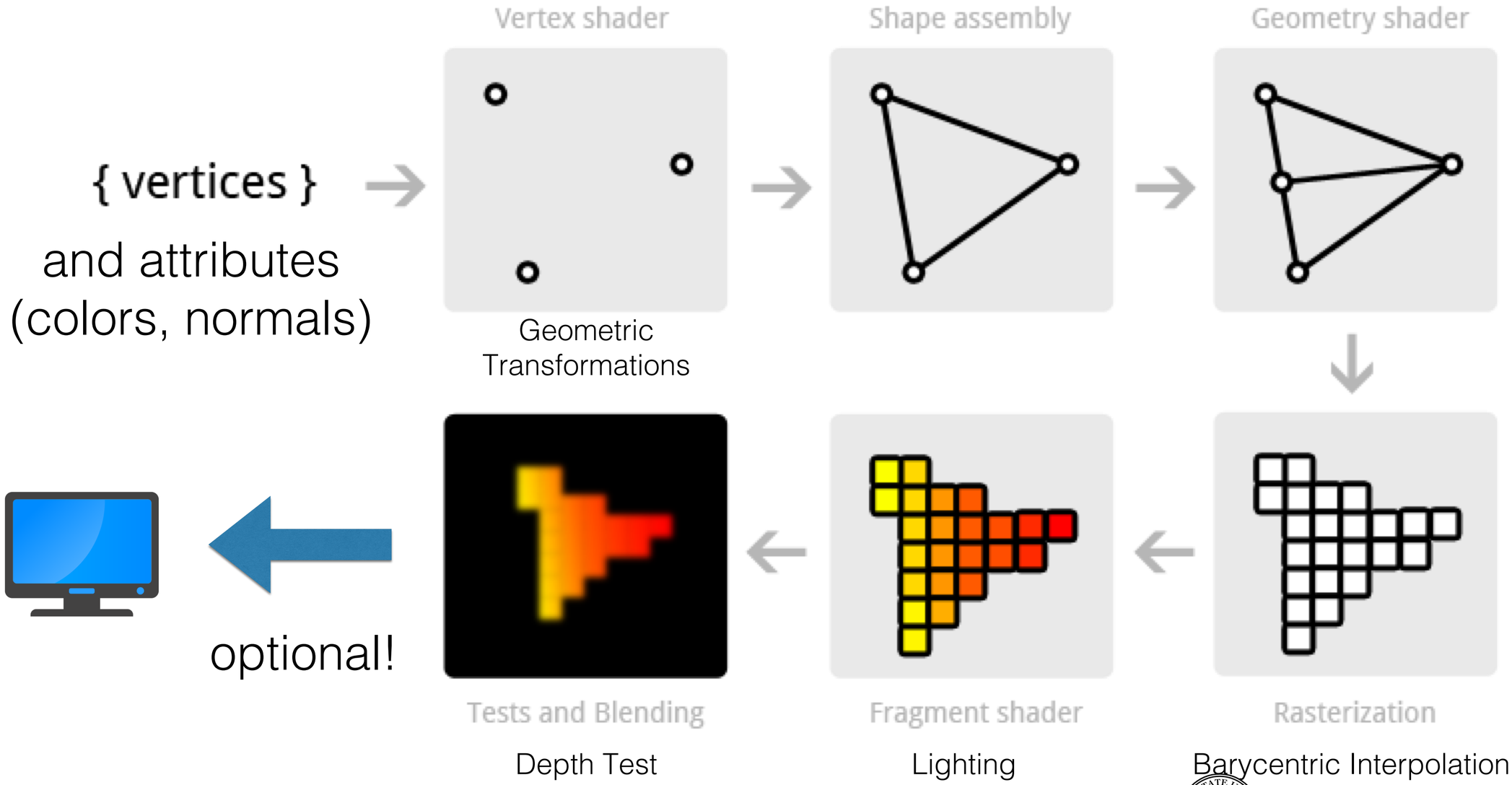
10 - The OpenGL Graphics Pipeline - Part 3

Acknowledgements: Daniele Panozzo

https://open.gl

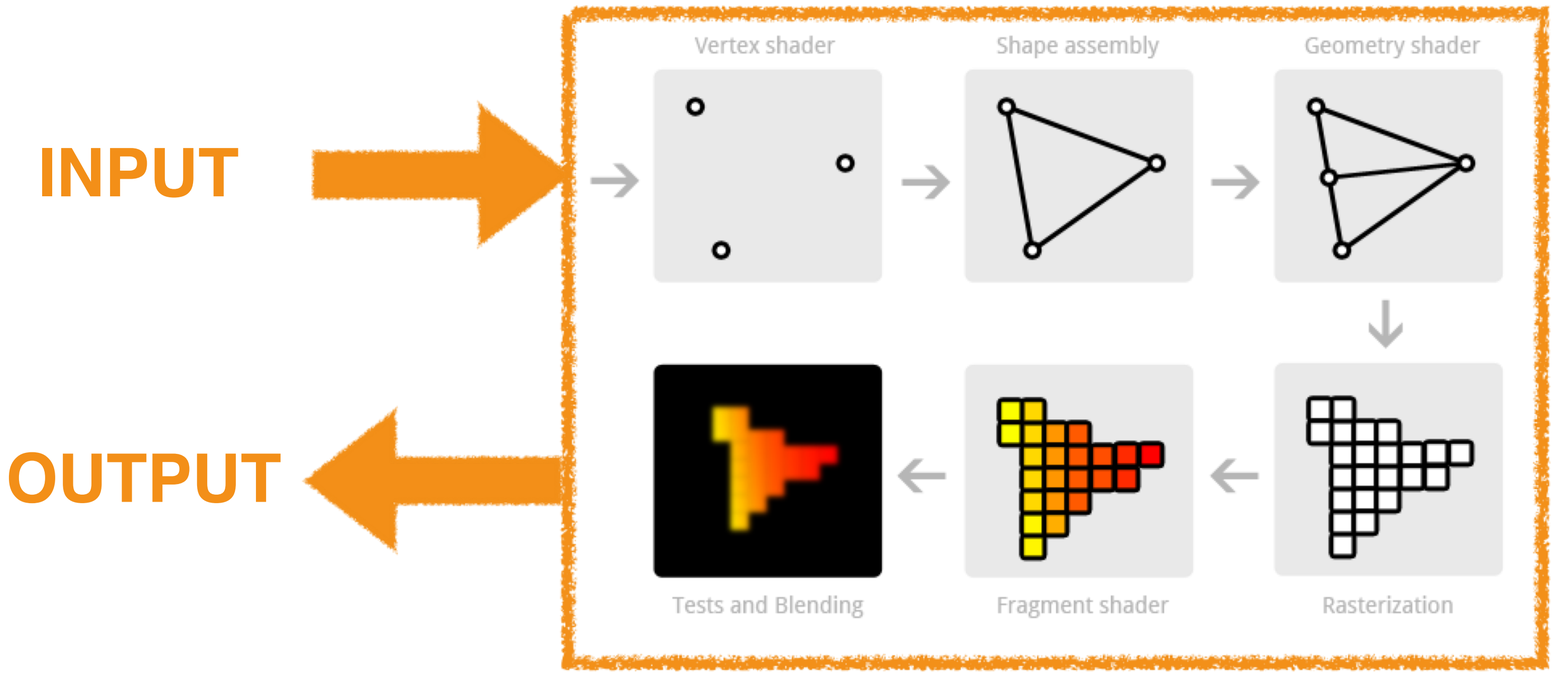
- This slide set is based on the excellent tutorial available at https://open.gl
- Many thanks to:
  - Toby Rufinus
  - Eric Engeström
  - Elliott Sales de Andrade
  - Aaron Hamilton

OpenGL pipeline



OpenGL Program

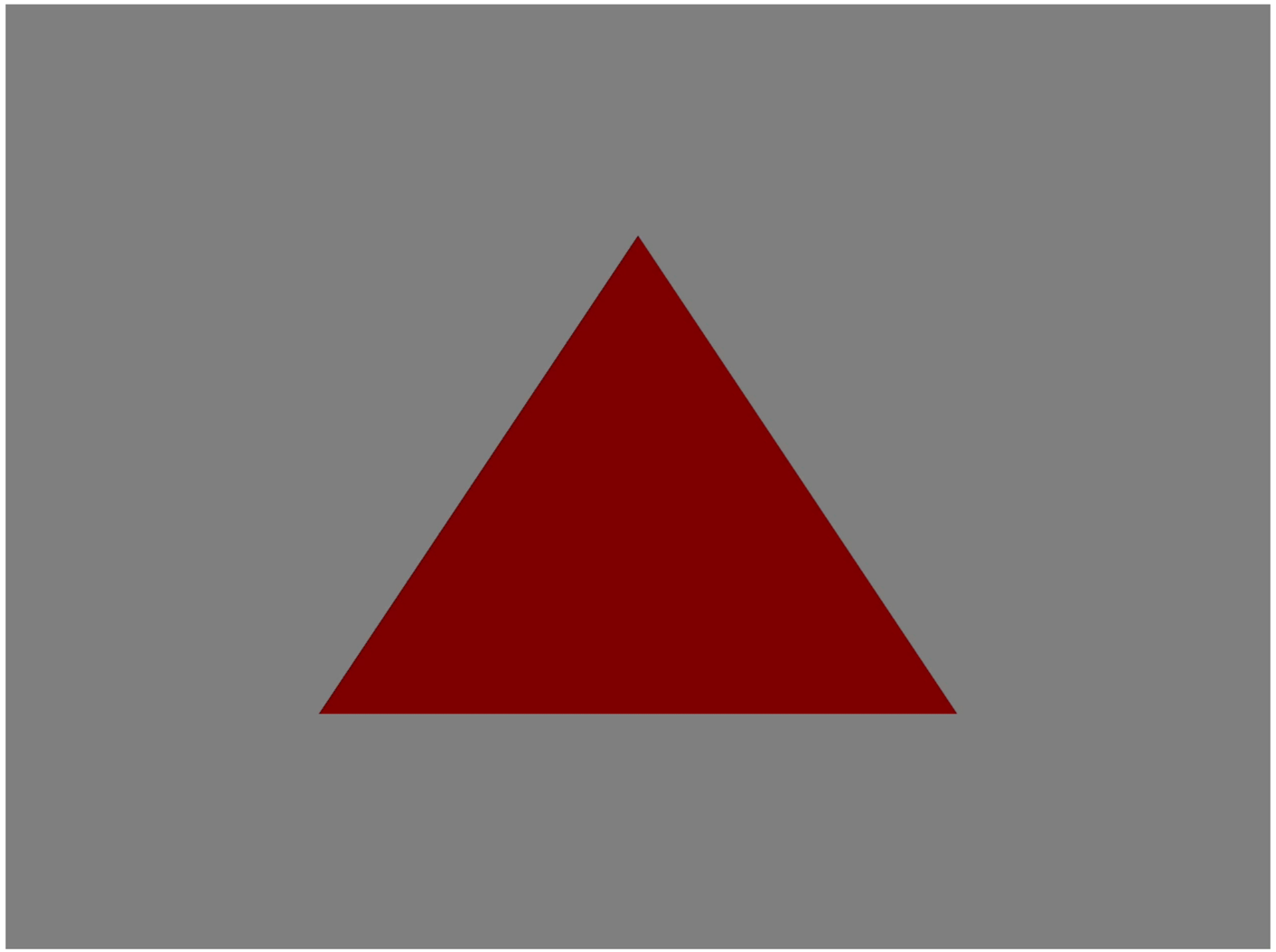
We need to connect the program with our input data and map the output to a memory buffer or to the screen



