OVERVIEW OF LIBRARIES
OPENGL CORE LIBRARY (OR BASIC LIBRARY)

• Hardware and platform independent
  • Specification that each platform must implement

• Provides functionality for specifying:
  • Graphics primitives
  • Attributes
  • Geometric and viewing transformation
  • …and more!

• However, because hardware/platform independent, it does NOT provide:
  • Input/output functionality (e.g., mouse and keyboard)
  • Display windows (depends on operating system / windowing system)
Every implementation of OpenGL also includes **OpenGL Utility (GLU)**

- Separate library that provides routines for:
  - Setting up viewing and projection matrices
  - Describing complex objects with line and polygon approximations
  - Displaying quadrics and B-splines (using linear approximations)
  - Surface rendering operations
  - Etc.

- Also hardware/platform independent
Arguably the most important thing OpenGL leaves out is creating a window.
Fortunately, there are many libraries that handle this for us:

- GLX (OpenGL Extension to the X Windows System)
- AGL (Apple GL)
- WGL (Windows-to-OpenGL)

You’ll notice, however, these are still tied to their respective window systems and/or operating systems...
OPENGL UTILITY TOOLKIT (GLUT)

- Developed by Mark Kilgard while he was working at SGI
  - Built originally to work with GLX
  - Ported to WGL (Windows) by Nate Robins
  - Also version for Apple as well
- Designed to allow programmers to create windows, access input devices, etc., without having to know the specific window system and/or OS
- Still copyrighted by Kilgard
- Admittedly, hasn’t been updated in a bit (last Windows port was 2001)
  - Open source version (this is still maintained) \(\rightarrow\) freeglut
SO WHY ARE WE USING GLUT?

• Simple and straightforward
• Platform-independent
• Easy to install

• (However, you can use freeglut instead if you prefer, since it should be compatible)

• This is not to say that GLUT is the only way to go:
  • GLUI
  • GLFW
  • ...and others: https://www.opengl.org/resources/libraries/windowtoolkits/
TELLING THEM APART

• Functions, constants, and data types will have a different prefix, depending on which library it comes from

  • **OpenGL → gl**
    • Examples: glBegin, glClear, glCopyPixels, GL_2D, GL_RGB, GLint, GLfloat, etc.

  • **GLU → glu**
    • Examples: gluLookAt, gluPerspective, etc.

  • **GLUT → glut**
    • Examples: glutCreateWindow, glutDisplayFunc, etc.
OPENGL VERSIONS

• OpenGL has gone through some significant changes over the years
  • For the full narrative, see: https://www.opengl.org/wiki/History_of_OpenGL

• OpenGL 1.X
  • Effectively the version in the Hearn-Baker book
  • Uses fixed-function pipeline only

• OpenGL 2.X
  • Introduced GLSL (OpenGL Shader Language)
  • Still used OpenGL 1.X code/functions
**OPENGL VERSIONS**

- **OpenGL 3.X**
  - The fixed function pipeline and most of the associated functions were declared *deprecated*:
    - `glBegin()`, `glEnd()`, `glVertex*`, the matrix stacks, etc.
    - Referred to as *legacy OpenGL* now
    - Many functions outright *REMOVED* after 3.1 and 3.2
  - Everything pretty much done with vertex and pixel shaders
    - Copy data you need to render to buffers on GPU
    - Have to manage your own matrices
  - Now have notions of **core** and **compatibility** OpenGL contexts
    - Implementations do not HAVE to support compatibility contexts, although NVIDIA and ATI are still doing so in Windows and Linux
- **OpenGL 4.5**
  - Most recent version at time of this slide deck
WHAT CAN I SUPPORT?

• If you downloaded GLEW, there’s a program they have called `glewinfo.exe`
  • Run it → this will create a text file called `glewinfo.txt`
  • Somewhere near the top it will tell you what OpenGL version your system supports:
    • Example: OpenGL version 4.4.0 NVIDIA 344.75 is supported
• In newer OpenGL versions, you need to manage matrices yourself
• Thankfully, the header-only library GLM makes this a LOT easier
  • Download it here: http://glm.g-truc.net/0.9.6/index.html
  • Just copy glm folder into your project folder (it’s the folder on the same level as the doc and test folders)
IS THIS THE SHAPE OF THINGS TO COME?

