15-462: Computer Graphics
Originally by: Kristin Siu, Eric Butler
Edited by: Ilya Gershgorin
Given by: Derek Basehore

OpenGL Programming

TA Office Hours

- Chun How:
  - Mon. 8pm-10pm GHC 5205
- Derek:
  - Wed. 8pm-10pm GHC 5205

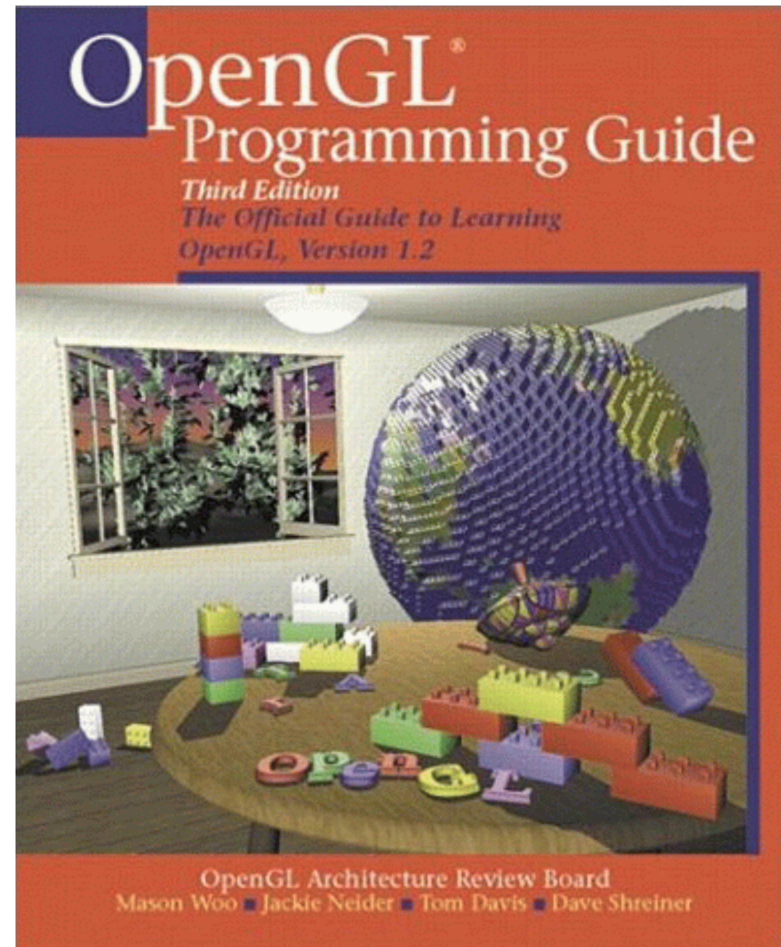
Summary

- OpenGL Overview
- The OpenGL Pipeline
- Drawing Primitives
- Transformations, Lighting, and Materials
- Projects

OpenGL Overview

The Red Book

- Your best resource for OpenGL is this book. You'll need it for most class projects.
- But you don't have to buy it because there's an online version! (Not the most recent, but it covers everything you'll need)



What is OpenGL?

- A software API with many functions that allow communication with graphics hardware.
- Cross-platform
- Commonly used in many graphics applications
  - Games
  - CAD
  - Visualization

Design and Limitations

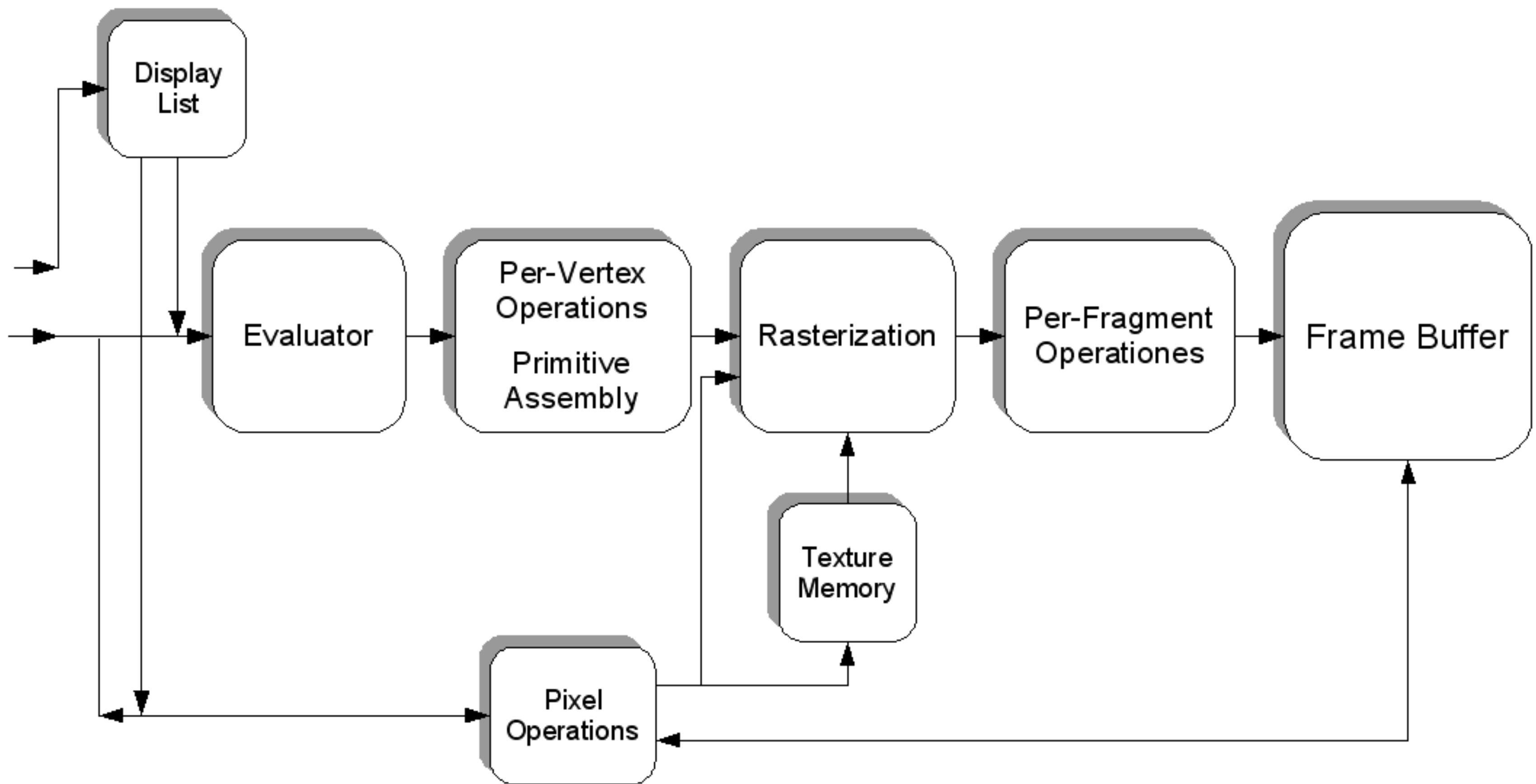
- OpenGL was designed to produce reasonably looking 3D images quickly and simply.
  - A lot of its design is a rough approximation of how visual phenomena behave in the real world.
  - Meant to only handle rendering, nothing else.
    - So no window management, event-handling, etc.
  - Meant to abstract away graphics hardware into a standard, clean interface.