Recap

- Compile, link and enable an OpenGL program (vertex + fragment shader)
- Connects the output of the fragment shader to the frame buffer
- Connect the VBOs to the input slots of the vertex shader using a VAO
- Assign the uniform parameters (if you use them in the shaders)
- Clear the framebuffer
- Draw the primitives
- Swap the framebuffer to make the newly rendered frame visible

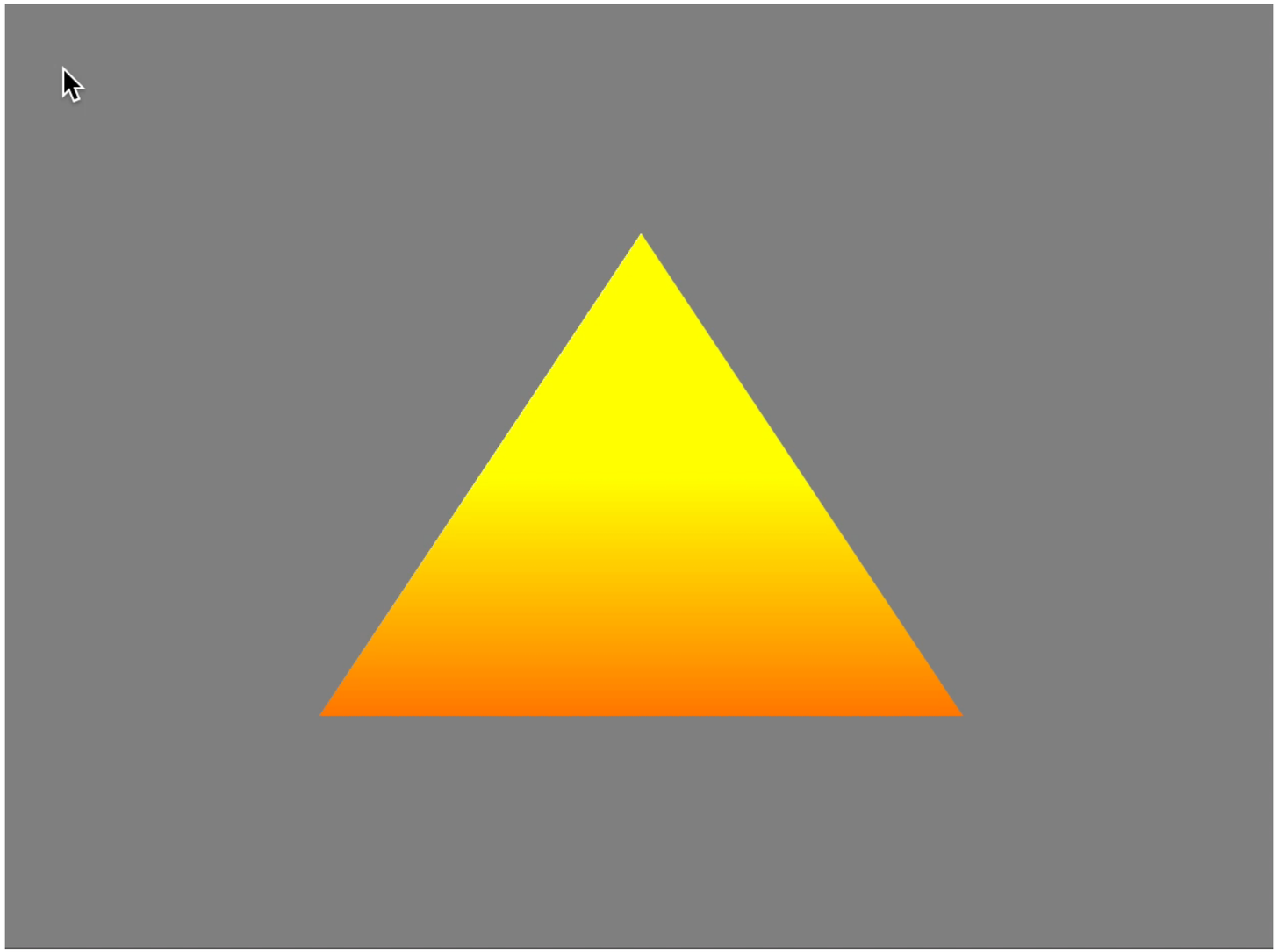
Assignment 2



Live Coding

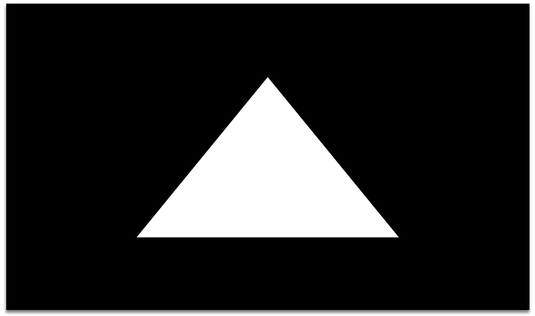
- Color the triangle depending on the position in screen coordinates
- The vertex and fragment shaders have standard input/outputs, you can find more info here: https://www.opengl.org/wiki/Built-in_Variable_(GLSL)
- See example in Assignment_2/extra/main_positions.cpp

main_positions.cpp



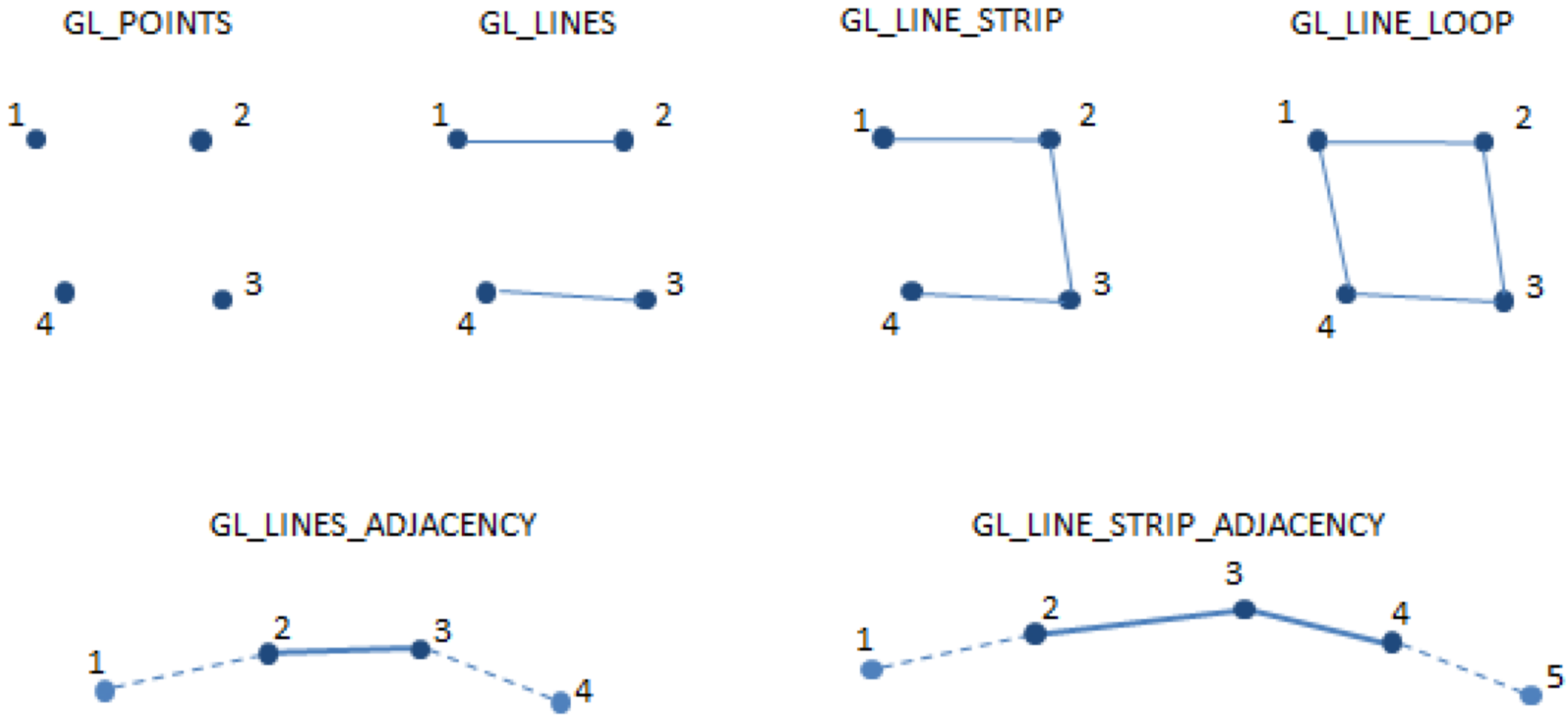
Supported Primitives

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```



- GL_POINTS - Every vertex is drawn separately
- GL_LINE_STRIP - Sequence of lines
- GL_LINE_LOOP - Sequence of lines, always closed
- GL_LINES - Draws a line for each pair of points
- GL_TRIANGLES - Draws a triangle for each three points

