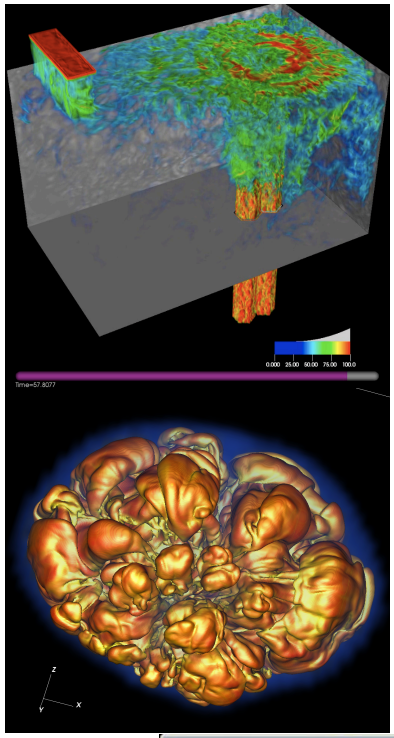
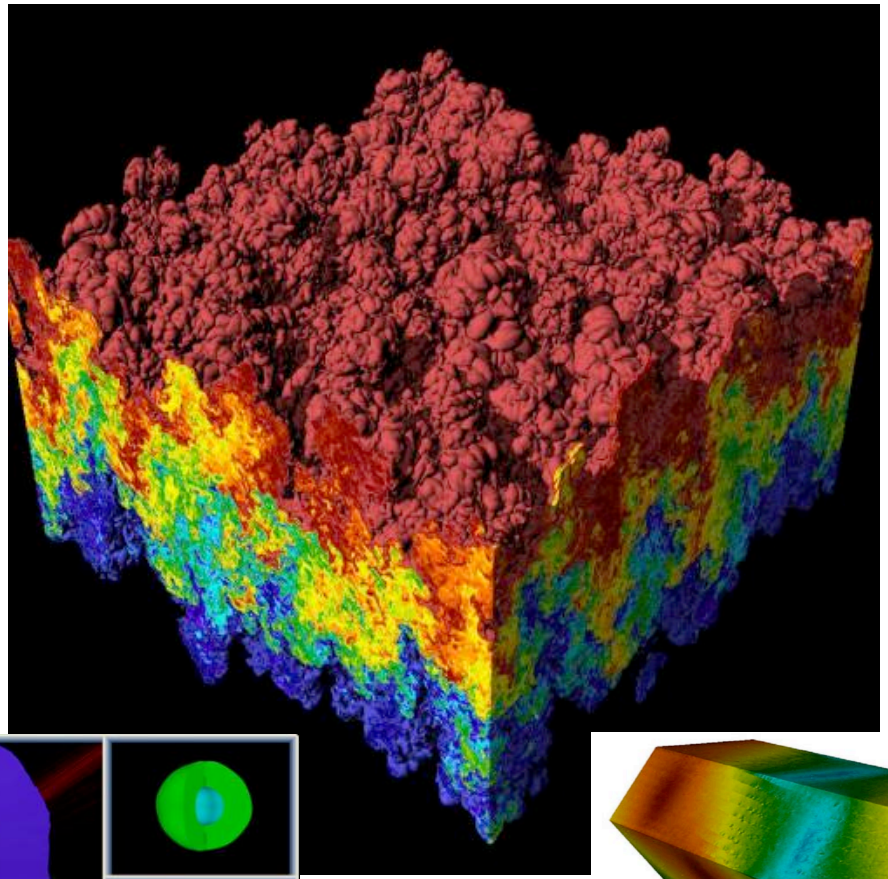
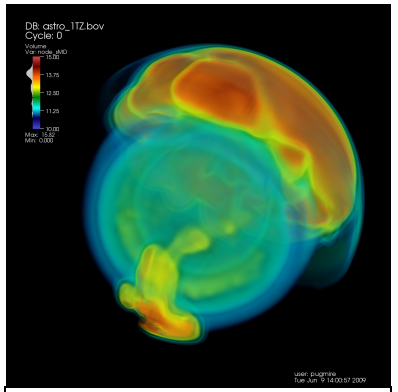
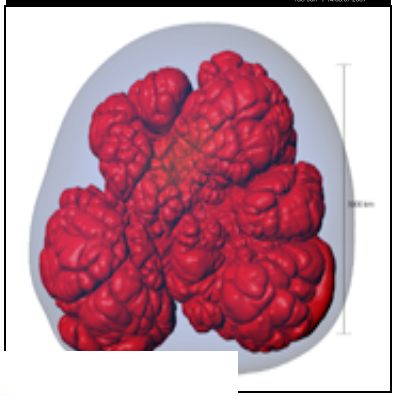
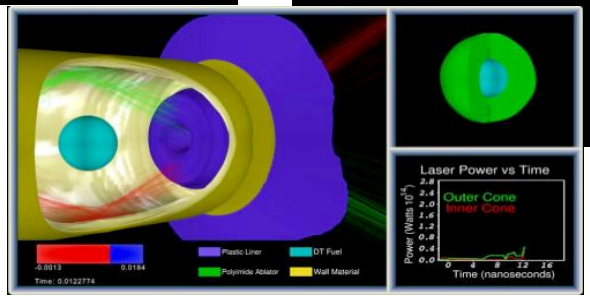
CIS 441/541: Intro to Computer Graphics
Lecture 9: OpenGL