OpenGL is a state-based API

- Most functions manipulate global state.
- 3 types of functions:
  - Those that modify global state.
  - Those that query global state.
  - Those that cause something to be rendered.
    - e.g., `glEnd` or `glDrawElements`

The OpenGL Pipeline

The OpenGL Pipeline

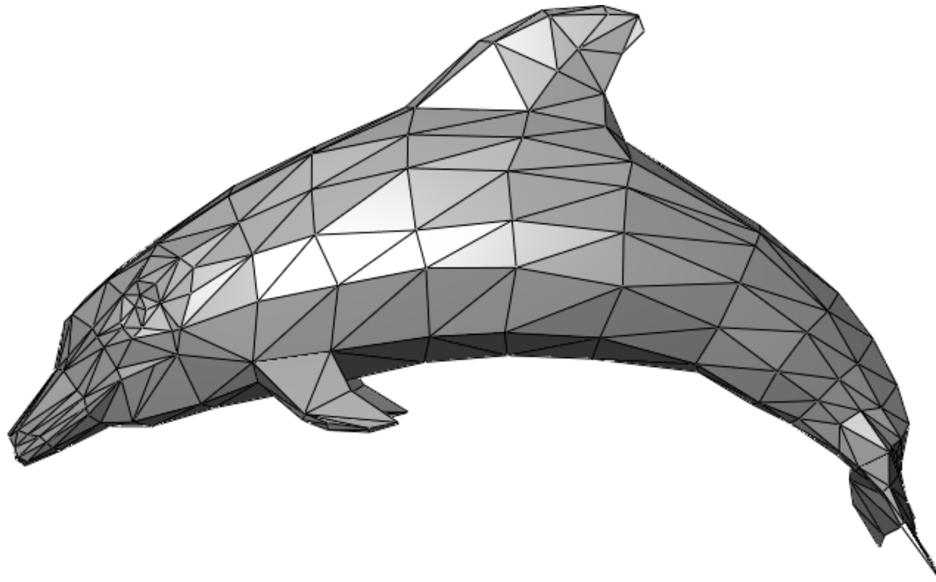


The OpenGL Pipeline

- Rendering is *object-based*
  - Vertices and fragments are processed in parallel, independently of each other
  - No global effects
- Basic Process:
  - Evaluation of *uniform* and *attribute* data
  - Processing of *vertices*
  - Assembly of *vertices* into *primitives*
  - Rasterization of *primitives* into *fragments*
  - Processing of *fragments*
  - Composition of *fragments* into the *frame buffer*

Vertices and Primitives

- Objects in OpenGL are built by assembly *vertices* into *primitives*.
  - Example: A triangle is a primitive with 3 vertices.



Specifying Data

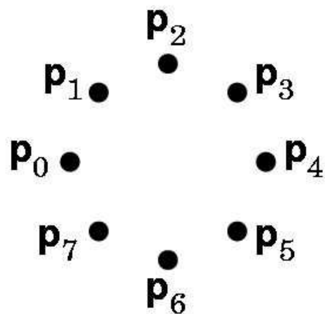
- There are 2 primary types of data passed to OpenGL to render.
- *Attribute* Data
  - Per-vertex information.
  - e.g., position, normal, color, texture coordinate.
- *Uniform* Data
  - Data that is the same for entire primitive.
  - e.g., transformation matrix, diffuse color.