• In this course, we will attempt to cover both legacy OpenGL and the newer OpenGL model

• Reasons:
  • Legacy is a little easier to debug and understand
  • You’ll probably encounter legacy code floating around (and you’ll want to know how to deal with it)

• Note that GLSL has changed quite a bit over of the years, so some tutorials online will contradict each other
DISSECTING A BASIC OPENGL/GLUT PROGRAM
When one talks about OpenGL, a phrase that will be thrown around occasionally is an (or the) **OpenGL context**.

**OpenGL context** = all of the state information associated with an instance of OpenGL

- “Think of a context as an object that holds all of OpenGL; when a context is destroyed, OpenGL is destroyed.” -- [https://www.opengl.org/wiki/OpenGL_Context](https://www.opengl.org/wiki/OpenGL_Context)
  - Includes a default framebuffer, the thing we are going to draw on
- Local to a particular execution process → often, one application has one OpenGL context
  - However, a process can make multiple contexts
  - OpenGL commands act on the **current context**, so you have to switch to the one you want
- Contexts can be completely independent of each other → changes to one context do not affect the other
  - However, it is possible to share some objects (like textures)
- Most importantly, **OpenGL commands DO NOTHING until there is an OpenGL context to act on**
If we were using Microsoft Windows window system (WGL), then we would need the following:

- `#include <windows.h>`
- `#include <GL/gl.h>` // Gives us OpenGL core
- `#include <GL/glu.h>` // Gives us GLU

However, since we’re using GLUT, we can just do:

- `#include <GL/glut.h>`
- ...since glut.h will include gl.h and glu.h for us
INITIALIZATION AND WINDOW/CONTEXT CREATION

- **glutInit(&argc, argv)**
  - This initializes GLUT
  - We’re passing in the command-line parameters, because GLUT can process these
    - *Recall*: `int main (int argc, char **argv)`

- **glutCreateWindow(“My OpenGL Program that is Supremely Awesome”)**
  - States that a display window is going to be created and defined
  - After calling, we have an OpenGL context to work with!
    - However, window will NOT display until we call glutMainLoop() later
  - Caption for the window will read: “My OpenGL Program that is Supremely Awesome”
SETTING WINDOW POSITION AND SIZE

- **glutInitWindowPosition(x, y)**
  - Sets the initial position of the top-left corner of the window to (x,y)
  - Coordinates start at top-left corner of screen

- **glutInitWindowSize(width, height)**
  - Sets initial window size
  - NOTE: The height does NOT include the menu bar!
SETTING WINDOW CHARACTERISTICS

- `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)`
  - This sets options for the window
  - Different options can be ORed together
    - **GLUT_DOUBLE** = use double-buffering (GLUT_SINGLE = single buffer)
    - **GLUT_RGB** = use RGB color mode (red, green, and blue)
    - **GLUT_DEPTH** = add a depth buffer (z-buffer)
  - Default options are GLUT_RGB and GLUT_SINGLE
CALLBACK FUNCTIONS AND GLUT

- When certain events occur, GLUT will call a certain function ➔ callback functions
  - We need to tell GLUT to call our functions for the events we want
  - Our function is “registered” as the function to call when a certain thing happens
  - Note, however, these functions must have a certain prototype for them to work
DRAWING CALLBACK FUNCTIONS

• **glutDisplayFunc**(myDrawFunction)
  • *Prototype:* void myDrawFunction(void)
  • When the screen needs to be redrawn, call myDrawFunction()
  • NOTE: myDrawFunction() will only be called if the window HAS to be redrawn  \( \rightarrow \) e.g., is minimized then maximized

• **glutIdleFunc**(myDrawFunction)
  • *Prototype:* void myDrawFunction(void)
  • When the window is doing nothing (i.e., idling), call myDrawFunction()
  • If myDrawFunction() is your drawing function, you will repeatedly draw on the screen

• You need to define glutDisplayFunc(), even if you also define glutIdleFunc()
RESHAPE CALLBACK FUNCTION

- **glutReshapeFunc(reshape)**
  - *Prototype*: `void reshape(int width, int height)`
  - Called when the window is resized
    - Also called when the window is first created.
  - Width and height are in pixels
ENTER THE MATRIX...

