LIGHT0, GL_SPECULAR, red); // Sets specular component of light 0 to red
}

void display_triangle() {
    glBegin(GL_TRIANGLES);
    glNormal3f(0.0, 0.0, 1.0); // Specifying normals
    glVertex3f(0.0, 0.0, 0.0);
    glNormal3f(0.0, 0.0, 1.0);
    glVertex3f(0.0, 1.0, 0.0);
    glNormal3f(0.0, 0.0, 1.0);
    glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
    glEnd();
}
Limitations

- The pipeline is a one way machine.
- Once a polygon has been rendered, it is forgotten and can’t be used for any other lighting calculations.
- Also notice that the Blinn-Phong equation only takes into account an object’s material properties and the properties of the light.
Limitations

- OpenGL can only capture illuminations from the light source -> object -> viewer, or “direct illumination”.
- Global, or indirect illumination is never accounted for (along with other things like reflections, caustics, etc.).
Be happy with Blinn-Phong.
- Projects 0 and 1
- Program the pipeline to behave differently using shaders.
  - Check out the Orange Book for OpenGL’s shader language (GLSL).
  - This is what every modern OpenGL application uses.
- Ditch OpenGL and use a completely different algorithm such as ray tracing or photon mapping, which supports global illumination.
  - Project 2
Project 0
Outline of the Projects

- **Project 0: Basic OpenGL**
  - An introduction to starter code framework and basic OpenGL.
  - Render a model in OpenGL.
- **Project 1: Splines**
  - A multi-part project using 2D and 3D splines to draw objects and animate scenes.
- **Project 2: Ray Tracer**
  - Break away from OpenGL to implement a more complicated rendering model.
Project 0 Preview
You must use OpenGL techniques discussed today to render a simple triangle mesh.

- You must use the transformation matrices to position a given camera correctly.
- You must render the model as triangle primitives.
- You must add lights and basic materials to your object.
- You must make screen shots of your program.
The starter code for each assignment is in C++.  
- We find that matrix algebra can be much more cleanly implemented using classes and operator overloading than just C functions.
- We know that some of you may have never programmed in C++.
  - We restrict ourselves to a very limited subset of C++.
  - The first assignment is small to allow you to become familiar with it.
  - If you do not know it, you should begin learning it very shortly. Talk to course staff for help.
- We do expect you to be intimately familiar with C.
Handin directories have been created on AFS.
- You will turn in each project there before the deadline.
- There are **no late days**. We will write lock the directory at the deadline.

Your submission includes:
- Code files
- Screen shots
- Models
- A file describing your submission and implementation
Grading

- You are graded on both program output and the code itself
  - Program output.
    - We look at screen shots
    - We also see how your program runs.
  - Code.
    - We do read and grade your actual code.
    - Readability is important. The graders must be able to read and understand what your code is doing. Good documentation is essential.
    - Your implementation should be clean and correct. Part of your grade is dependent on how you organize your implementation.
    - You should follow basic good practices. For example...
      - Free any memory you allocate
      - Make sure to initialize memory
Questions?
P0 will be out later today!