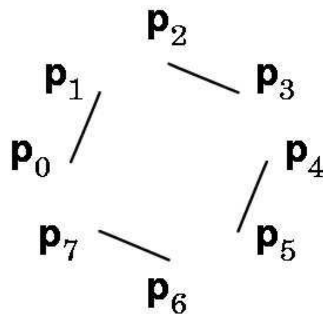
Vertex Processing, Primitive Assembly

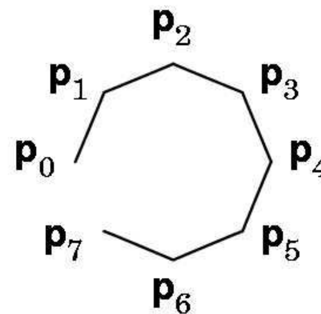
- Per-vertex operations are run on each vertex.
  - Example: Transformations
    - Figures out where vertex is relative to screen.
    - Rotation, scaling, translation
    - Camera projection
- Then vertices are assembled into primitives, where more processing occurs.
  - e.g., clipping, depth culling.

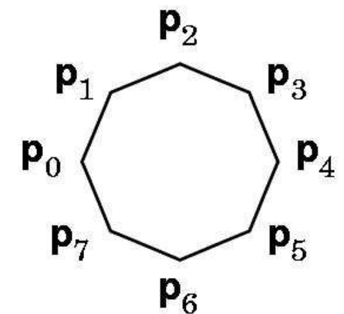
Types of Primitives

- 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.



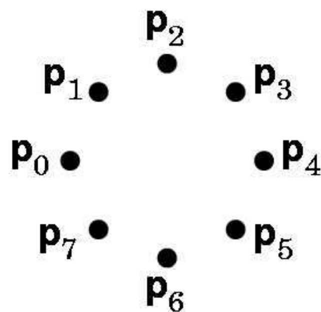


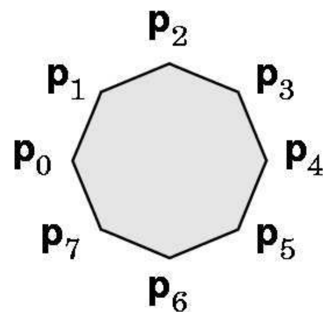


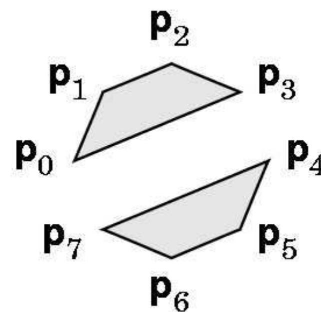


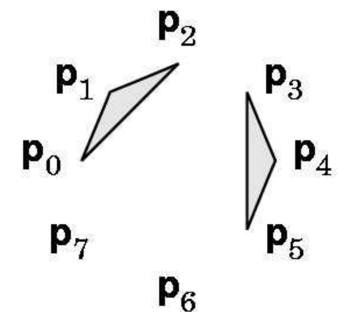
Types of Primitives

- `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.



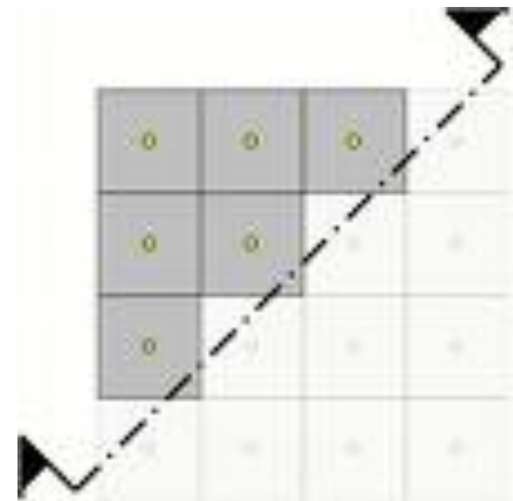






Rasterization

- After primitive assembly, primitives are converted into *fragments*, which are the part of a pixel that represents a single primitive.
- Then per-fragment operations are performed, which includes things such as texturing.



Buffers

- OpenGL composites the fragments and stores its final output in buffers.
- Buffers are 2D arrays of data, generally correlating to per-pixel information.
- Can use buffers to store the results of intermediate stages for use in later rendering passes.
  - You'll get to see this in project 3.
- The final frame buffer is output to the screen.
  - Note: this it not automatically cleared each frame. It must be manually cleared with `glClear`.

Buffers

- Examples of common uses of buffers:
  - Color buffers
    - Contain information about the color of pixel
  - Depth (Z) buffer
    - Stores depth information of each pixel, which is used to correctly draw closer objects in front of farther ones
  - Stencil buffer
    - Used for cropping with complicated shapes
  - Accumulation buffer
    - Used to store intermediate results for use in later passes.
- Buffers are simply arrays with some specified format; they can be used for any purpose.