• Our display window is not yet on the screen
• After all our initialization calls, we need to make one more GLUT call to get everything rolling...

• `glutMainLoop()`
  • Displays initial graphics
  • Puts program in infinite loop that:
    • Checks for input
    • Redraws screen (if necessary)
    • Deals with any other events GLUT is looking for
  • We will NEVER return from this call!
    • It therefore must be the last line of your program!
A BASIC LEGACY OPENGL PROGRAM
SETTING UP OPENGL

• We’ve done all of the window-system stuff

• However, there’s still more to do to make it a complete OpenGL program

• It is assumed we have two basic functions:
  • `init()`
    • A function that sets up some OpenGL-related things
    • Called AFTER `glutCreateWindow()` but BEFORE `glutMainLoop()`
  • `myDrawingFunction()`
    • Function that will be called every time the screen must be redraw

• We will use these two functions for both our legacy and new OpenGL examples
SETTING UP THE VIRTUAL CAMERA

• For starters, we’ll set up a very simple **2D orthographic/orthogonal camera**
  • We’ll do all this in the `init()` function

• We need to change the projection of the camera, so:
  • `glMatrixMode(GL_PROJECTION)`
    • Switches to the OpenGL's matrix stack for the projection matrix (we’ll explain this later)
  • `glLoadIdentity()`
    • Loads the identity matrix → basically makes sure we’re starting from scratch
  • `gluOrtho2D(0, 200, 0, 150)`
    • Sets up a 2D orthographic projection
    • View volume in WORLD coordinates going from 0 to 200 in X and 0 to 150 in Y
  • `glMatrixMode(GL_MODELVIEW)`
    • Switches (back) to OpenGL's matrix stack for model-view transformation
  • `glLoadIdentity()`
    • Just to be safe...
RESHAPE FUNCTION

• In our reshape callback, we will run the following:

  • `glViewport(0, 0, width, height);`
    • This resizes the “viewport” (i.e., the area on the window we are rendering to)
    • In this case, we want to render to the entire window size
    • NOTE: In this (and other) simple 2D examples, our VIEW VOLUME is still 200 units wide and 150 high
      • So, if we resize the window → do NOT have one-to-one correspondence with world and pixel coordinates anymore!
ANATOMY OF A DRAW LOOP

• In our drawing function (draw loop), we will do the following basic steps:
  • Clear the window
  • Draw stuff
  • Swap buffers (if double-buffering)
  • Sleep a bit
CLEARING THE WINDOW: SETTING THE COLOR

• `glClearColor(red, green, blue, alpha)`
  • Sets what color we will clear our screen with
  • Values range from [0, 1]
    • *Example:* set background color to red → `glClearColor(1.0, 0.0, 0.0, 0.0);`
  • This does NOT clear the screen; it only sets the color we will use to clear the screen
  • Usually call in our init() function
CLEARING THE WINDOW: ACTUALLY CLEARING IT

- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
  - Clears out buffers specified → additional buffers to clear can be ORed together
  - `GL_COLOR_BUFFER_BIT = clear color buffer (with color set by glClearColor())`
  - `GL_DEPTH_BUFFER_BIT = clears z-buffer (we’ll get to this later)`
To set the current color to draw in, use `glColor*`

* depends on (number of values) + (data type of values).
  * Example: `glColor3f(0.0, 1.0, 0.0) →` 3 floating point values (red, green, blue)

Color remains the same until you change it
To draw a line going from (180,15) to (10, 145) in WORLD coordinates:

- `glBegin(GL_LINES)`
  - Specifies we’re drawing lines
  - For every 2 vertex calls, we will draw a line
- `glVertex2i(180, 15);`
  - There is a vertex at (180,15)
  - Note the “2i” → 2 integers
- `glVertex2i(10, 145);`
  - There is a vertex at (10, 145)
- `glEnd();`
  - We are done

- By convention, you usually indent the code inside the `glBegin() / glEnd()` “block”
- The line will be the color that we set with `glColor*` before
SWAPPING BUFFERS

• If we were doing single-buffering, we would have to call glFlush() after we were done to make sure everything gets rendered

• **Double-buffering**
  • Two buffers (one for drawing on and one for display)
  • Draw on first buffer while showing contents of second buffer, swap which buffer does what
  • Prevents flickering (you don’t see things in the middle of being drawn)

• **glutSwapBuffers()**
  • Swaps the buffers 😊
SLEEPING ON THE JOB

• In your drawing function, you will probably want to sleep a bit
  • Doesn’t hog system resources by trying to draw as fast as possible
  • If you want to be fancy, you can time how long it takes to render and then fill the remaining time with sleep

  *Example: `std::this_thread::sleep_for(std::chrono::milliseconds(15));`

  • Sleep for 15ms; this will give us about ~60 fps
    • Obviously fps will be lower if what we’re drawing is more complicated
  • NOTE: This uses C++11 features from the standard C++ library!
    • Otherwise you have to use OS-specific sleep functions
COMPLETE INIT FUNCTION

• `glClearColor(1.0, 1.0, 1.0, 0.0); // Set clear color to white`

• `glMatrixMode(GL_PROJECTION)`
• `glLoadIdentity()`
• `glMatrixMode(GL_MODELVIEW)`
• `glLoadIdentity()`
• `gluOrtho2D(0, 200, 0, 150)`
COMPLETE DRAWING FUNCTION

- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

- `glColor3f(0.0, 1.0, 0.0);`

- `glBegin(GL_LINES)`
  - `glVertex2i(180, 15);`
  - `glVertex2i(10, 145);`
- `glEnd();`

- `glutSwapBuffers();`

- `std::this_thread::sleep_for(std::chrono::milliseconds(15));`
A BASIC OPENGL PROGRAM WITH SHADERS
LIFE IS TRADE-OFFS

• The newer OpenGL model is great for the following reasons:
  • Flexible and very controllable
  • More efficient
  • Better suited to today’s graphics hardware

• The downsides:
  • Rendering something really simple (like a line) is a bit of an ordeal
  • “If you wish to make an apple pie from scratch, you must first invent the universe.” – Carl Sagan
Since we’ll be using GLEW and GLM (and shader loading code from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/), the following includes will be needed:

- #include <iostream>
- #include <fstream>
- #include <algorithm>
- #include <vector>
- #include <string>
- #include <cmath>
- #include <thread>
- #include <chrono>
- #include <GL/glew.h>  // MUST be included before GLUT!
- #include <GL/glut.h>
- #include “glm/glm.hpp”  // NOTE: Need to compile, hence the quotes
- #include “glm/gtc/matrix_transform.hpp”
INITIALIZING GLEW

- We will use GLEW so we can use shaders in OpenGL
- After glutCreateWindow(), call the following:

  ```cpp
  GLenum err = glewInit();
  if (GLEW_OK != err) {
    cout << "ERROR: GLEW could not start: " << glewGetErrorString(err) << endl;
    return 1;
  }
  cout << "GLEW initialized; version " << glewGetString(GLEW_VERSION) << endl;
  ```

- If glew fails to initialize, program will exit.
- Otherwise, it will print the current GLEW version.
ANATOMY OF AN OPENGL PROGRAM

- At bare minimum, we’ll need at least three files:
  - Main C++ file
    - Your main OpenGL/GLUT program
  - Vertex shader file
  - Fragment shader file
- The vertex + fragment shaders are called a shader program
CHANGES FROM LEGACY OPENGL