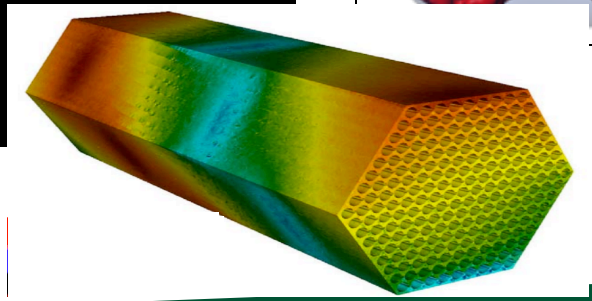












May 4, 2021

Hank Childs, University of Oregon

Current Plan (1/2)

Week	Sun	Mon	Tues	Weds	Thurs	Fri	Sat
5			Lec 7 (shading), ~~1F assigned,~~ 1E due		Lec 8 (finish shading, GL), ~~2A assigned~~ 1F assigned		
6		~~1F due~~	Lec 9 (GL), ~~2B assigned~~ 2A assigned (sort of)	1F due	~~2B assigned~~ ~~Discussion of final projects /~~ More GL ~~Quiz 3~~		~~2A due~~
7			~~Lec 11 – ray tracing~~ Even more GL 2B assigned 2A due		~~More~~ discussion of final projects Quiz 3	~~2B due~~	

Quiz 3

- Old: Thursday of Week 6 (in two days)
- Old: on matrices
- New: Thursday of Week 7 (in nine days)
- New: on Phong shading

Current Plan (2/2)

- Early part of Week 8: 2B due
- Rest of Week 8 -> Week 10 → you work on final projects
- Lectures will be on misc. topics in graphics, esp. in support of final projects
- ~~Quiz 3 (Week 6): likely on matrices~~
- Quiz 3 (Week 7): Phong shading
- Quiz 4 (Week 8): likely on GL
- Quiz 5 (Week 10): likely on topics in final weeks

Office Hours

Published | Edit

How to access Office Hours

Hank Childs

Apr 4 at 2:02pm

All Sections

Hi Everyone,

We currently have an asymmetry for accessing Hank and Abhishek's Office Hours.

As of now, Abhishek's are always at: COVERED UP (THIS IS POSTED ONLINE)

And Hank's are accessible via the Zoom Meetings area in Canvas.

Let's chat on Tuesday about the most standard way to do this.

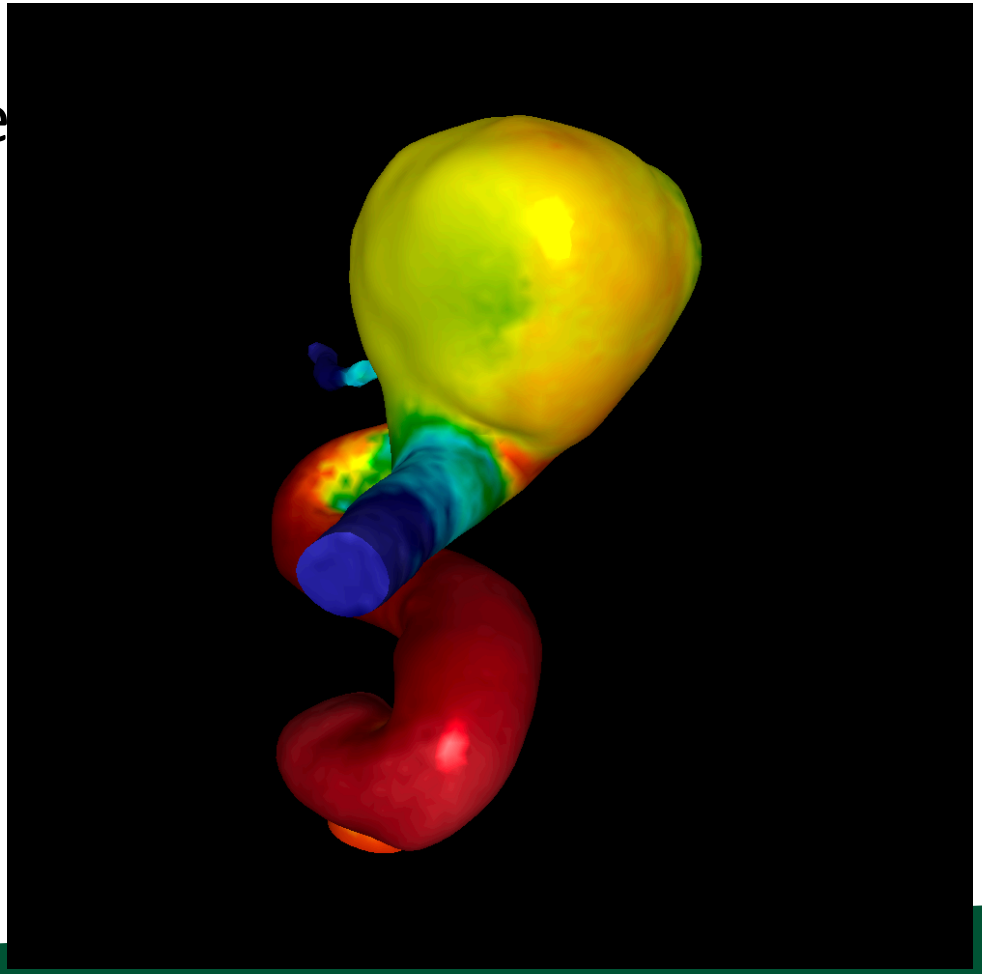
Finally, here is the OH schedule again:

Monday (Abhishek): 10am-11am
Tuesday (Abhishek): 945am-1045am
Wednesday (Hank): 230pm-330pm
Thursday (Abhishek): 945am-1045am

Best,
Hank

Project #1F (8%), Wed May 5th

- Goal: add shading, movie
- Extend your project1E code
- Important:
- add #define NORMALS
- Download new file, update to new file



Changes to data structures

```
class Triangle
{
  public:
      double X[3], Y[3], Z[3];
      double colors[3][3];
      double normals[3][3];
};
```

→reader1e.cxx will not compile (with #define NORMALS) until you make these changes

→reader1e.cxx will initialize normals at each vertex

More comments (1/3)

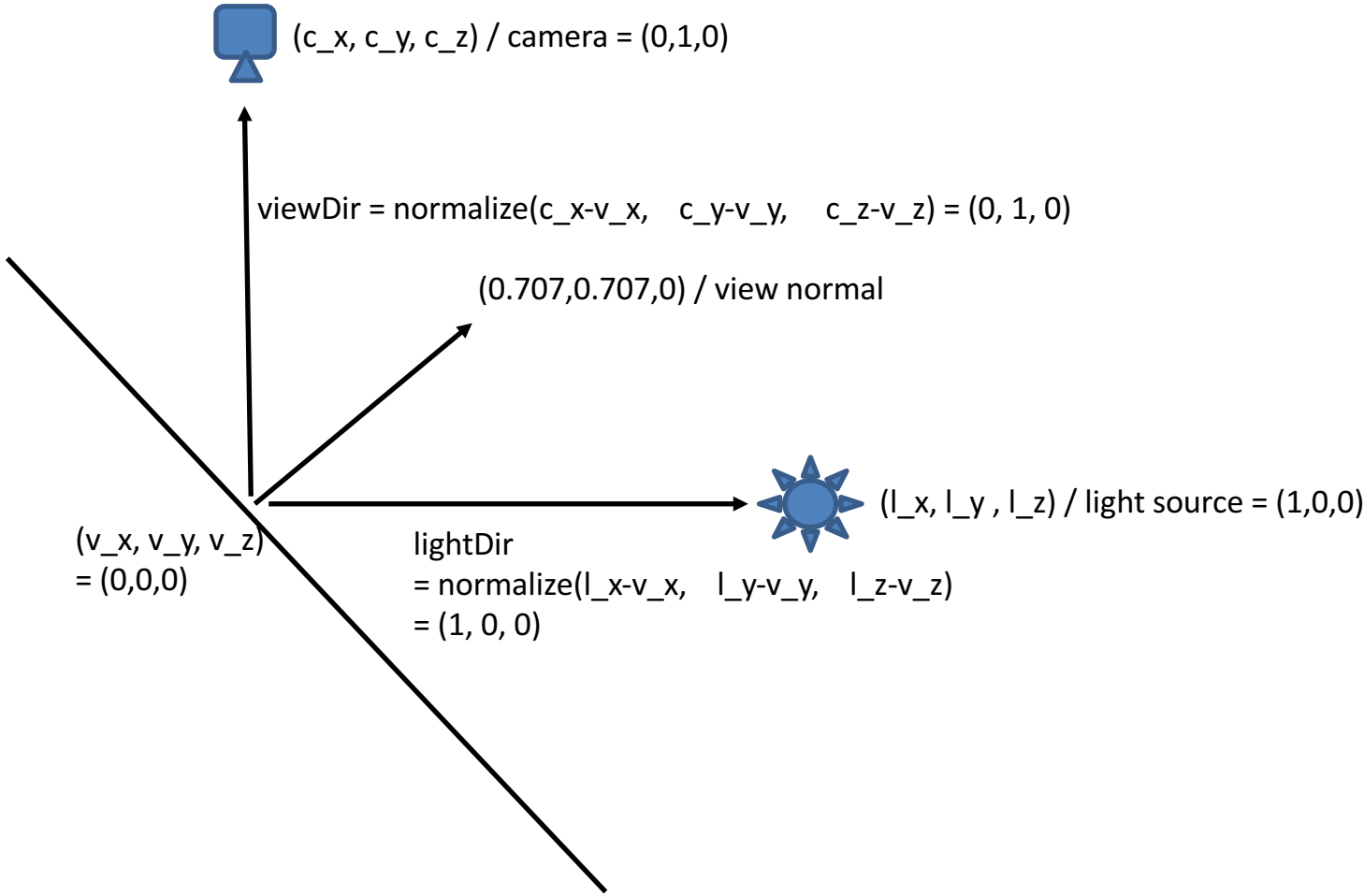
- This project in a nutshell:
  - Add method called “CalculateShading”
    - My version of CalculateShading is about ten lines of code.
  - Call CalculateShading for each vertex
  - This is a new field, which you will LERP
  - Modify RGB calculation to use shading