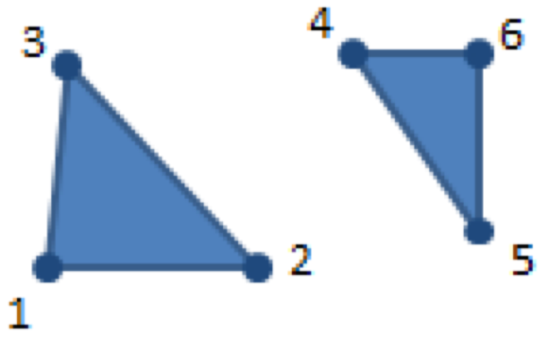
Supported Primitives



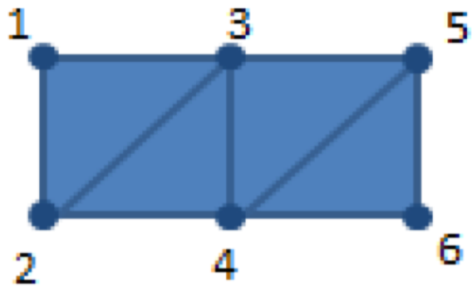
http://www.lighthouse3d.com/tutorials/glsl-tutorial/primitive-assembly/

Supported Primitives

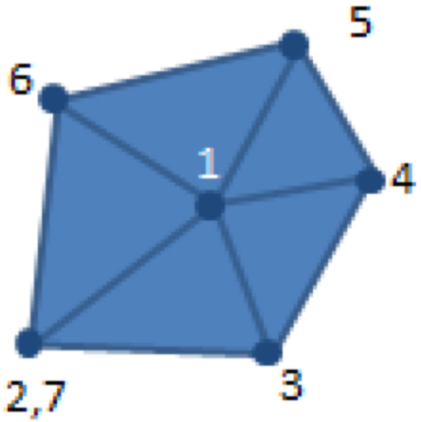
GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

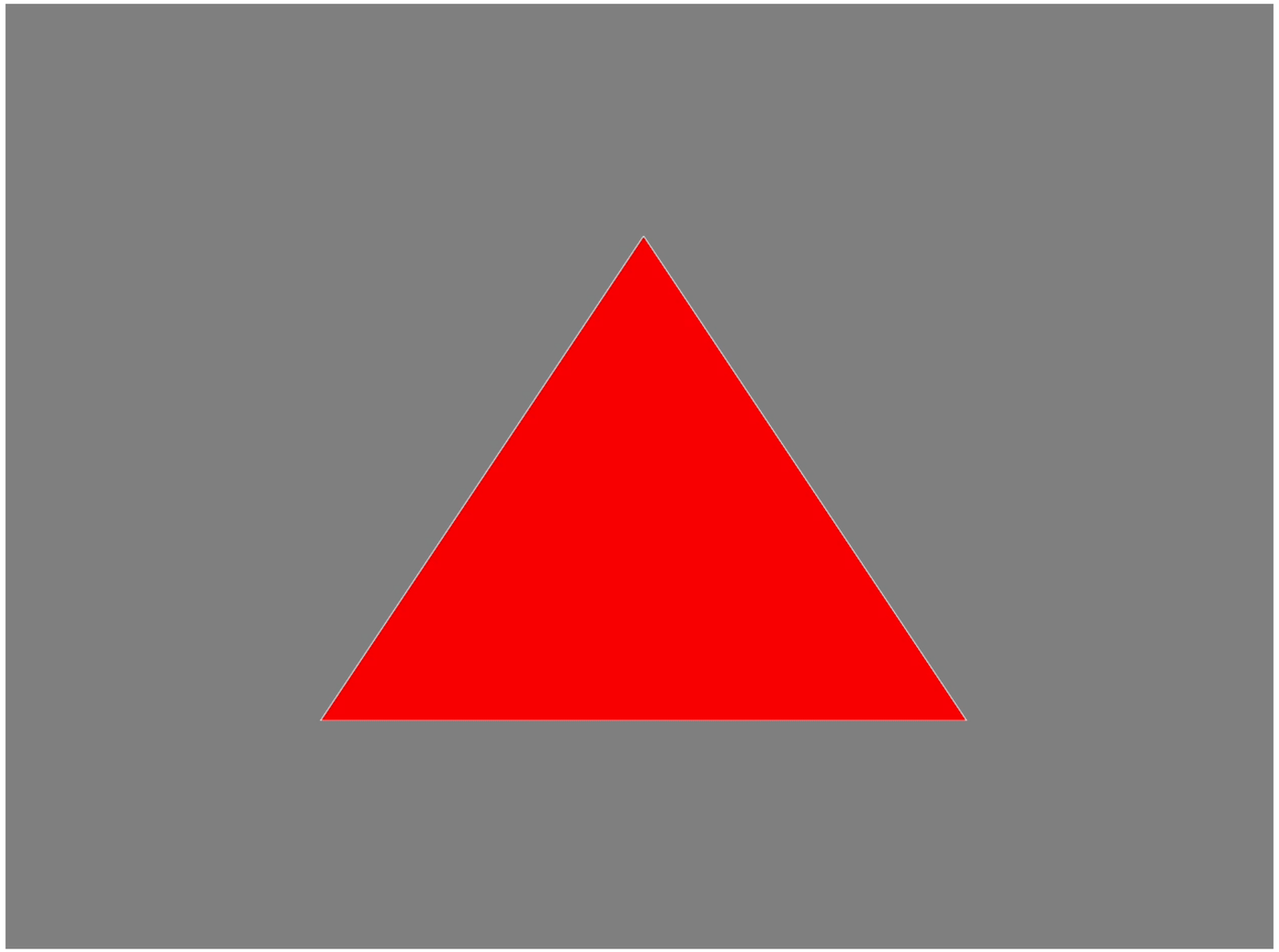


http://www.lighthouse3d.com/tutorials/glsl-tutorial/primitive-assembly/

Live Coding

- Let's add a white border to the triangle using a line loop
- See example in Assignment_2/extra/main_border.cpp

main_border.cpp



Element Buffer

- For describing complex shapes it is convenient to reuse the same vertex multiple times
- Similar to what is done in the indexed mesh format (OFF)
- You need to provide an additional VBO with the ids of the vertices that form a primitive
- Let's see how to draw a square with and without the element buffer

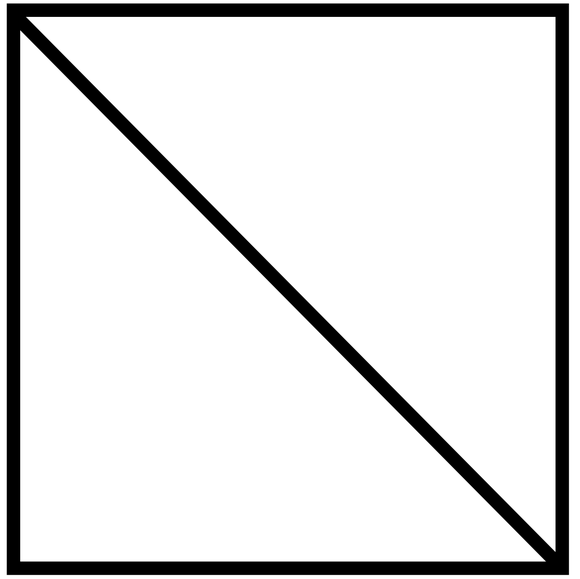
Without Element Buffer

```
float vertices[] = {
    -0.5f,  0.5f, 1.0f, // Top-left
     0.5f,  0.5f, 0.0f, // Top-right
     0.5f, -0.5f, 0.0f, // Bottom-right

     0.5f, -0.5f, 0.0f, // Bottom-right
    -0.5f, -0.5f, 1.0f, // Bottom-left
    -0.5f,  0.5f, 1.0f  // Top-left
};

// Upload the VBO to the GPU

glDrawArrays(GL_TRIANGLES, 0, 6);
```



- We are using one VBO with 6 vertices
- Some vertices are repeated

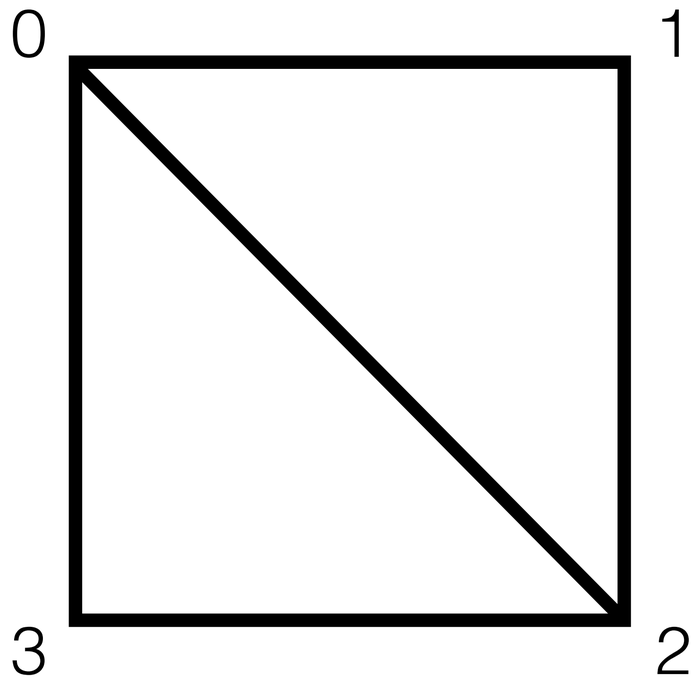
With Element Buffer

```
float vertices[] = {
    -0.5f,  0.5f, 1.0f, // Top-left
     0.5f,  0.5f, 0.0f, // Top-right
     0.5f, -0.5f, 0.0f, // Bottom-right
    -0.5f, -0.5f, 1.0f, // Bottom-left
};

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

// Upload both VBOs to the GPU

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```



- Vertices can be reused and are uploaded only once
- It saves space if you have many properties attached to the vertices

Careful when you upload the VBOs