• The major changes from legacy OpenGL:
  • Must handle matrices/transformations ourselves
    • This means we will construct a matrix for the model, view, and projection transformations and pass it into the vertex shader
  • Need to construct buffers to hold data
    • In this example, for each object, we will have a buffer for vertex (position) data and a buffer for color data
    • These buffers will be referred to as “Vertex Attribute” buffers or arrays
VERTEX ARRAY OBJECT

• An object we must create AFTER we get an OpenGL context but BEFORE any OpenGL calls is a Vertex Array Object
  • Effectively manages all of buffers we will make in this program
• Code:
  • GLuint VertexArrayID;
  • glGenVertexArrays(1, &VertexArrayID);
  • glBindVertexArray(VertexArrayID);
LOADING/COMPILING/USING SHADER CODE

- Shader code is loaded from text files and compiled at run-time.
- The code for loading the vertex and fragment shaders is pretty standard, and plenty of examples of it can be found on the web.
- When it’s done, however, we will have an ID for our shader program.
  - Example: Gluint programID

- `glUseProgram(programID)`
  - Tells OpenGL to use the shader program identified by “programID” to render anything afterwards.
  - To turn off the shader program: `glUseProgram(0)`
VARIABLES IN SHADER PROGRAMS

• **Uniform** variable = variable in shader program that does NOT change inside a render call
  • I.e., rendering an object → model transform remains the same

• We need to get the identifier for the model-view-projection transform matrix in the vertex shader
  • *In vertex shader:*
    • uniform mat4 MVP;

• ```Gluint matrixID = glGetUniformLocation(programID, “MVP”);```
  • Gets identifier for uniform variable “MVP” in shader code

• ```
uniform mat4 MVP;
```
CREATING A LINE

- First, we need to create two buffers (one for the vertex positions and one for the color information) and fill them with data:
  - GLfloat *vertices = new GLfloat[3 * 2];
  - GLfloat *colors = new GLfloat[3 * 2];
  - vertices[0] = 180;  // First vertex (180, 15, 0)
  - vertices[1] = 15;
  - vertices[2] = 0;
  - vertices[3] = 10;  // Second vertex (10, 145, 0)
  - vertices[5] = 0;
  - colors[0] = colors[3] = 0.0f;  // Color is green for both vertices
  - colors[2] = colors[5] = 0.0f;
CREATING AN ARRAY BUFFER FOR VERTICES

• `glGenBuffers(1, &vertex_buffer);`
  • Create a buffer identifier → store it in vertex_buffer

• `glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);`
  • Bind this buffer as the current array buffer we’re working with

• `glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3 * 2, vertices, GL_STATIC_DRAW);`
  • (Re)create buffer data and COPY data from vertices to that buffer
  • NOTE: Data is COPIED, so changes to vertices after this will NOT affect the OpenGL buffer data!
  • Last parameter = “usage hint”
    • GL_STATIC_DRAW = buffer data will not be updated very often
    • Other options like GL_DYNAMIC_DRAW, etc. → will determine where buffer data gets stored
CREATING AN ARRAY BUFFER FOR COLOR

- glGenBuffers(1, &color_buffer);
- glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
- glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3 * 2, colors, GL_STATIC_DRAW);
INIT() FUNCTION

- `glClearColor(1.0, 1.0, 1.0, 0.0);`
- `programID = loadShaders("VerySimpleVertexShader.vert", "VerySimpleFragmentShader.frag");`
- `matrixID = glGetUniformLocation(programID, "MVP");`
- `GLfloat *vertices = new GLfloat[3 * 2];`
- `GLfloat *colors = new GLfloat[3 * 2];`
- `vertices[0] = 180;`
- `vertices[1] = 15;`
- `vertices[2] = 0;`
- `vertices[3] = 10;`
- `vertices[5] = 0;`
- `colors[0] = colors[3] = 0.0f;`
- `colors[2] = colors[5] = 0.0f;`
- `glGenBuffers(1, &vertex_buffer);`
- `glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);`
- `glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3 * 2, vertices, GL_STATIC_DRAW);`
- `glGenBuffers(1, &color_buffer);`
- `glBindBuffer(GL_ARRAY_BUFFER, color_buffer);`
- `glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 3 * 2, colors, GL_STATIC_DRAW);`
- `delete[] vertices;`
- `delete[] colors;`
MANAGING MATRICES: PROJECTION MATRIX

• We will use the GLM library to make matrix management easier...
• The following code we will put in our draw loop

