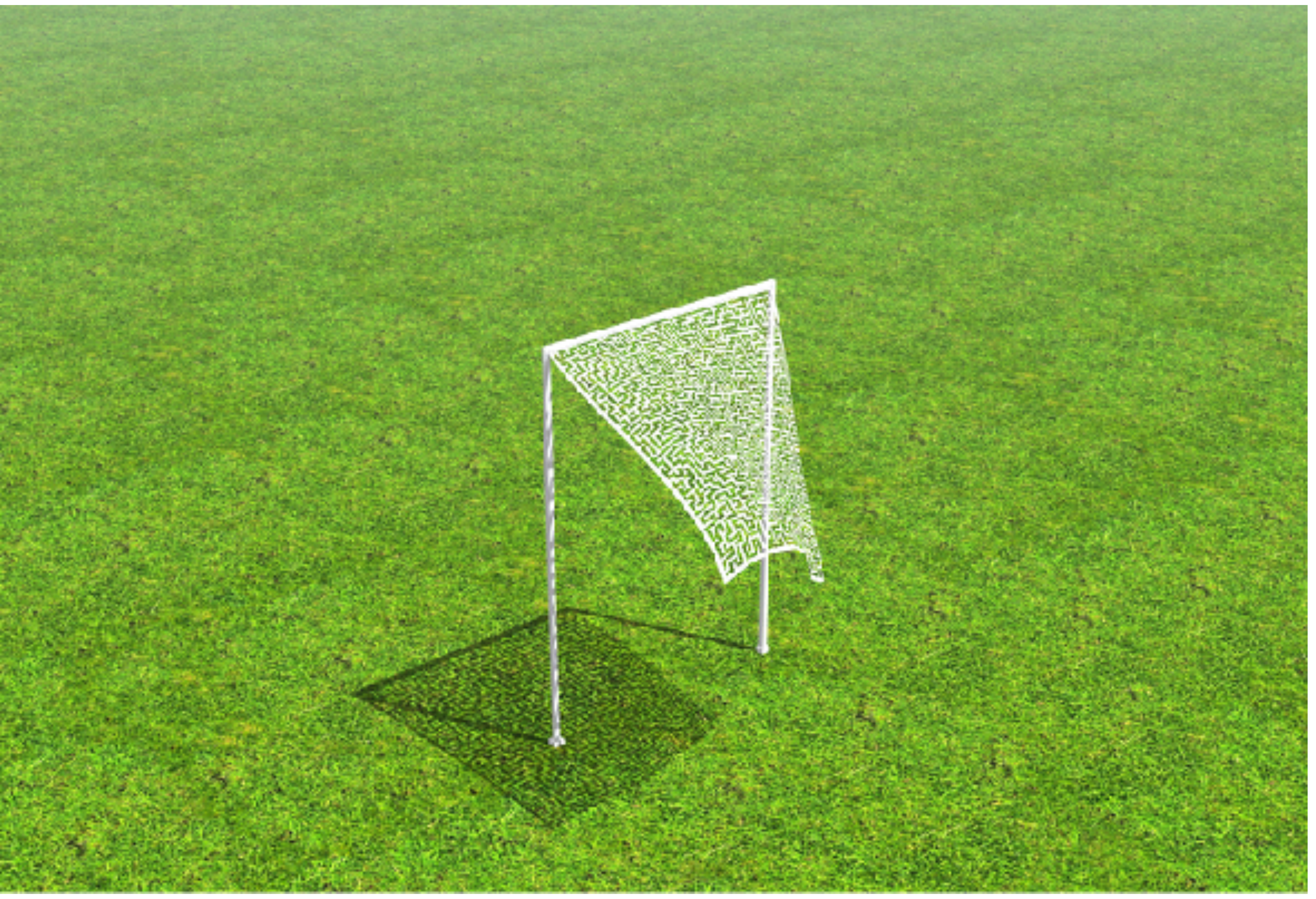
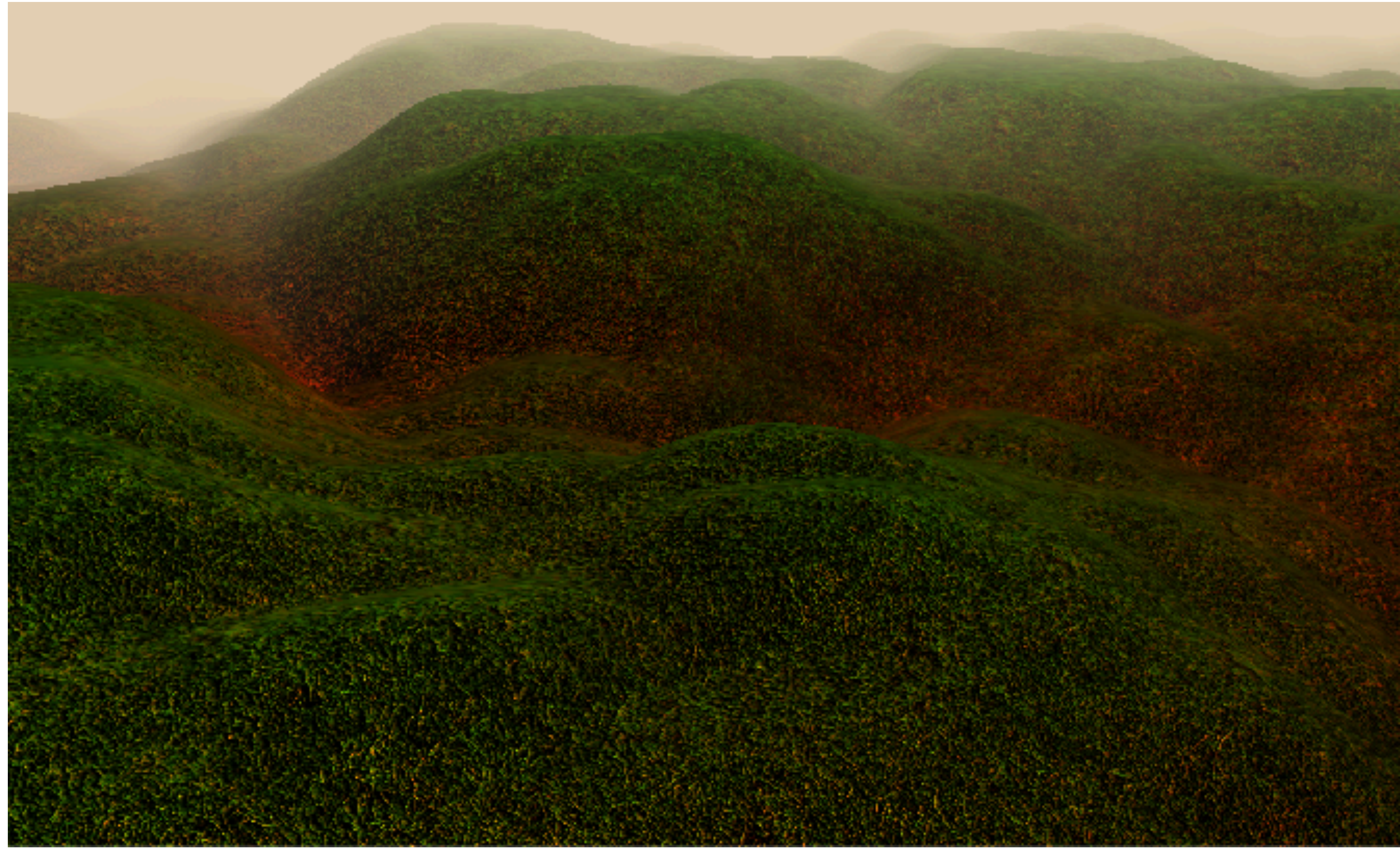