```
// Uploading the VBO for vertices
glBindBuffer(GL_ARRAY_BUFFER, ebo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

// Uploading the VBO for elements
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ele);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
GL_STATIC_DRAW);
```

- The type of buffer is different
- It will not draw anything if you assign it wrong
- Note: The provided VBO class does not support ELEMENT_ARRAY_BUFFER

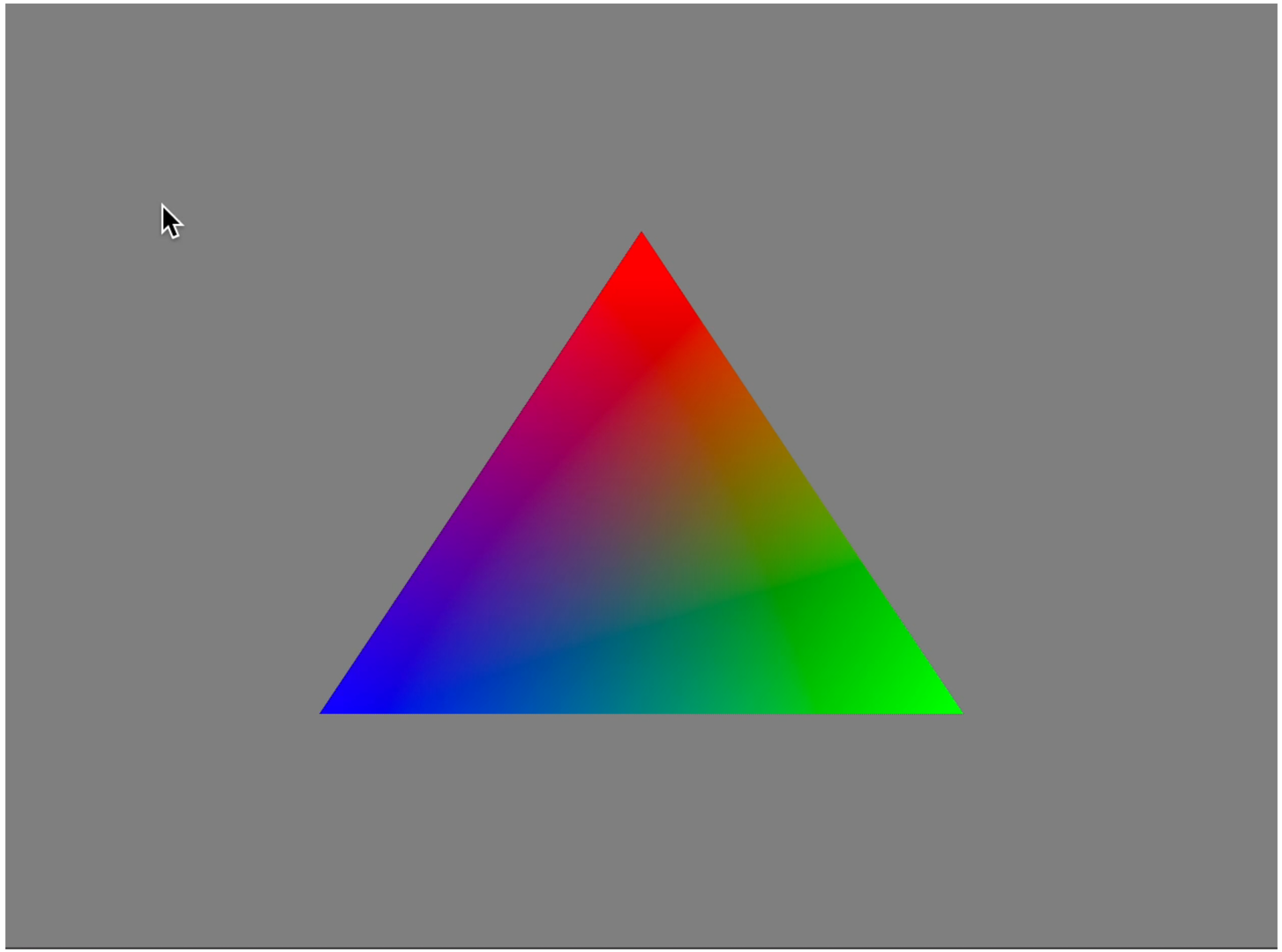
Multiple Properties

- There are different ways to send multiple properties to a vertex shader
  - Each property could be sent in a separate VBO
  - All properties could be packed in a single VBO and then “sliced”

Live Coding

- Example of adding a color property and interpolating it inside the triangle
- Important note: The names of the outputs of the vertex shader should match the names of the inputs of the fragment shader
- See example in Assignment_2/extra/main_properties.cpp

main_properties.cpp



Per-face colors?

- How can we get the same color on each face?
- And how can we do it if we share the vertices with an element buffer?

Alternatively, a single VBO

```
float vertices[] = {
     0.0f,  0.5f, 1.0f, 0.0f, 0.0f, // Vertex 1: Red
     0.5f, -0.5f, 0.0f, 1.0f, 0.0f, // Vertex 2: Green
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f  // Vertex 3: Blue
};
```

Vertex Shader

```
#version 150
in vec2 position;
in vec3 color;
out vec3 Color;

void main()
{
    Color = color;
    gl_Position = vec4(position, 0.0, 1.0);
}
```

Fragment Shader

```
#version 150
in vec3 Color;
out vec4 outColor;
void main()
{
    outColor = vec4(Color, 1.0);
}
```

Alternatively, a single VBO

```
float vertices[] = {
     0.0f,  0.5f, 1.0f, 0.0f, 0.0f, // Vertex 1: Red
     0.5f, -0.5f, 0.0f, 1.0f, 0.0f, // Vertex 2: Green
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f  // Vertex 3: Blue
};
```

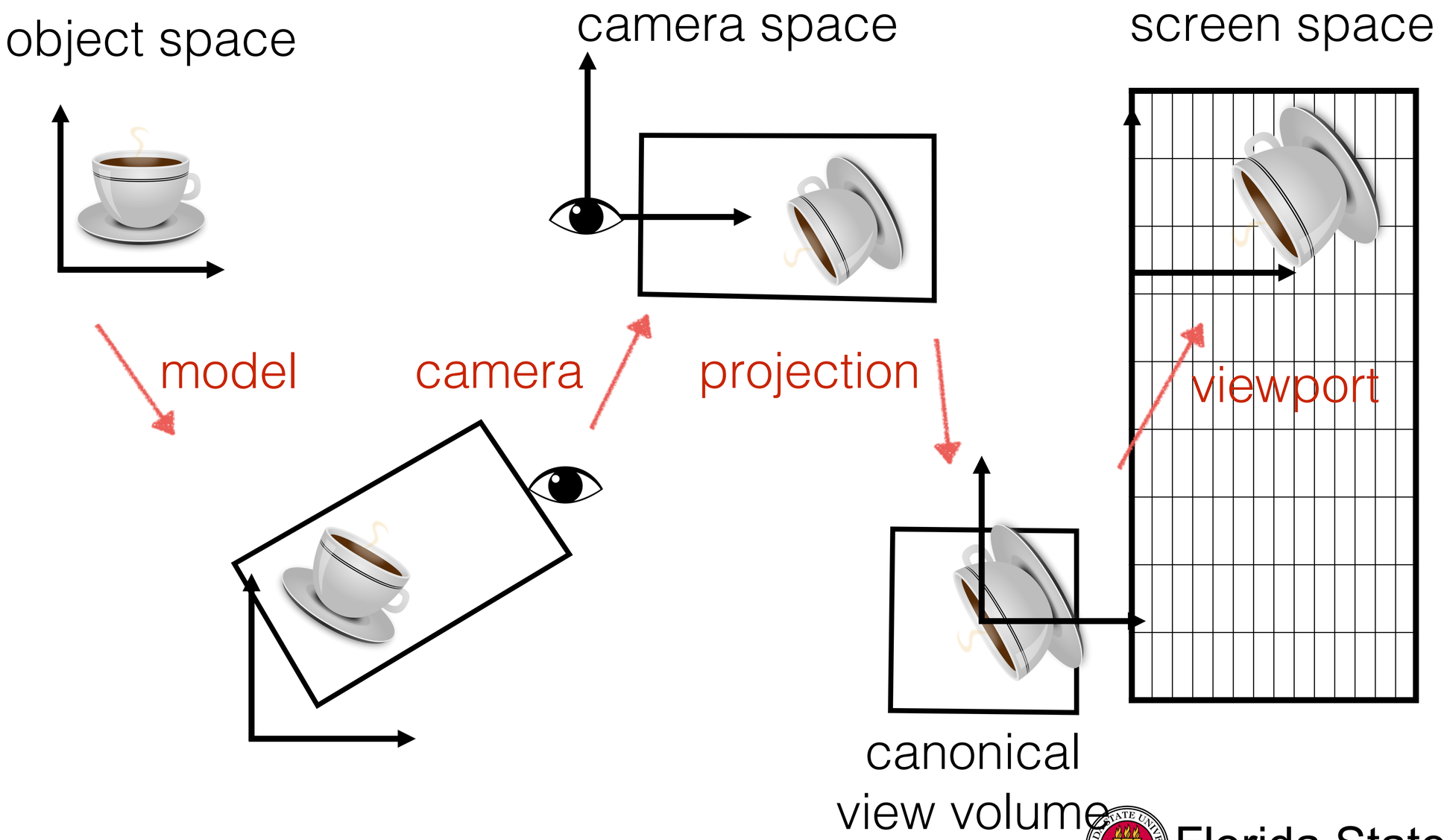
```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(
posAttrib, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(
colAttrib, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)(2*sizeof(float)));
```

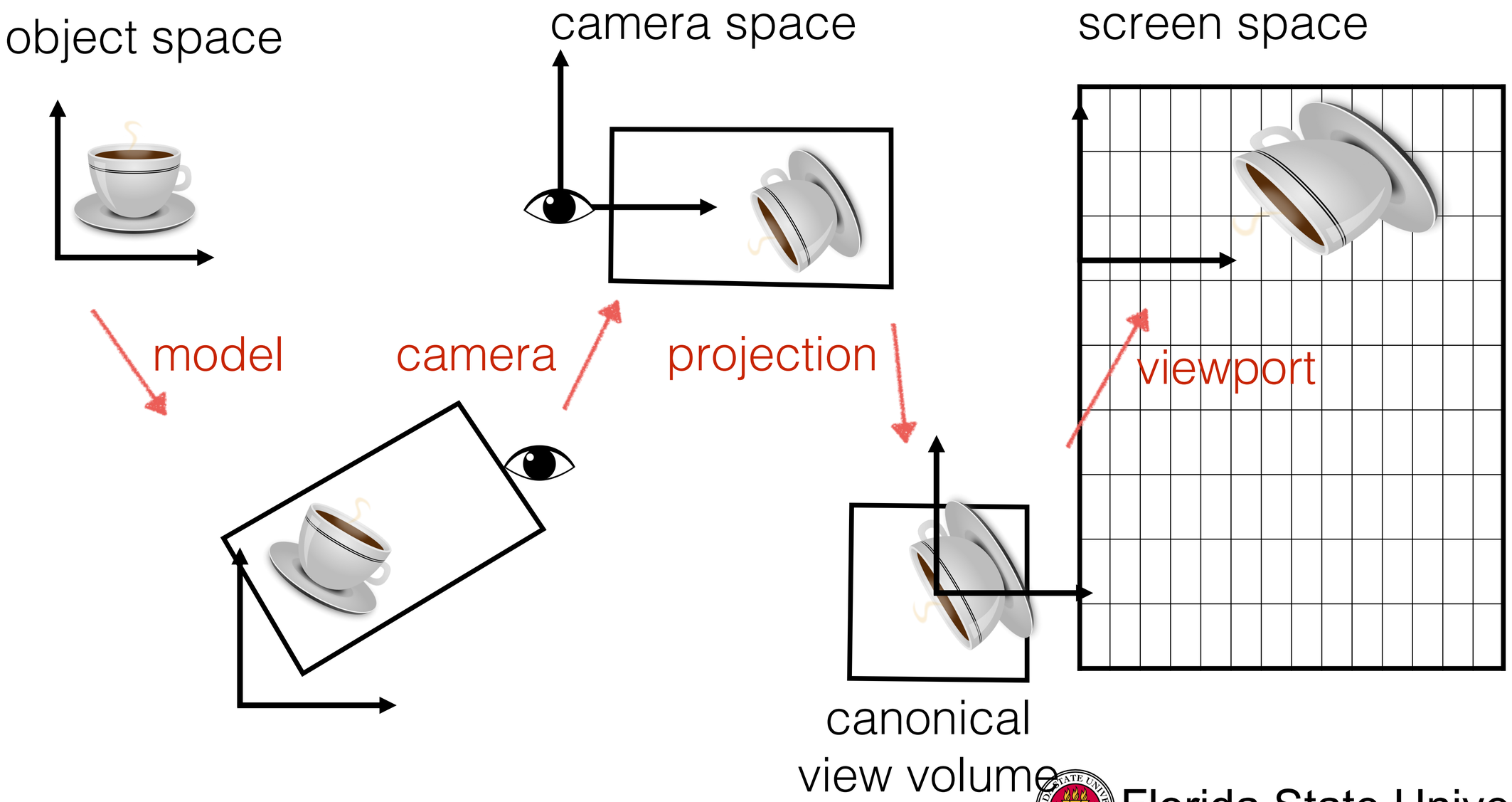
View/Model Transformation

- View/Model transformations are applied in the vertex shader
- They are passed to the shader as uniform
- To create transformation matrices you can do it by hand or use the geometry module of Eigen: https://eigen.tuxfamily.org/dox/group__TutorialGeometry.html

Viewport and View Transformation



Viewport and View Transformation



How to prevent viewport distortion?

- We need to query glfw to know the size of the window