Drawing Primitives

Drawing Primitives

- There are many ways to specify vertex attribute data and render primitives.
- We'll show you a few, though what you use it up to you.
- The methods here are the older, slower, but easier-to-use and better-supported methods.

glBegin/glEnd

- The simplest way to draw primitives is to use the OpenGL begin and end calls.
  - Attributes specified begin `glBegin/glEnd` pair.
  - Specify all vertex attributes, ending each vertex with `glVertex`.
- Can draw several of same primitive in one begin/end pair.

glBegin/glEnd

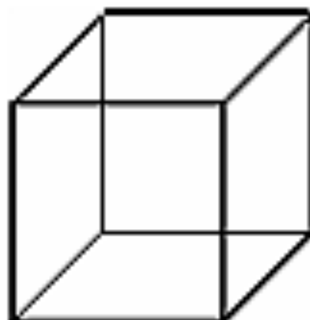
```
// Sample drawing function
void display_triangle() {
   glBegin( GL_TRIANGLES );
      glColor3f( 0.0f, 0.0f, 1.0f ); // sets color to blue
      glVertex2f( 0.0f, 0.0f ); // draw a vertex at 0,0
      glColor3f( 1.0f, 0.0f, 0.0f ); // sets color to red
      glVertex2f( 0.0f, 1.0f ); // draw a vertex at 0,1
      // this will use same color as previous vertex
      glVertex2f( 1.0f, 0.0f );
   glEnd();
}
```

glBegin/glEnd

- Pros:
  - Very simple to use for simple objects.
  - Can easily specify attributes.
- Cons:
  - Very cumbersome to specify many objects
  - Code is verbose.
  - Extremely slow.
    - Several function calls per-vertex, which add up really fast.
    - Have to send all vertex data from CPU to GPU every frame.

Vertex Arrays

- Major downside of glBegin/End: for anything but simple models, we need a ton 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.
  - This gets even worse for models with thousands of vertices!



Vertex Arrays

- OpenGL vertex arrays allow you to specify vertex data using arrays and few function calls.
  - Place all attribute data into an array.
  - Render the entire set of primitives at once.

Vertex Arrays

- Several ways to specify arrays:
  - `glVertexPointer` (and siblings such as `glNormalPointer`) to specify separate arrays for each attribute.
  - `glInterleavedArrays` to store all attributes in a single array.
  - Vertex array objects to store data on GPU.

position	normal	color	position	normal	color	position	normal	color

Vertex Arrays

- Several ways to draw primitives:
  - `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.

Vertex Arrays

```
// Sample code using vertex arrays
void display_triangle() {
  float vertices[] = { 1.0f, 0.0f, 0.0f,
                       0.0f, 1.0f, 0.0f
                       0.0f, 0.0f, 1.0 };
  glEnableClientState( GL_VERTEX_ARRAY );
  // specify where to get vertex data
  glVertexPointer( 3, GL_FLOAT, 0, vertices );

  unsigned int indices[] = { 0, 1, 2 };
  // this is the actual draw call
  glDrawElements( GL_TRIANGLES, 3,
                  GL_UNSIGNED_INT, indices );
  glDisableClientState( GL_VERTEX_ARRAY );
}
```

Vertex Arrays

- Pros:
  - Still quite easy to use.
  - Requires only a bit of setup.
  - Much faster than Begin/End.
- Cons:
  - For client-side vertex arrays, still pretty slow.
    - Still have to send all the vertex data from CPU to GPU every frame.

Display Lists

- Display lists provide a way for OpenGL to redraw arbitrary primitives with a single call.
- Display lists compile a set of commands that draw a particular object.
- Only work with completely static geometry.

Display Lists

```
int list;

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

// Sample drawing function
void display_callback(){
  glCallList( list ); // renders the compiled list
}
```

Display Lists

- Pros:
  - Very fast, sometimes fastest of any method.
- Cons:
  - Only works with unchanging geometry.
  - Can consume a lot of GPU memory.

Depth Sorting

- OpenGL uses a depth test to determine which primitives go in front of others.
  - By default, all primitives are drawn on top.
- Must explicitly enable `GL_DEPTH_TEST` to get depth testing.