```cpp
glm::mat4 Projection = glm::ortho(0.0f, 200.0f, 0.0f, 150.0f);
```
• Creates a 2D orthographic projection matrix from (0,200) to (0,150)
MANAGING MATRICES: VIEW MATRIX

```cpp
glm::mat4 View = glm::lookAt(
    glm::vec3(0, 0, 0), // Camera is at (0,0,0), in World Space
    glm::vec3(0, 0, -1), // and looks at (0,0,-1)
    glm::vec3(0, 1, 0)  // Head is up
);
```

• Similar to gluLookAt() function from legacy OpenGL
MANAGING MATRICES: MODEL MATRIX

- `glm::mat4 Model = glm::mat4(1.0f);`
  - Just making an identity matrix, because we don’t want to do anything special
MANAGING MATRICES: PUTTING IT ALL TOGETHER

- \texttt{glm::mat4 MVP = Projection * View * Model;}
  - Create our Model-View-Projection matrix
  - Keep in mind, given a matrix M that transforms a vector V to vector U:
    - \( U = M \cdot V \)
  - So, the matrices are applied from RIGHT to LEFT (so Model, then View, then Projection)
  - Keep in mind, if we change the Model matrix, we will have to recompute MVP
SENDING THE MATRIX TO THE SHADER

• First, we turn on the shader program
  • `glUseProgram(programID);`

• Then, we send the matrix to the shader using the MVP uniform variable
  • `glUniformMatrix4fv(matrixID, 1, GL_FALSE, &MVP[0][0]);`
    • Takes ID of variable, number of items (matrices) to send, whether it’s transposed or not, and a pointer to the data
    • 4fv → 4 = 4x4 matrix, f = floating point, v = passing in array data
MATRIX CODE IN DRAW LOOP

```cpp
glm::mat4 Projection = glm::ortho(0.0f, 200.0f, 0.0f, 150.0f);

glm::mat4 View = glm::lookAt(
    glm::vec3(0, 0, 0), // Camera is at (0,0,0), in World Space
    glm::vec3(0, 0, -1), // and looks at (0,0,-1)
    glm::vec3(0, 1, 0)  // Head is up
);

glm::mat4 Model = glm::mat4(1.0f);

glm::mat4 MVP = Projection * View * Model;

glUseProgram(programID);

glUniformMatrix4fv(matrixID, 1, GL_FALSE, &MVP[0][0]);
```
We will pass in the vertex information as vertex attribute arrays

- Position data → vertex attribute array 0
- Color data → vertex attribute array 1

NOTE: Could choose something other than 0 and 1

- glEnableVertexAttribArray(0);
- glEnableVertexAttribArray(1);
  - Enables vertex attribute arrays 0 and 1
DRAWING A LINE: POSITION DATA

We must tell OpenGL how to interpret the buffer data; we’ll start with the position data in vertex_buffer:

```c
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);

glVertexAttribPointer(0,
    // Attribute array 0
  3,
    // Number of components per vertex (3 = 3D point)
  GL_FLOAT,    // Type of data
  GL_FALSE,    // Normalize data?
  0,           // Stride → if 0, assumes data is tightly packed
  (void*)0,    // Array buffer offset → if 0, start at beginning
);
```
DRAWING A LINE: COLOR DATA

Do the same with the color data, only this time it will be Vertex Attribute Array 1