```
int width, height;
glfwGetWindowSize(window, &width, &height);
```

- We can then create a view transformation that maps a box with the same aspect-ratio of the viewport into the unit cube

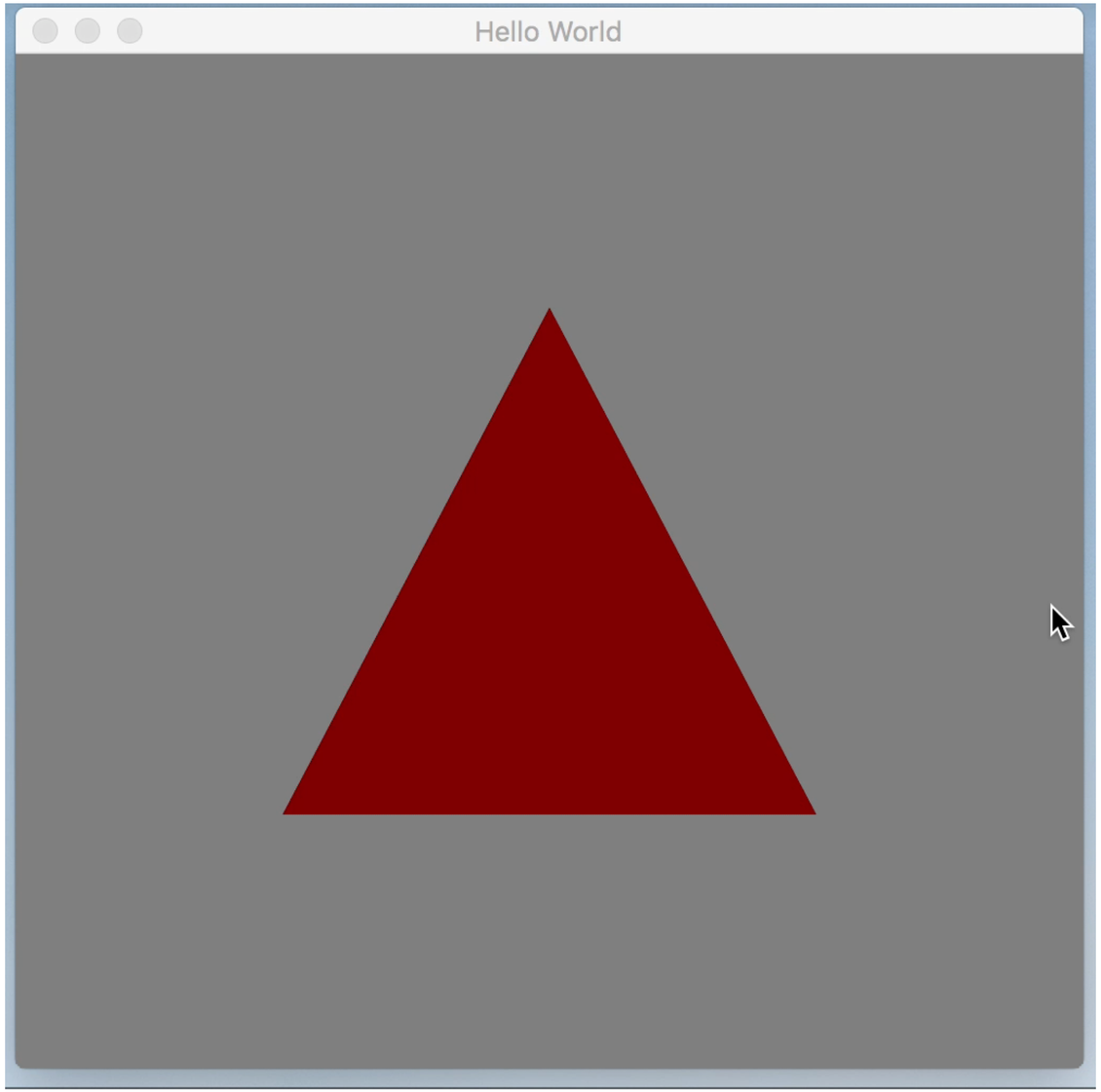
- Equivalently, we are using a “camera” that has the same aspect ratio as the window that we use for rendering

- In this way, the distortion introduced by the viewport transformation will cancel out

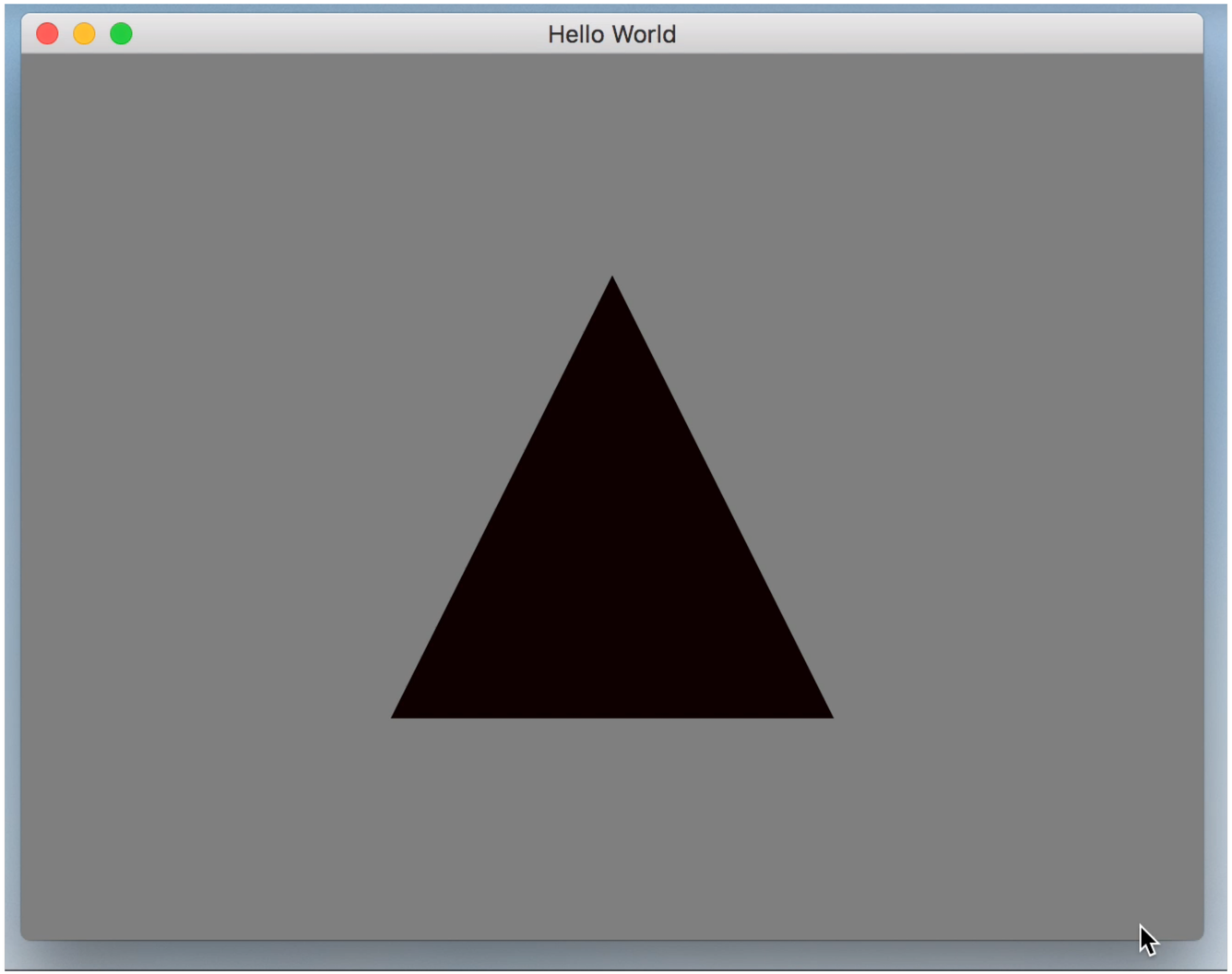
Live Coding

- Let’s add a view transformation to prevent viewport distortion
- See example in Assignment_2/extra/main_view.cpp

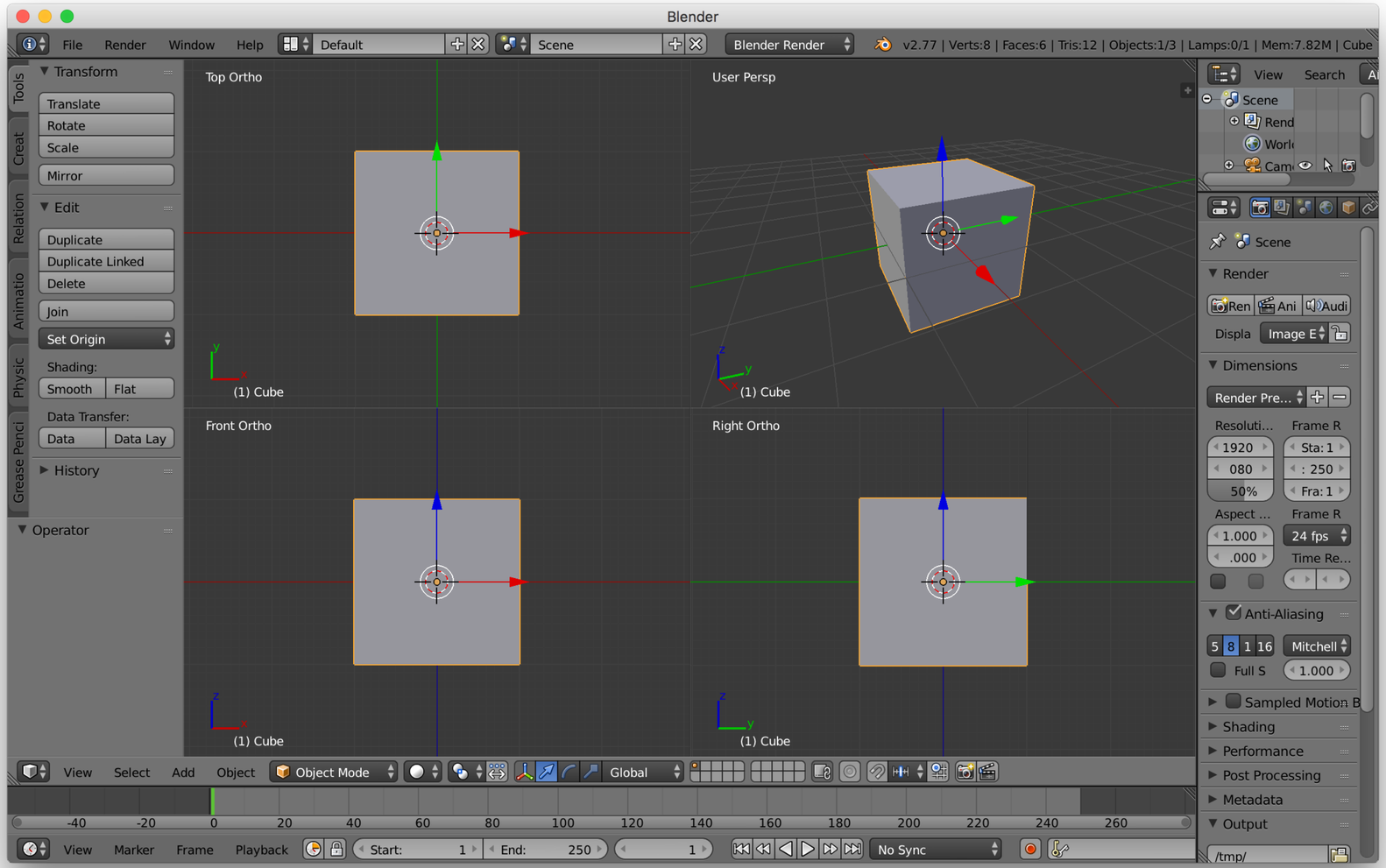
main.cpp



main_view.cpp



Changing the viewport

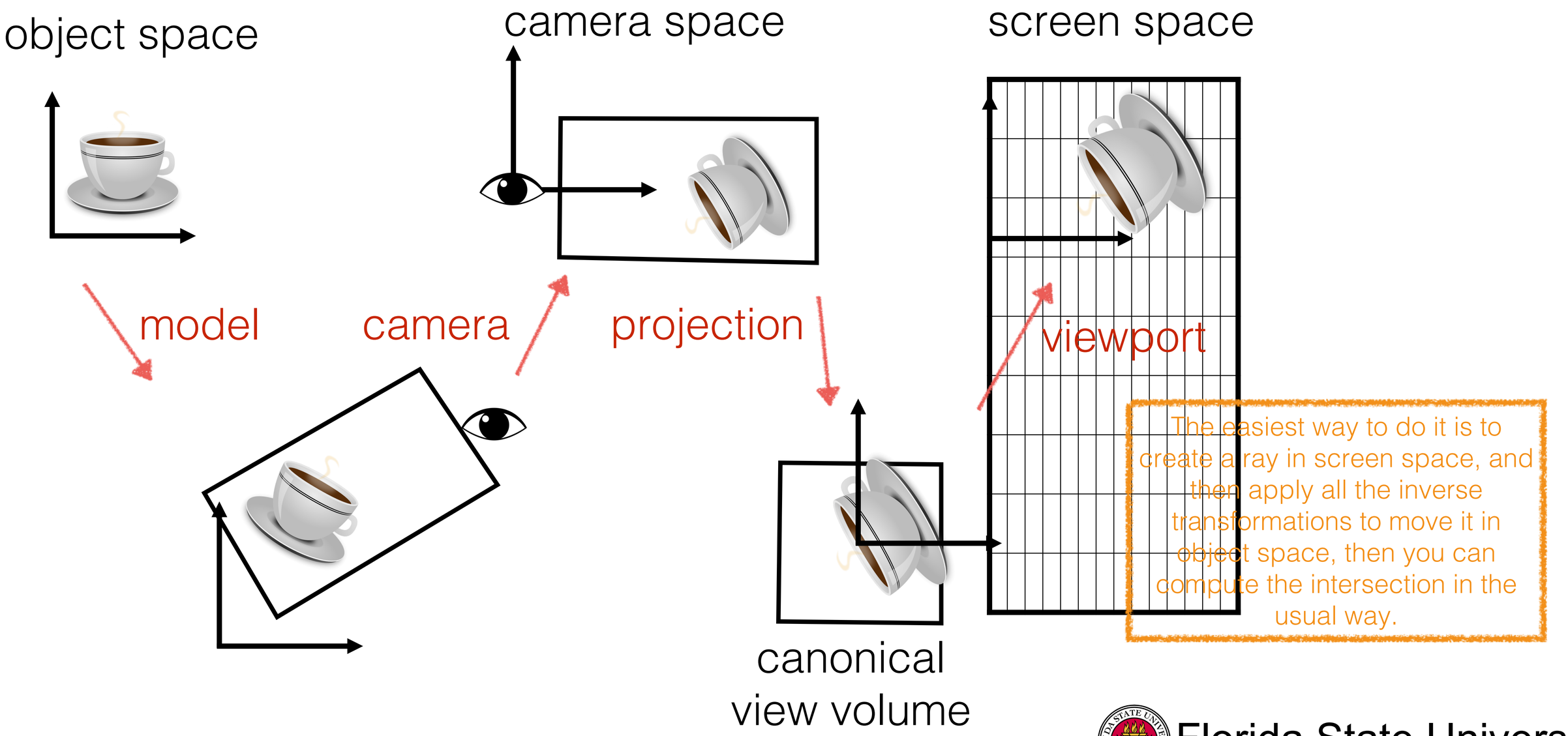