COSC342: Computer Graphics

2017



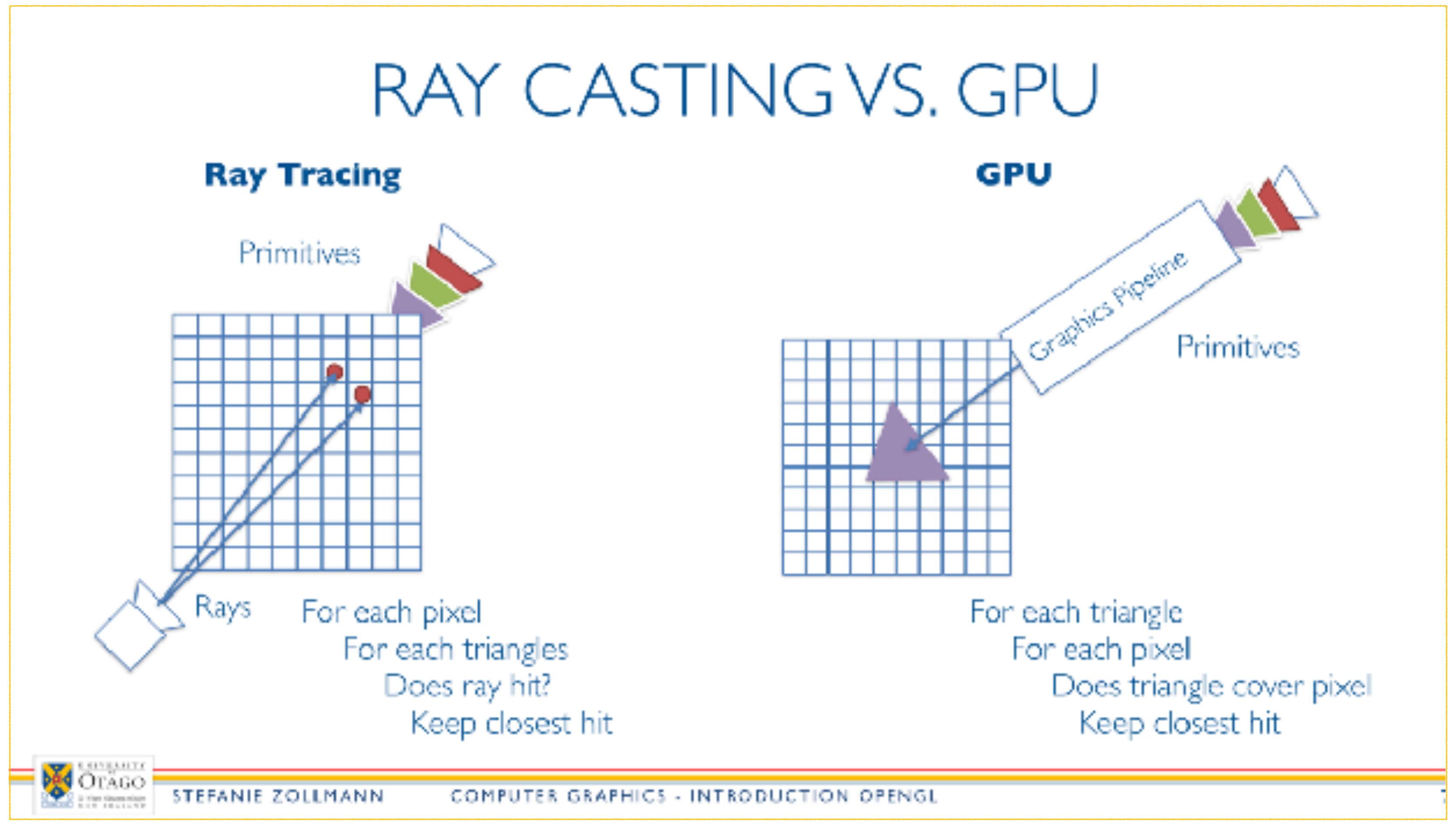


Lecture 14

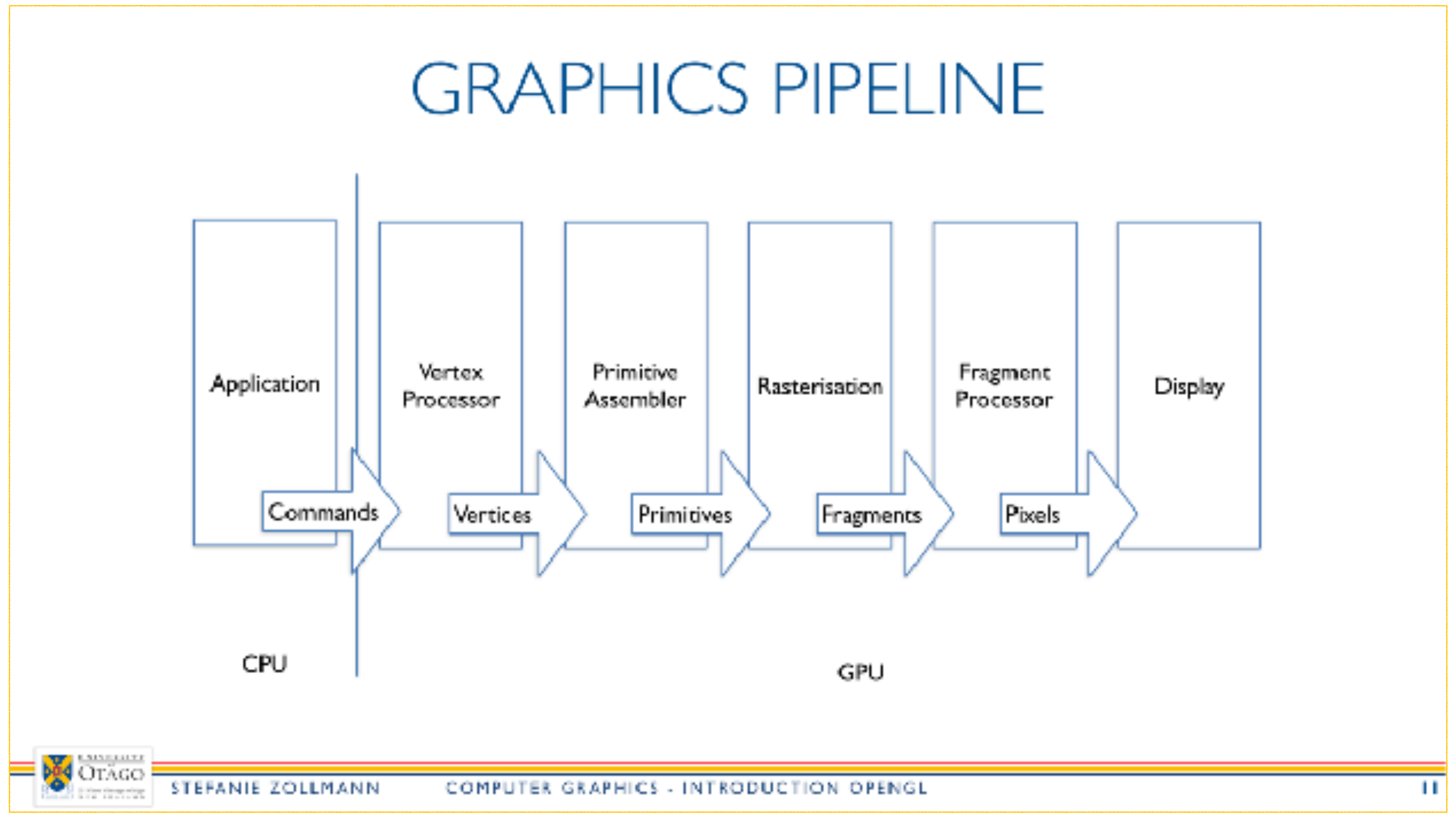