More comments (2/3)

- New: more data to help debug
  - I will make the shading value for each pixel available
  - I will also make it available for ambient, diffuse, specular
- ~~Don't forget to do two-sided lighting~~
- REVERSAL: do one-sided lighting

This example has a triangle vertex, v, at the origin, the camera one unit along the Y-axis and the light source one unit along the X-axis.

The lightDir and viewDir formulas show the conventions we should use for direction for general positions.

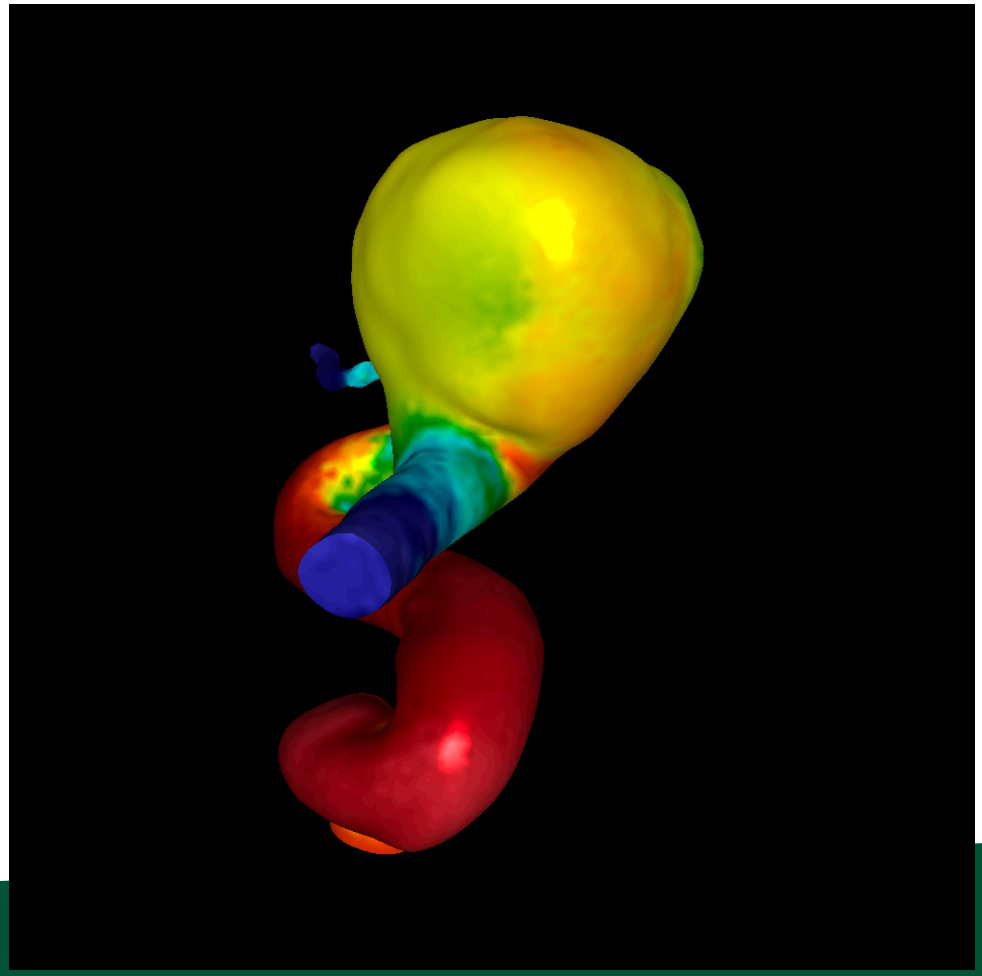


More comments (3/3)

- I haven’t said anything about movie encoders

Project #1F (8%), Due Weds May 5th

- Goal: add shading, movie



(Lecture Begins)

GLFW: Graphics Library FrameWork

- Open Source, multi-platform library for
  - OpenGL,
  - OpenGL ES, and
  - Vulkan development
- on the desktop

Source: https://www.glfw.org/

OpenGL ES?

- **OpenGL ES** is an "embeddable subset" of OpenGL
- Slims down large OpenGL API to bare essentials
- Enables implementation on devices with
  - simpler, cheaper hardware
  - power requirements (runs on batteries)
- Standard on smartphones running both Apple's **IOS** and Google's **Android** operating

Source: khronos.org

Vulkan?