```c
glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
glVertexAttribPointer(1,    // Attribute array 1
    3,    
    GL_FLOAT,    
    GL_FALSE,    
    0,    
    (void*)0    
);
```
DRAWING A LINE: FINALLY...

- **glDrawArrays(GL_LINES, 0, 2);**
  - Renders the data passed in with the attribute buffers
  - **Parameters:**
    - Type of primitives to render
    - First index
    - Number of indices/vertices
DRAWING A LINE: CLEANING UP

• Unbind buffer and disable arrays
  • `glBindBuffer(GL_ARRAY_BUFFER, 0);`
  • `glDisableVertexAttribArray(1);`
  • `glDisableVertexAttribArray(0);`

• Turn off shader program (unless we have other objects we want to render)
  • `glUseProgram(0);`
DRAWING LOOP

- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
- `glm::mat4 Projection = glm::ortho(0.0f, 200.0f, 0.0f, 150.0f);`
- `glm::mat4 View = glm::lookAt(`
  - `glm::vec3(0, 0, 0),`
  - `glm::vec3(0, 0, -1),`
  - `glm::vec3(0, 1, 0)`
  `);`
- `glm::mat4 Model = glm::mat4(1.0f);`
- `glm::mat4 MVP = Projection * View * Model;`
- `glUseProgram(programID);`
- `glUniformMatrix4fv(matrixID, 1, GL_FALSE, &MVP[0][0]);`
- `glEnableVertexAttribArray(0);`
- `glEnableVertexAttribArray(1);`
- `glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);`
- `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,0, (void*)0);`
- `glBindBuffer(GL_ARRAY_BUFFER, color_buffer);`
- `glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,0, (void*)0);`
- `glDrawArrays(GL_LINES, 0, 2);`
- `glBindBuffer(GL_ARRAY_BUFFER, 0);`
- `glDisableVertexAttribArray(1);`
- `glDisableVertexAttribArray(0);`
- `glUseProgram(0);`
- `glutSwapBuffers();`
- `std::this_thread::sleep_for(std::chrono::milliseconds(15));`
#version 330 core

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;
uniform mat4 MVP;
out vec3 fragmentColor;

void main() {
    vec4 v = vec4(position, 1.0);
    gl_Position = MVP * v;
    fragmentColor = color;
}
**VERTEX SHADER CODE: EXPLAINED**

- `#version 330 core`
  - Using OpenGL 3.3 core
- `layout(location = 0) in vec3 position;`
- `layout(location = 1) in vec3 color;`
  - The position and color data for each vertex were passed in as **Vertex Attribute Arrays**
  - `layout(location = N) → Vertex Attribute Array N`
  - `vec3` = vector of 3 components
  - `in` = input variable
  - Names in code will be “position” and “color” (although we could have called them anything)
**VERTEX SHADER CODE: EXPLAINED**

- **uniform mat4 MVP;**
  - Model-View-Projection matrix
  - **uniform** = not modified within a render call (i.e., when `glDrawArrays()` is called)
  - **mat4** = 4x4 matrix

- **out vec3 fragmentColor;**
  - Per-vertex color
  - **out** = output variable
  - Will match input variable in fragment shader
VERTEX SHADER CODE: EXPLAINED

- void main() {
  - vec4 v = vec4(position, 1.0);
  - gl_Position = MVP * v;
    - Transforms each vertex using Model-View-Projection matrix
    - gl_Position = output position of each vertex (assumed in normalized device coordinates)
  - fragmentColor = color;
    - Just set output color to input color (values will be interpolated in fragment shader)
  - }

#version 330 core

// Interpolated values from the vertex shaders
in vec3 fragmentColor;

// Output color for each fragment
out vec3 color;

void main() {
    color = fragmentColor; // Do nothing to color; just output interpolated color
}
PROGRAM OUTPUT

• Same as before
REFERENCES

• The code in these slides is taken in part from the following sources:
  • Page 80 as well as Chapter 11 of “Computer Graphics with OpenGL” (3rd edition) by Hearn and Baker.
  • The following tutorial on OpenGL 2.0 and onwards: http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html
  • Another tutorial on OpenGL using shaders: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/