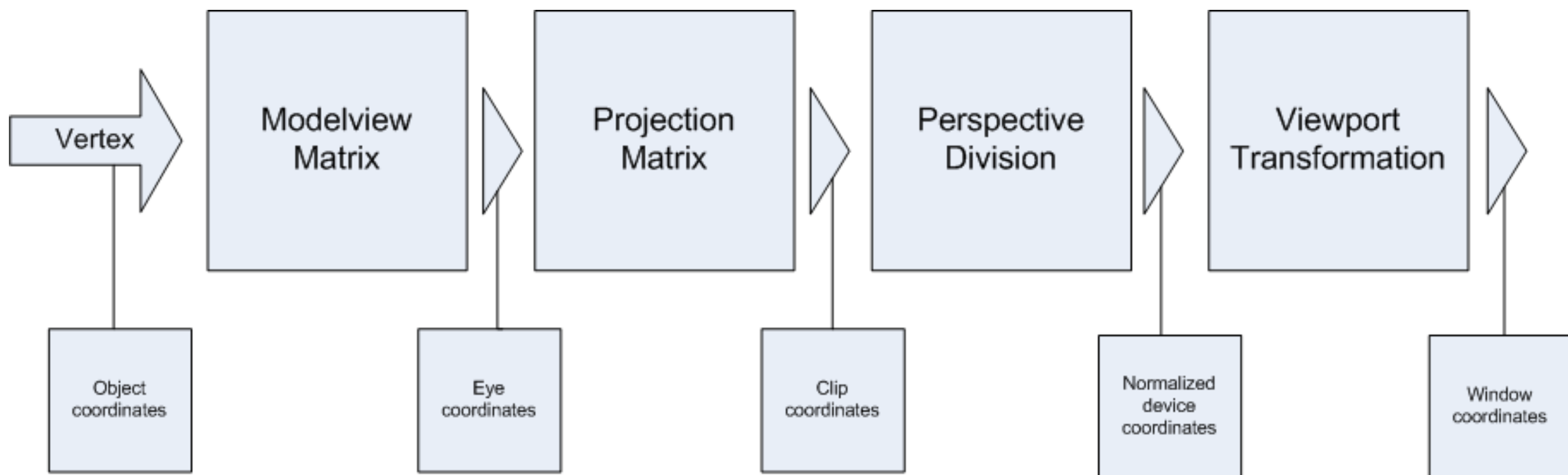
Transformations

Word of Caution

- Everything you will learn about in the next two sections is part of the *fixed-functionality* pipeline.
  - Matrices, materials, lighting, etc.
- This is the only thing GPUs supported before programmable shaders.
- With shaders, all of this is obsolete.
  - Fixed-functionality is just the “default” shader.
  - When using shaders, none of this stuff has any effect (unless you program the shader to use it).

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.



Matrix Modes

- Transformations in OpenGL are accomplished using matrices.
- OpenGL tracks two different matrices 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.

Basic Transformations

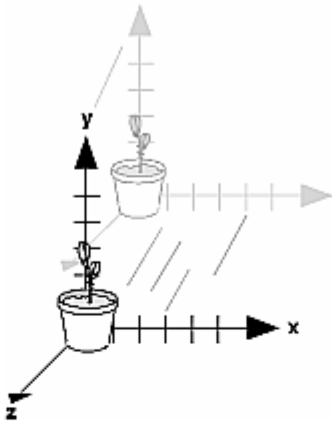
- 3 basic model transformations:
  - `glTranslate(T x, T y, T z)`
    - Translates an object by given *x*, *y*, and *z*.
  - `glRotate(T angle, T x, T y, T z)`
    - Rotates an object by given angle (in degrees) counterclockwise around the given vector(*x*, *y*, *z*).
  - `glScale(T x, T y, T z)`
    - Scales an object by given *x*, *y*, and *z*.
    - Note: this will mess up normals, so you must enable `GL_NORMALIZE` to counter it.

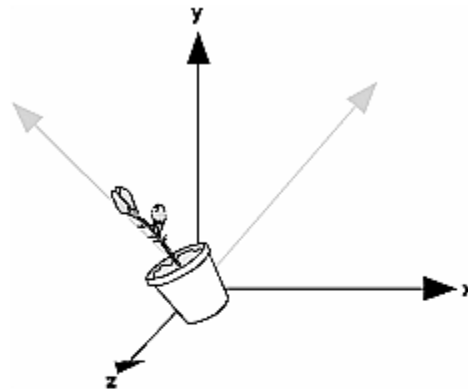
Model Transformations

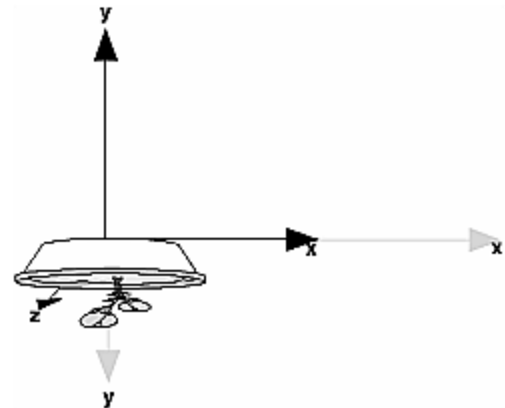
```
// Sample code showing matrix transformations
void display_object(){
   glMatrixMode( GL_MODELVIEW ); // set current matrix
   glLoadIdentity(); // Clears the matrix
   glTranslatef( 0.0f, 0.0f, -2.0f );
   glRotatef( 45.0f, 0.0f, 0.0f, 1.0f );
   glScalef( 2.0f, 2.0f, 2.0f );
   draw_object(); // our own drawing function
}
```

Order of Transformation

- 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).

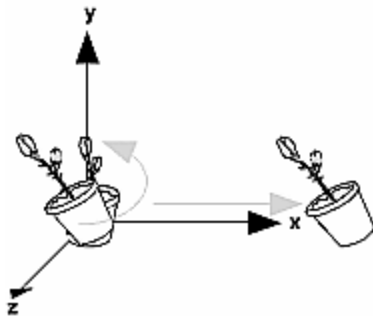




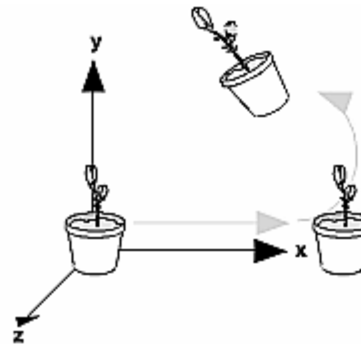


Order Matters!

- 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).



Rotate then Translate



Translate then Rotate

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.