- New generation graphics and compute API
- Features:
  - high-efficiency
  - cross-platform access to modern GPUs
    - PCs
    - consoles
    - mobile phones
    - embedded platforms

Source: khronos.org

GLFW: Graphics Library FrameWork

- Written in C
- Supports
  - Windows
  - macOS
  - two Unix (X11 and Wayland)

Source: https://www.glfw.org/

GLFW: Does Things We Don't Want to Do

- GLFW provides a simple API for
  - creating windows
  - receiving input and events

Source: https://www.glfw.org/

GLFW

Gives you a window and OpenGL context with just two function calls

Support for OpenGL, OpenGL ES, Vulkan and related options, flags and extensions

Support for multiple windows, multiple monitors, high-DPI and gamma ramps

Support for keyboard, mouse, gamepad, time and window event input, via polling or callbacks

Comes with a tutorial, guides and reference documentation, examples and test programs

Open Source with an OSI-certified license allowing commercial use

Access to native objects and compile-time options for platform specific features

Community-maintained bindings for many different languages

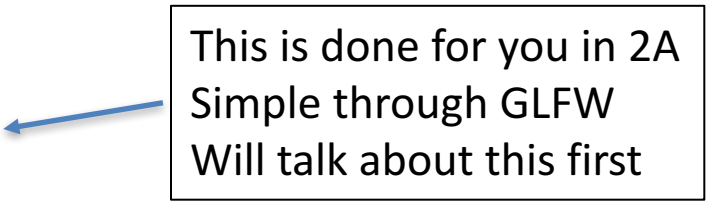
Source: https://www.glfw.org/

Layout of Simple OpenGL Program

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

Layout of Simple OpenGL Program

- Set up windows



- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

Layout of Simple OpenGL Program

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

This is done for you in 2A
Simple through GLFW
Will talk about this second

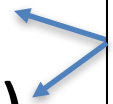
Layout of Simple OpenGL Program

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

This will be discussed for 2B. In 2A, renders one time and you are done.

Layout of Simple OpenGL Program

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render



Majority of this lecture & next.
You do both in 2A.

The remainder of this lecture and Thursday's lecture are made up of 4 parts

1) Set up windows
2) Doing a render
3) Set up things to render (VBOs)
4) Set up how to render (shaders) (Thursday)

Part 1

- Set up windows

This is done for you in 2A
Simple through GLFW
Will talk about this first

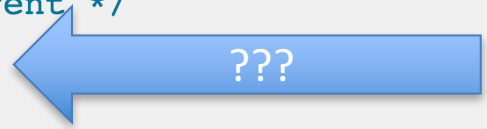
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

```
int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);
```



https://www.glfw.org/documentation.html#example-code

OpenGL Context

- An **OpenGL context** represents many things
  - A context stores all of the state associated with this instance of OpenGL
  - All of your buffers are within this context
- If you have two OpenGL programs running, they can co-exist since each works in its own context
- (Not something you need to worry about when writing your first GL programs)

https://www.khronos.org/opengl/wiki/OpenGL_Context

Note: Abhishek's program has some extra stuff – not worth worrying about

- EXCEPT:

```
// uncomment these lines if on Apple OS X
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT,
                        GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE,
                        GLFW_OPENGL_CORE_PROFILE);
```

Part 2

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

This is done for you in 2A
Simple through GLFW
Will talk about this second

Render: 3 steps

1) Initialize
2) Perform Render Actions
3) Finalize

Rendering Step #1: Initialize (1/2)

- Need to clear everything off the screen from the last render
- You did this in Project 1

```
for (int i = 0 ; i < npixels ; i++)
{
    zbuffer[i] = -1.0;
    buffer[3*i+0] = 0;
    buffer[3*i+1] = 0;
    buffer[3*i+2] = 0;
}
```

Rendering Step #1: Initialize (2/2)

- GL command: glClear
- Arguments: what to clear
  - Color buffer
  - Depth buffer ← → Z buffer

- Actual invocation:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Render Step #2: Perform Render Actions

- GL needs to know:
  - Geometry to render
  - How to render that geometry
- After a clear, you have to instruct GL to render geometry
- You can optionally tell it how to render that geometry during a render cycle
  - Or you can tell it ahead of time

From Example Program (a little modified)

```
glUseProgram(shader_programme);
glBindVertexArray(vao);
while (!glfwWindowShouldClose(window)) {
   // wipe the drawing surface clear
   glClear(GL_COLOR_BUFFER_BIT |
               GL_DEPTH_BUFFER_BIT);
   // draw points 0-3 from the currently bound VAO
   glDrawElements( GL_TRIANGLES, 6,
                           GL_UNSIGNED_INT, NULL );
    …
```

glUseProgram, glBindVertexArray, glDrawElements will be discussed later this lecture

… But this works too

```
while (!glfwWindowShouldClose(window)) {
  // wipe the drawing surface clear
  glClear(GL_COLOR_BUFFER_BIT |
                GL_DEPTH_BUFFER_BIT);
  glUseProgram(shader_programme); // modify shader each render
  glBindVertexArray(vao); // modify geometry each render
  // draw points 0-3 from the currently bound VAO
  glDrawElements( GL_TRIANGLES, 6,
                            GL_UNSIGNED_INT, NULL );
  …
```

glUseProgram, glBindVertexArray, glDrawElements will be discussed later this lecture

Rendering Step #3: Finalize

- Finalize means getting the image to the viewer on the display
- In graphics, maintain two copies of buffers
  - Copy #1 (“front buffer”): given to the display for user to see
  - Copy #2 (“back buffer”): being generated “right now”
- When rendering is done, swap “copy #2” into “copy #1” and start over
- Command: glfwSwapBuffers(window);
  - (And there is an OpenGL equivalent)

Why Double Buffered?