```
glViewport(x,y,width,height);
```

Picking Objects

- To interact with the scene, it is common to “pick” or “select” objects in the scene
- The most common way to do it is to cast a ray, starting from the point where the mouse is and going “inside” the screen
- The first object that is hit by the ray is going to be the selected object
- For picking in the exercises, you can reuse the code that you developed in the first assignment
- You must account for viewing and model transformations!

Picking via Ray Casting



Tests

- The tests determine if a fragment will effect the color in the framebuffer or if it should be discarded
- There are two tests:
  - Depth Test - Discards all fragments with a z coordinate ***bigger*** than the value in the depth buffer
  - Stencil Test - An additional buffer that is fully customizable

Depth Test

- To use the depth test you need to do two things:
  - Enable it

```
glEnable(GL_DEPTH_TEST);
```

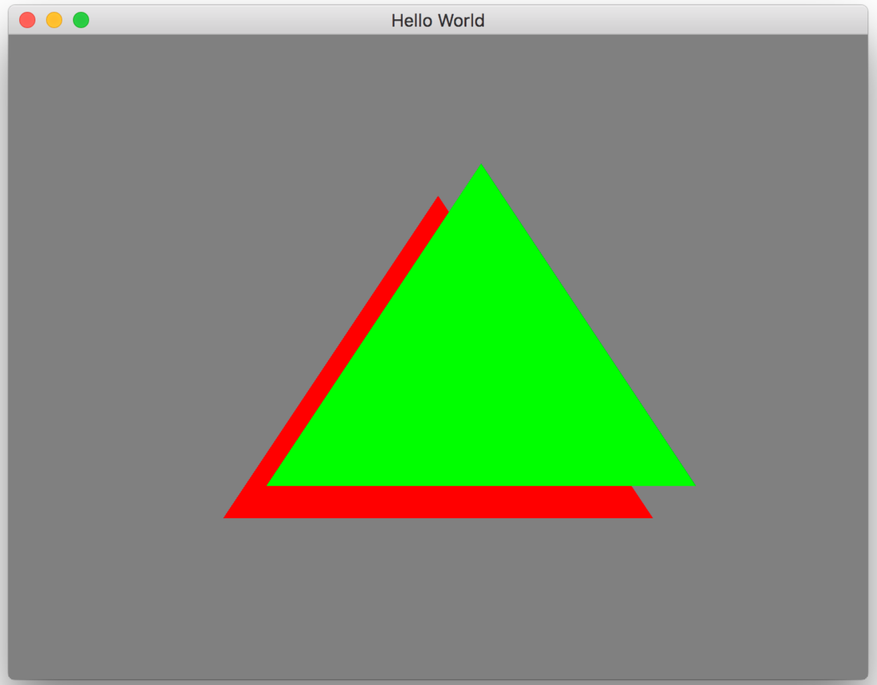
  - Remember to clear the depth buffer