Pushing and Popping

```
// 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?

Viewing

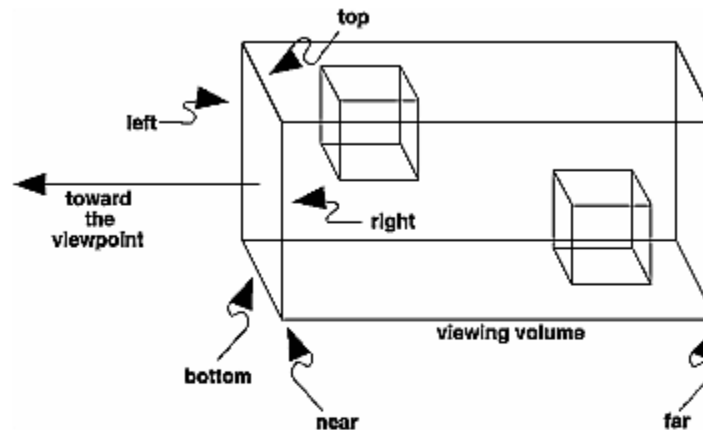
- Fortunately, there's a handy function already:
  - `gluLookAt( GLdouble e_x, GLdouble e_y, GLdouble e_z, GLdouble c_x, GLdouble c_y, GLdouble c_z, GLdouble u_x, GLdouble u_y, GLdouble u_z )`
    - e_x, e_y, and e_z specify the desired viewpoint
    - c_x, c_y, c_z specify some point along the desired 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).

Projection

- 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.

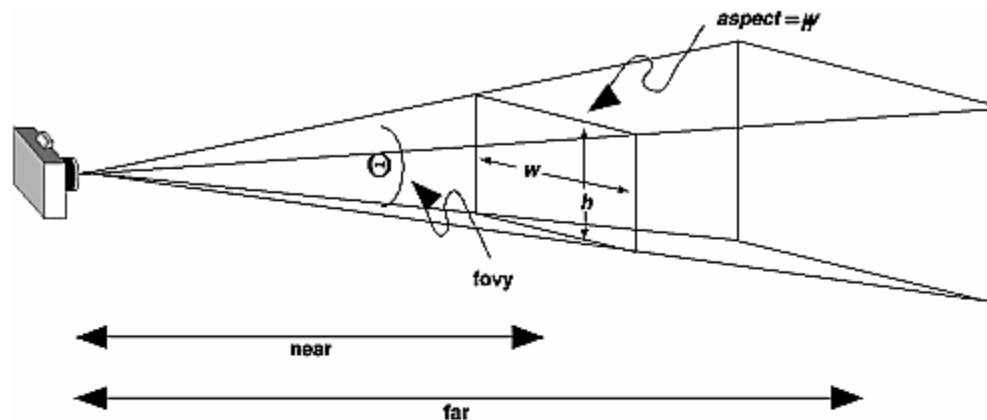
Orthographic Projection

- `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



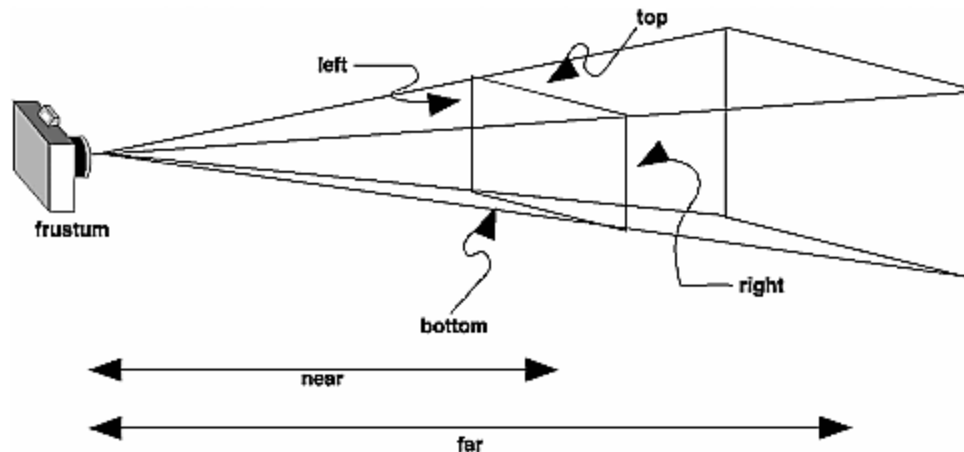
Perspective Projection

- `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



Perspective Projection

- `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

Lighting and Materials

- By default, OpenGL's fixed-function pipeline implements the Blinn-Phong Shading Model.
  - Developed by Bui Tuoung Phong in 1973 and modified by James Blinn in 1977.
- The model provides an approximation for how to represent color and lighting for both lights and object materials.

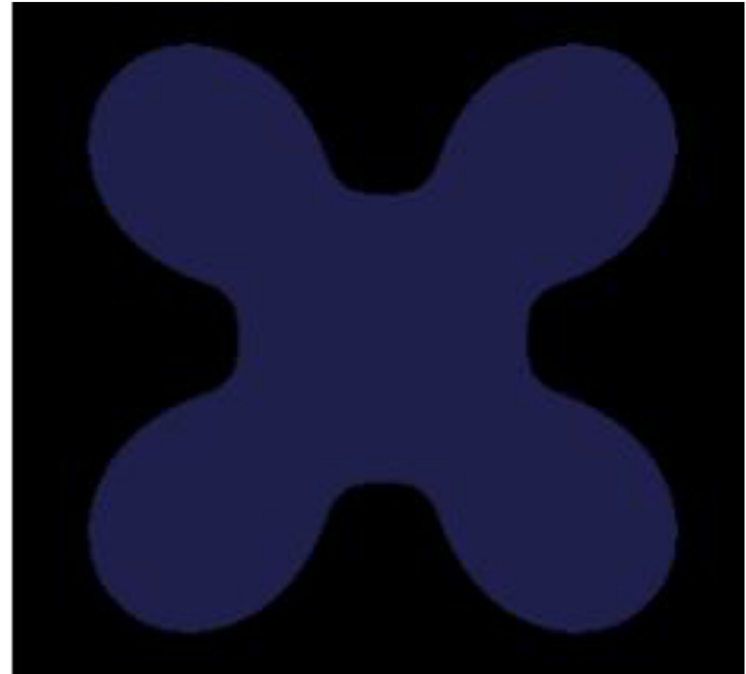
Blinn-Phong

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

- 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 of the lights