- General computer science idea: double buffering (or “multiple buffering”)
  - use of more than one buffer to hold a block of data
  - Why?
    - "reader" sees a complete (though perhaps old) version of the data, rather than a partially updated version of the data being created by a “writer”
- In other words: if you are continuously working on something, then regularly make a copy and show that to the user, rather than risking them see incomplete/partial versions

https://en.wikipedia.org/wiki/Multiple_buffering

Part 3

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

Majority of this lecture and you do both in 2A.

Game Plan

- Plan
  - Set up small things
  - Wrap the small things up into one big thing
- More detail
  - Small things are buffers / Vertex Buffer Objects (VBOs)
  - Big things are arrays of buffers / Vertex Array Object (VAOs)
- Lecture
  - Starts with VBO and then go on to VAO
  - Focuses on starter code for 2A

Walking Through the Starter Code

```
float points[] = {0.5f, 0.0f, 0.0f,
                  0.0f, 0.0f, 0.0f,
                  0.0f, 0.5f, 0.0f,
                 -0.5f, 0.0f, 0.0f};

float colors[] = {1.0f, 0.0f, 0.0f,
                  0.0f, 1.0f, 0.0f,
                  0.0f, 0.0f, 1.0f,
                  1.0f, 0.0f, 0.0f};

GLuint indices[] = {0 ,1, 2,
                    1, 2, 3};
```

- 4 points:
  - V0 = (0.5, 0, 0), red
  - V1 = (0, 0, 0), green
  - V2 = (0, .5, 0), blue
  - V3 = (-0.5, 0, 0), red
- 6 indices for 2 triangles
  - Triangle 0: (V0,V1,V2)
  - Triangle 1: (V1,V2,V3)

glGenBuffers / glBindBuffers / glBufferData

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- GLuint: this is an unsigned integer, but OpenGL defines its own type so it can deal with portability issues (like 32 bits vs 64 bits)

glGenBuffers / glBindBuffers / glBufferData

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- glGenBuffers
  - Asks OpenGL to generate a new buffer for the programmer to work with
  - That buffer will have a unique identifier (points_vbo)
  - This unique identifier is useful: lets programmer tell OpenGL which buffer they want to operate on

glGenBuffers / glBindBuffers / glBufferData

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- glBindBuffer
  - Buffers can operate on different types of “targets”
    - (I think types would be a better word than targets)
  - glBindBuffer says what type of target a buffer will operate on
  - It also makes the buffer “active,” meaning subsequent GL calls will use this buffer

Targets for glBindBuffers

Buffer Binding Target	Purpose
GL_ARRAY_BUFFER	Vertex attributes
GL_ATOMIC_COUNTER_BUFFER	Atomic counter storage
GL_COPY_READ_BUFFER	Buffer copy source
GL_COPY_WRITE_BUFFER	Buffer copy destination
GL_DISPATCH_INDIRECT_BUFFER	Indirect compute dispatch commands
GL_DRAW_INDIRECT_BUFFER	Indirect command arguments
GL_ELEMENT_ARRAY_BUFFER	Vertex array indices
GL_PIXEL_PACK_BUFFER	Pixel read target
GL_PIXEL_UNPACK_BUFFER	Texture data source
GL_QUERY_BUFFER	Query result buffer
GL_SHADER_STORAGE_BUFFER	Read-write storage for shaders
GL_TEXTURE_BUFFER	Texture data buffer
GL_TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer
GL_UNIFORM_BUFFER	Uniform block storage

- We will use the ones underlined in red
- Distinction
  - Is this the data?
  - Or are these indices into existing data?

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBindBuffer.xhtml

glGenBuffers / glBindBuffers / glBufferData

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- glBufferData
  - Tells OpenGL about your actual data
  - Notes:
    - “target” (GL_ARRAY_BUFFER) is repeated
    - Passing 48 bytes, but not saying anything (yet) about how to interpret the data
    - GL_STATIC_DRAW tells OpenGL about the usage

glBufferData: usage types

- Hint to OpenGL about how data will be used
- Two parts:
  - Frequency of access
  - Nature of access

STREAM

The data store contents will be modified once and used at most a few times.

STATIC

The data store contents will be modified once and used many times.

DYNAMIC

The data store contents will be modified repeatedly and used many times.

DRAW

The data store contents are modified by the application, and used as the source for GL drawing and image specification commands.

READ

The data store contents are modified by reading data from the GL, and used to return that data when queried by the application.

COPY

The data store contents are modified by reading data from the GL, and used as the source for GL drawing and image specification commands.