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

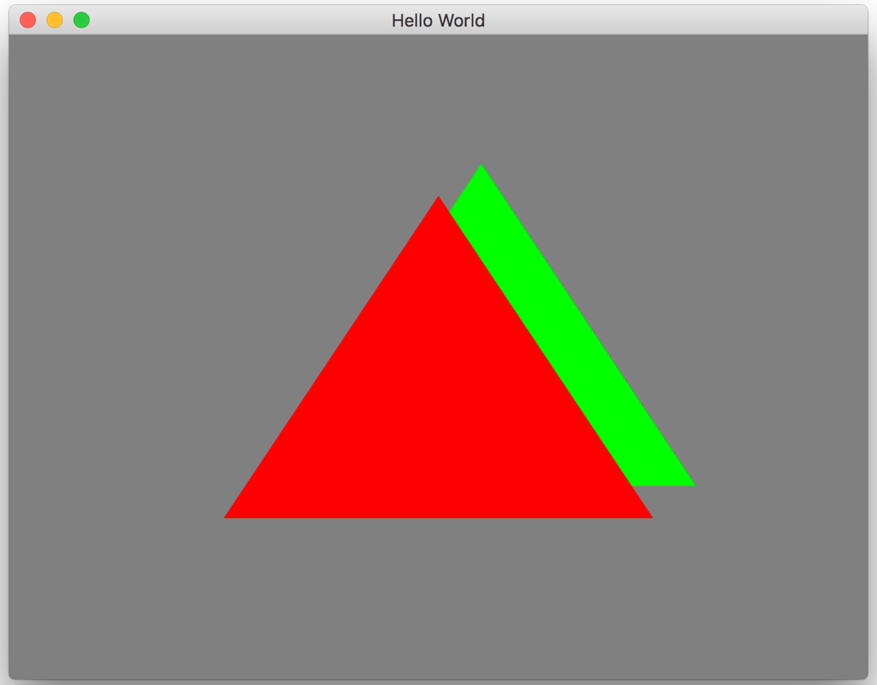
Live Coding

- Let’s draw a couple of triangles with depth test active
- See example in Assignment_2/extra/main_depth.cpp

main_depth.cpp



Without Depth Test

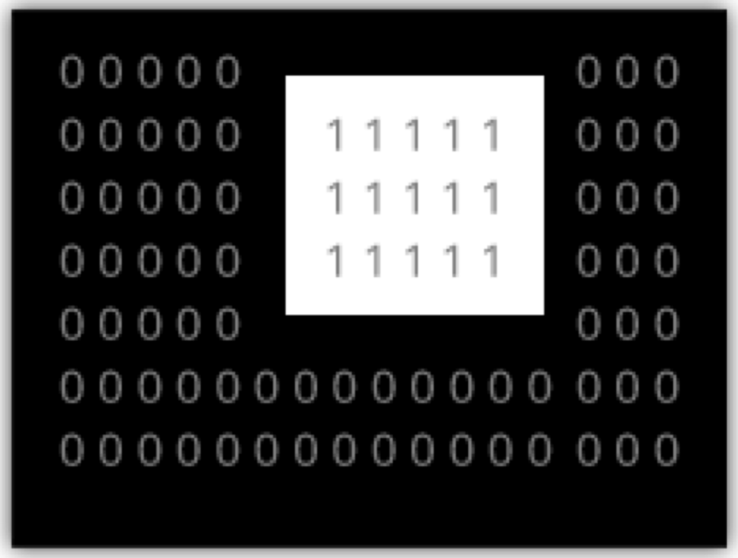


With Depth Test

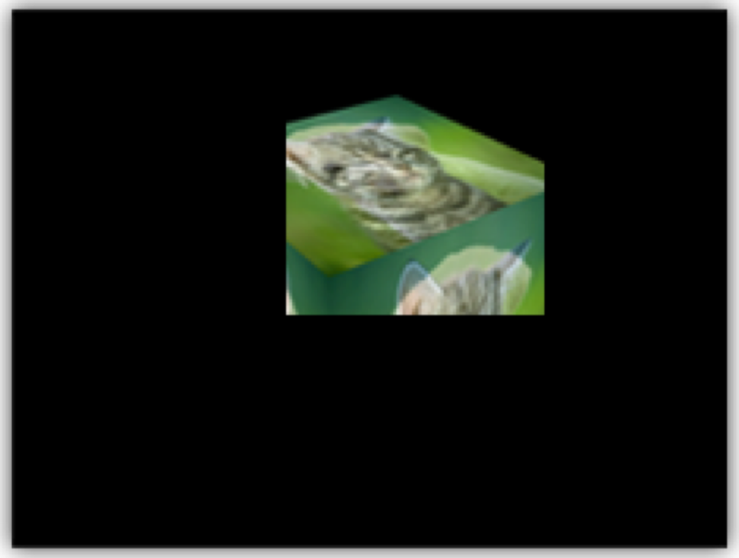
Stencil Buffer/Test

- The stencil buffer behaves similarly to the depth buffer, but you can customize how the test is performed and how the stencil buffer is updated

Color buffer without stencil test

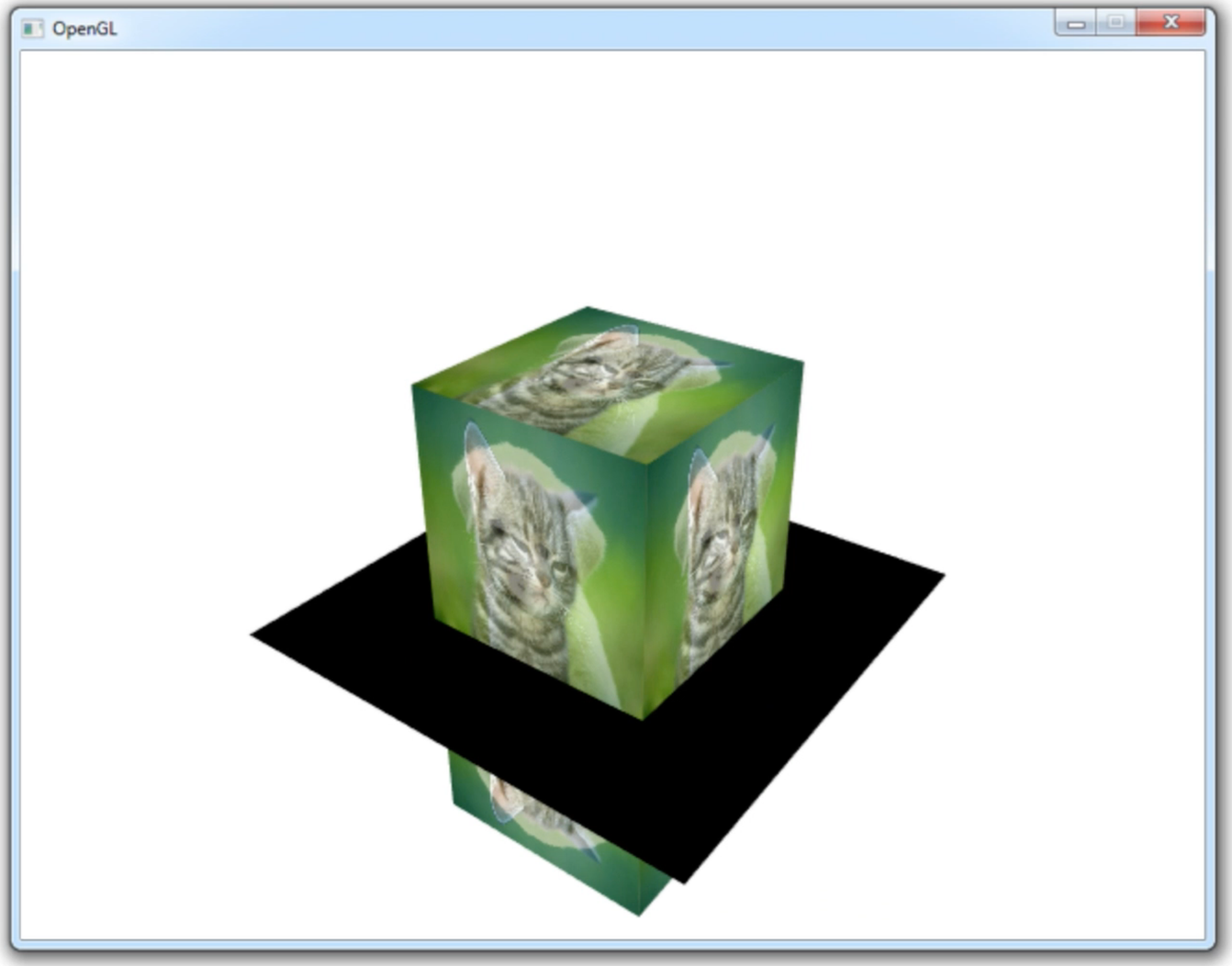


Stencil buffer

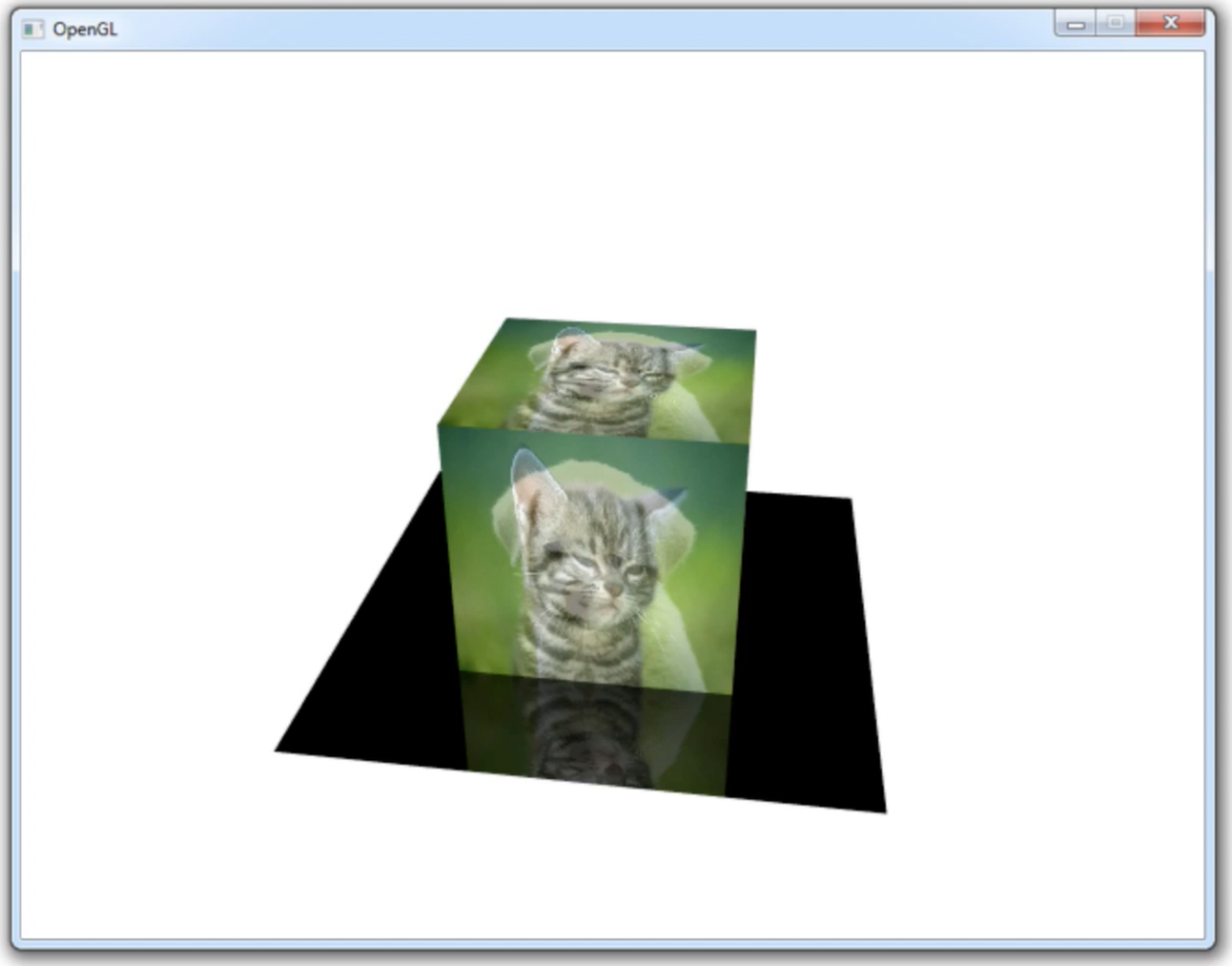


Color buffer with stencil test

Stencil Buffer



Stencil Buffer



Blending

- After a fragment is accepted, it overwrites the existing pixel in the framebuffer
- It is possible to enable “blending”, which allows to compute a new pixel combining the value of the new fragment and of the existing pixel color

Blending Formula

$$\mathbf{P} = f_{src}\mathbf{C}_{src} + f_{dst}\mathbf{C}_{dst}$$

- Enable it with:

```
glEnable(GL_BLEND);
```

- and select the factors *f* with:

```
glBlendFunc(GLenum sfactor, GLenum dfactor)
```

- *f* can be a scalar or a vector

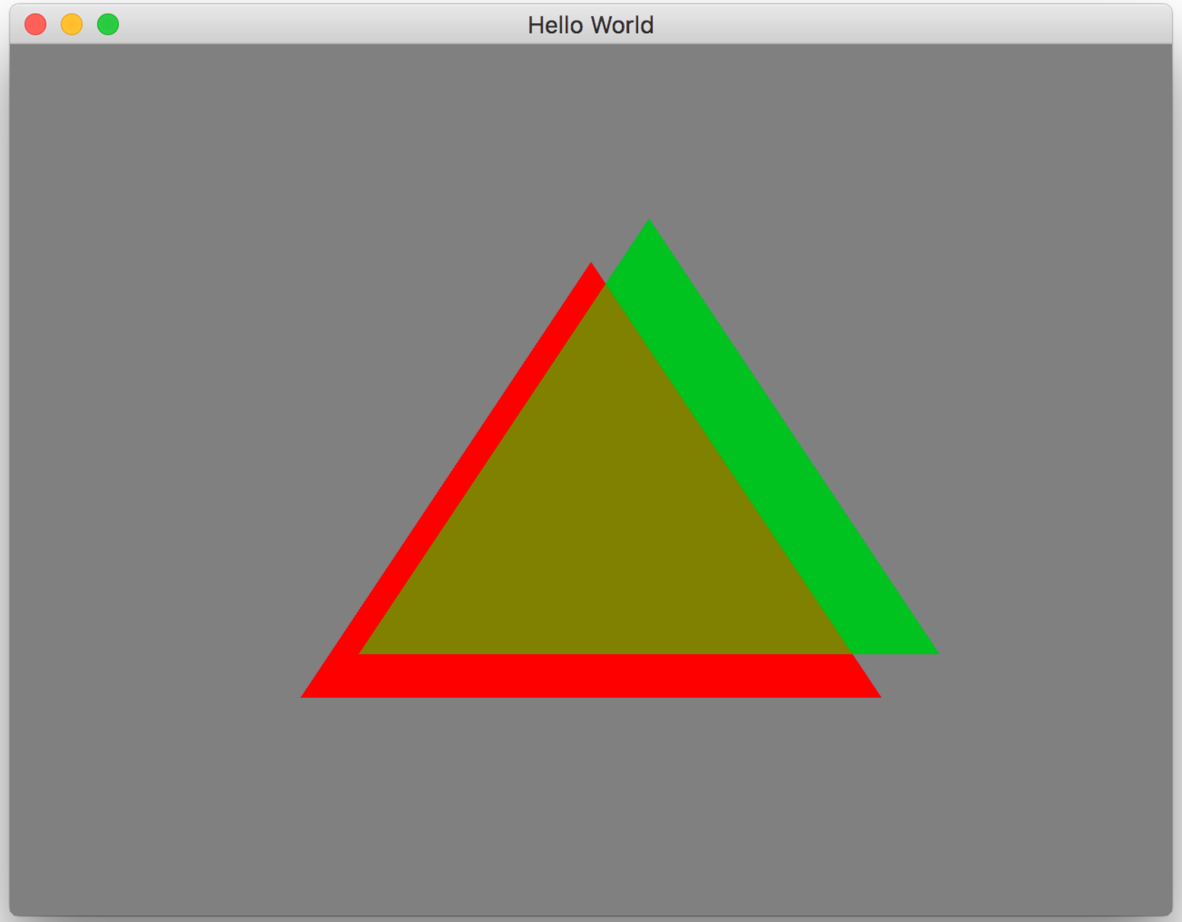
Blending Factor

GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector source
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector
GL_DST_COLOR	Factor is equal to the destination color vector
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector
GL_SRC_ALPHA	Factor is equal to the alpha component of the source color vector
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to 1–alpha of the source color vector
GL_DST_ALPHA	Factor is equal to the alpha component of the destination color vector
GL_ONE_MINUS_DST_ALPHA	Factor is equal to 1–alpha of the destination color vector
GL_CONSTANT_COLOR	Factor is equal to the constant color vector
GL_ONE_MINUS_CONSTANT_COLOR	Factor is equal to 1 - the constant color vector
GL_CONSTANT_ALPHA	Factor is equal to the alpha component of the constant color vector
GL_ONE_MINUS_CONSTANT_ALPHA	Factor is equal to 1-alpha of the constant color vector

Live Coding

- Let’s see how to render a transparent triangle
- See example in Assignment_2/extra/main_blending.cpp

main_blending.cpp



Framebuffers

```
glBindFragDataLocation(shaderProgram, 0, “outColor");
```

- This line binds the output of the fragment shader to the video buffer
- You can instead create a framebuffer object and save the produced image in GPU memory for further processing
- We will come back to this when we will study texture mapping

References

https://open.gl — Main reference

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 17