- Dot products: provide the dependence on the light-surface and reflection-viewer angles (this will be discussed during the lighting lecture)

Ambient Term

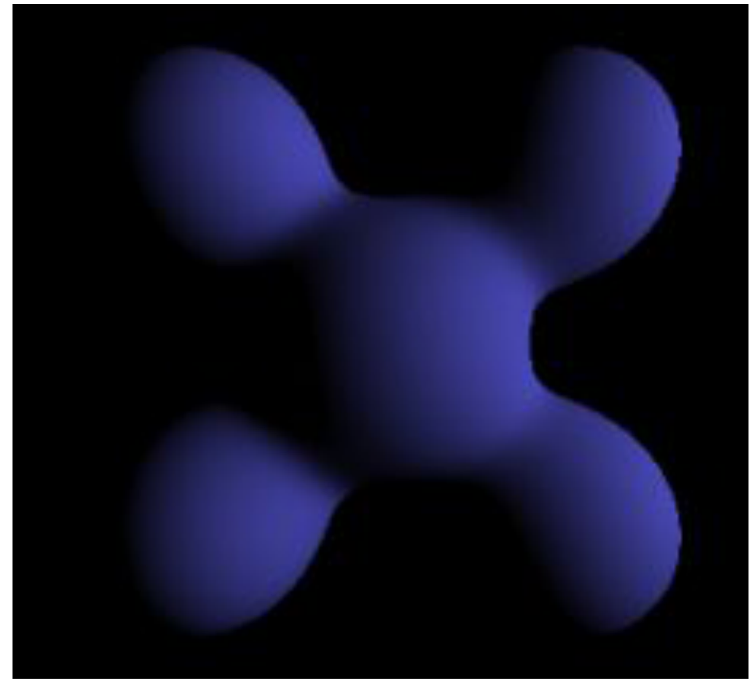
- The ambient term approximates the low level of light that is normally present everywhere in a scene (scattered by many objects before reaching the eye).
- Constant term; is applied equally to all points on the object



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

Diffuse Term

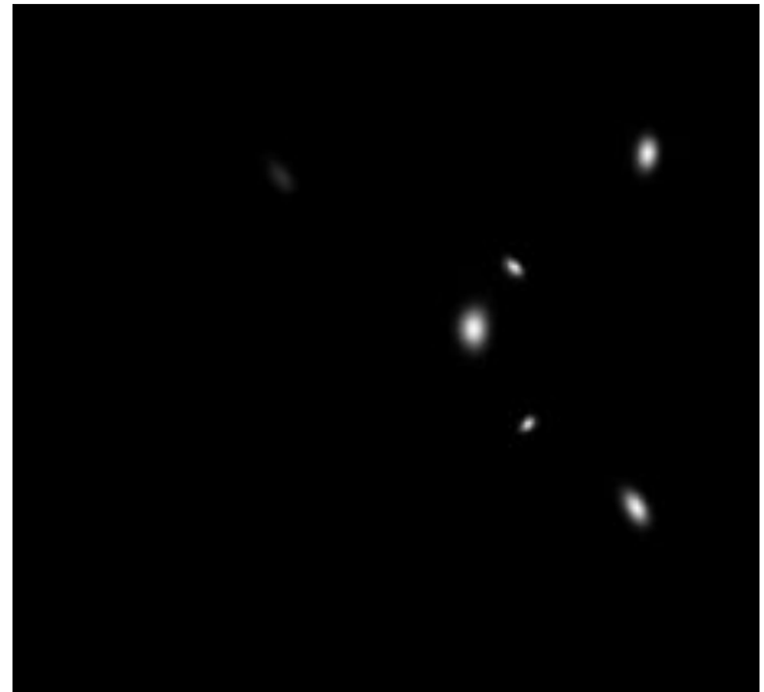
- The diffuse term approximations light scattered by objects with rough surfaces.
- Its intensity depends on the angle between the light source and the surface normal (*not* the direction to the viewer).



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

Specular Term

- 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.



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

Setting Lights

- 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)`, plus enabling specific lights.
- You also must set properties of each light:
  - `glLight`
    - Specify the particular light (e.g., `GL_LIGHT0`), the parameter of the light to set (e.g., `GL_POSITION`), and the value of the parameter.

Setting Materials

- Similarly, you must set material properties:
  - `glMaterial`
    - Specify what face to apply the material to (e.g., `GL_FRONT`), the parameter to set (e.g., `GL_DIFFUSE`), and the value for that parameter.

Setting Light and Materials