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBufferData.xhtml

glGenBuffers / glBindBuffers / glBufferData

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- So what did this code do?
- 1) asked GL to make a buffer
- 2) told GL the buffer would be used to store an array
- 3) told GL the actual data to put in the buffer

More Starter Code

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);

GLuint colors_vbo = 0;
glGenBuffers(1, &colors_vbo);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), colors, GL_STATIC_DRAW);

GLuint index_vbo;    // Index buffer object
glGenBuffers( 1, &index_vbo);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );
glBufferData( GL_ELEMENT_ARRAY_BUFFER, 6*sizeof(GLuint), indices, GL_STATIC_DRAW );
```

This one is indices, not data

Vertex Buffer Object versus Vertex Array Object

- Vertex Buffer Object (VBO):
  - Memory buffer in your GPU
  - Contains information about vertices
- Vertex Array Object (VAO):
  - Contains one or more VBOs
  - Should contain a “complete” renderable object
- Summary:
  - VBOs store your vertex data
  - VAOs wrap up VBOs into something that can be rendered

Next Step in Starter Code: Make a VAO and put VBOs into VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- glGenVertexArrays
  - Just like glGenBuffers, but for VAOs
  - Asks OpenGL to generate a new VAO for the programmer to work with
  - That buffer will have a unique identifier (vao)
  - This unique identifier is useful: lets programmer tell OpenGL which buffer they want to operate on

Next Step in Starter Code: Make a VAO and put VBOs into VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- glBindVertexArray
  - Just like glBindBuffer, but for VAOs
  - It also makes the buffer “active,” meaning subsequent GL calls will use this buffer
    - glBindBuffer commands will put the VBOs into this VAO

Next Step in Starter Code: Make a VAO and put VBOs into VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- We've seen this before!
- Further, this code could be tightened up
- Could start by building VAO, and then build VBOs are part of the VAO building process
  - (Call glBindBuffer once, not twice)
- I like how Abhishek set it up – easier to understand

Next Step in Starter Code: Make a VAO and put VBOs into VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- Tells GL how to interpret a VBO within the VAO
- This one is for the 0^{th} VBO, which is points_vbo
- Arguments:
  - 0: the 0th VBO – goes in “location 0” of the shader program
  - 3: there are 3 values per vertex
  - GL_FLOAT: they are floats
  - GL_FALSE: don’t normalize this data
  - 0/NULL: deals with data layout stuff (always 0/NULL for 441)

Next Step in Starter Code: Make a VAO and put VBOs into VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- Tells GL how that array 0 (i.e., points_vbo) should be enabled – it should be processed when vao is processed
- We always want to enable for 441
- Why disable? Improved performance if not using an array

(REPEAT SLIDE FROM PART 2) From Example Program

```
while (!glfwWindowShouldClose(window)) {
  // wipe the drawing surface clear
  glClear(GL_COLOR_BUFFER_BIT |
              GL_DEPTH_BUFFER_BIT);
  glUseProgram(shader_programme);
  glBindVertexArray(vao);
  // draw points 0-3 from the currently bound VAO
  glDrawElements( GL_TRIANGLES, 6,
                          GL_UNSIGNED_INT, NULL );
  ...
```

glUseProgram, glBindVertexArray, glDrawElements will be discussed ~~later this lecture~~ now

(REPEAT SLIDE FROM PART 2) From Example Program

```
glBindVertexArray(vao);
// draw points 0-3 from the currently bound VAO
glDrawElements( GL_TRIANGLES, 6,
                GL_UNSIGNED_INT, NULL );
```

- Tells OpenGL that commands that follow will be for vertex array object “vao”

(REPEAT SLIDE FROM PART 2) From Example Program

```
glBindVertexArray(vao);
// draw points 0-3 from the currently bound VAO
glDrawElements( GL_TRIANGLES, 6,
                GL_UNSIGNED_INT, NULL );
```

- Tells OpenGL to draw the elements in the current VAO
- And:
  - GL_TRIANGLES: the indices are describing triangles
  - 6: there are 6 indices (2 triangles total)
  - GL_UNSIGNED_INT: the indices are unsigned int
  - NULL: something for fancy array layouts (we don't need this for 441)

Project 2A

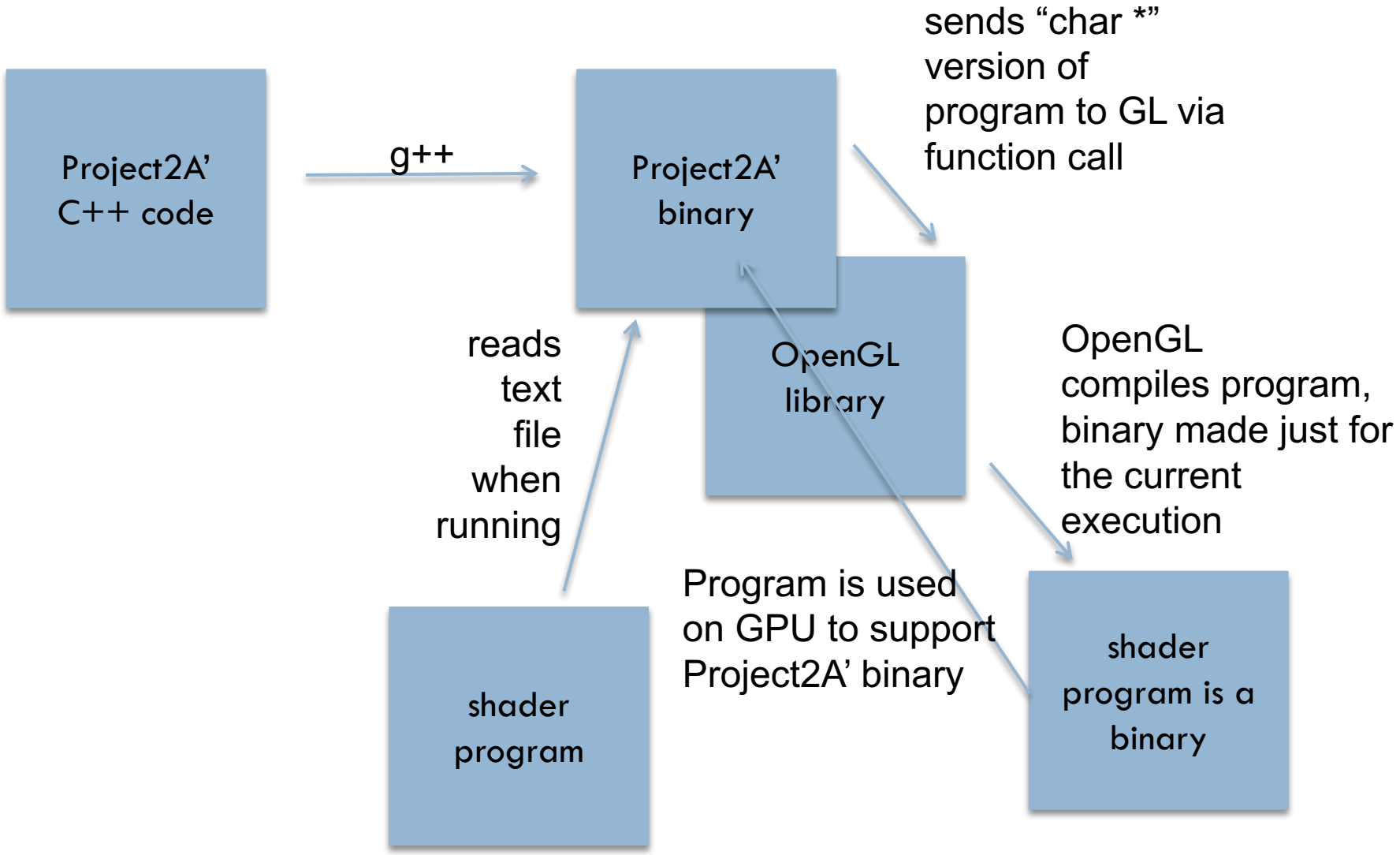
- Assigned today, due in one week (Tuesday May 11)
- Worth 8% of your grade
- Implementing Project 1 within OpenGL
- 5 phases
  - Phase 1: install GLFW
  - Phase 2: run example program
  - Phase 3: modify VBO/VAO
  - Phases 4 & 5: shader programs
- Please start ASAP on Phase 1-3
- Thursday's lecture will be on Phase 4 & 5

Finish lecture by talking again about compiling shaders

How to Use Shaders

- You write a shader program: a tiny C-like program
- You write C/C++ code for your application
- Your application loads the shader program from a text file
- Your application sends the shader program to the OpenGL library and directs the OpenGL library to compile the shader program
- If successful, the resulting GPU code can be attached to your (running) application and used
- It will then supplant the built-in GL operations

How to Use Shaders: Visual Version



Compiling Shader

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);
```

Compiling Shader: inspect if it works

```
if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}
```

Compiling Multiple Shaders

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);

if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
std::string fragmentProgram = loadFileToString("fs.glsl");
const char *fragment_shader_source = fragmentProgram.c_str();
GLint const fragment_shader_length = strlen(fragment_shader_source);
glShaderSource(fragmentShader, 1, &fragment_shader_source, &fragment_shader_length);
glCompileShader(fragmentShader);
GLint isCompiledFS = 0;
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &isCompiledFS);
```

Attaching Shaders to a Program

```
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);

glLinkProgram(program);

glDetachShader(program, vertexShader);
glDetachShader(program, fragmentShader);
```

Inspecting if program link worked…

```
GLint isLinked = 0;
glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
if(isLinked == GL_FALSE)
{
   GLint maxLength = 0;
   glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

   //The maxLength includes the NULL character
   std::vector<GLchar> infoLog(maxLength);
   glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
   cerr << "Couldn't link" << endl;
   cerr << "Log says " << &(infoLog[0]) << endl;

   exit(EXIT_FAILURE);
}
```

Simplest Vertex Shader

```
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Many built-in variables.
Some are input.
Some are required output (gl_Position).