```
// 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 );
   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.).

Scene

Direct

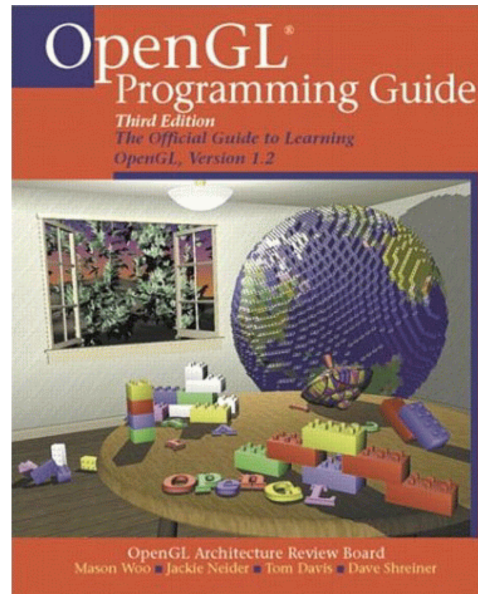
Global

Solutions?

- Be happy with Blinn-Phong.
  - Projects 1, 2
- Program the pipeline to behave differently using shaders.
  - Projects 3
  - 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 4
- Pick what you want (as long as it's real-time)
  - Project 5

The Red Book is Your Friend

- We haven't covered everything you need to know for the projects.
- The Red Book does cover everything.
  - So use it!

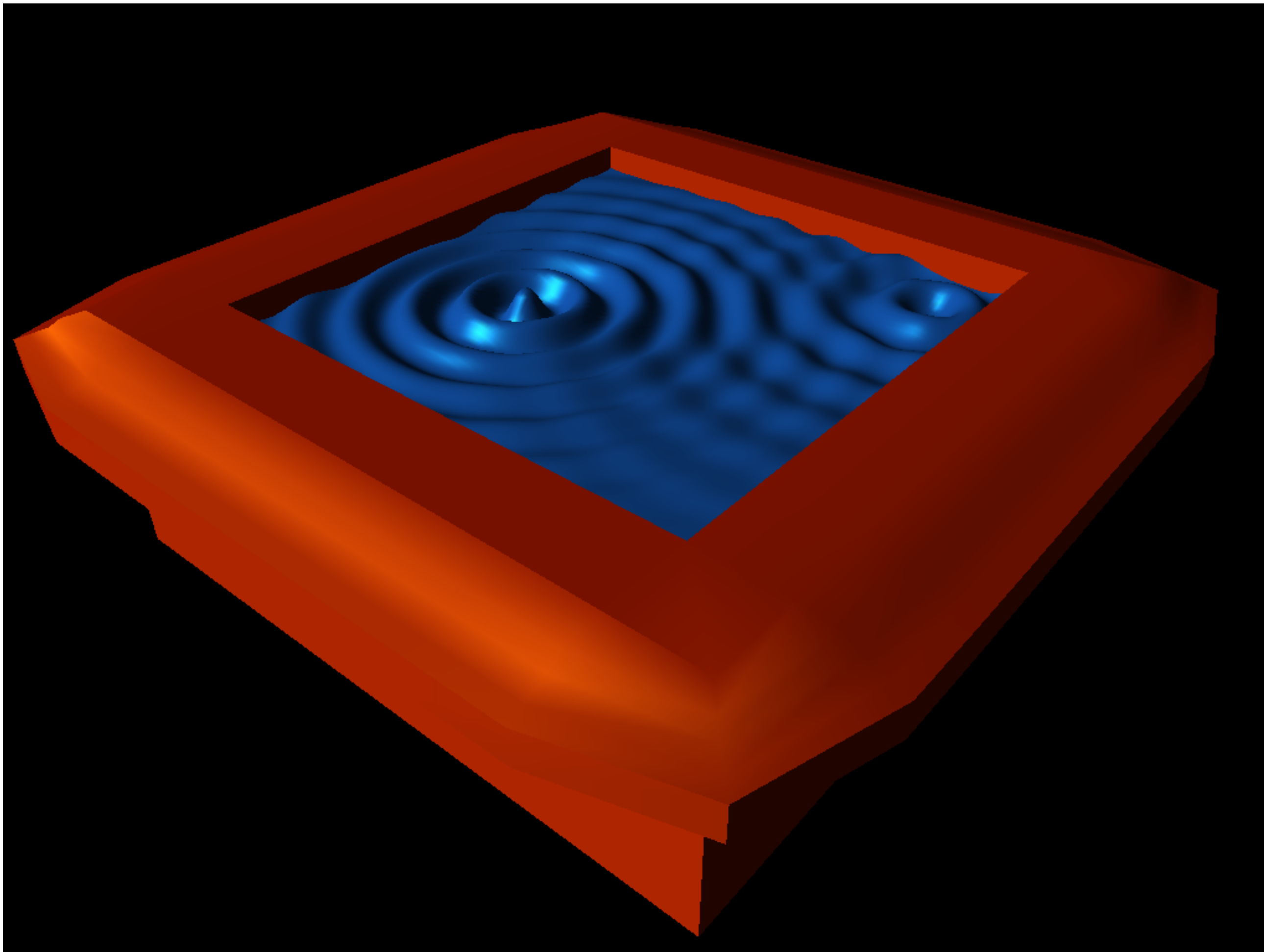


Project 1

Project: Basic OpenGL Rendering

- The first project involves using OpenGL to render a simple scene.
- The scene includes:
  - A triangle mesh.
    - Must compute normals for the mesh.
  - An animated heightmap.
    - Must create a mesh and compute normals.
  - Basic camera transformations.
  - Basic lighting and materials.

Project: Basic OpenGL Rendering



Questions?

Project 1 will be out later today!