OPENGL ESSENTIALS

Stefanie Zollmann

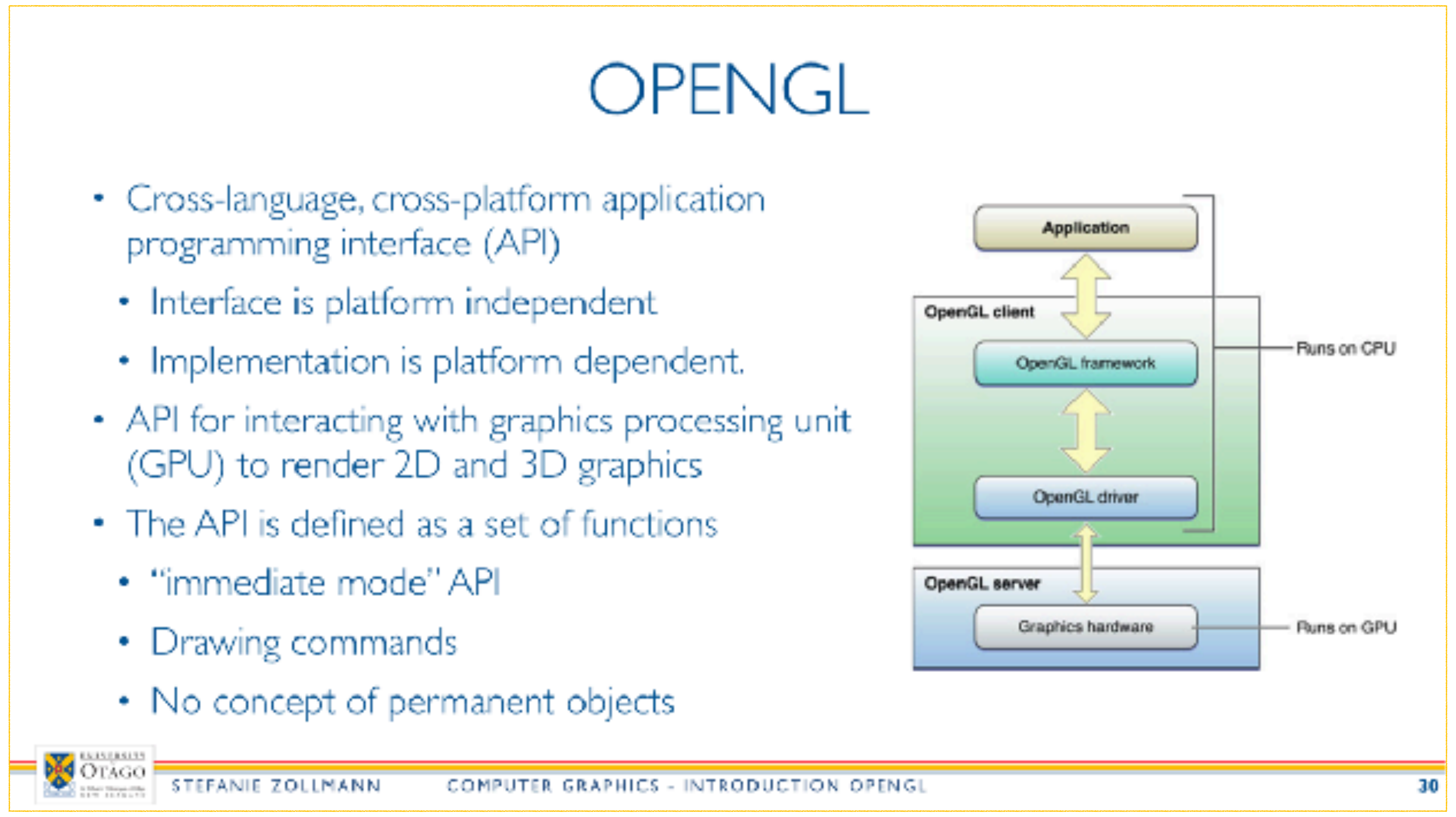
LAST LECTURE



Rasterisation vs. Raytracing

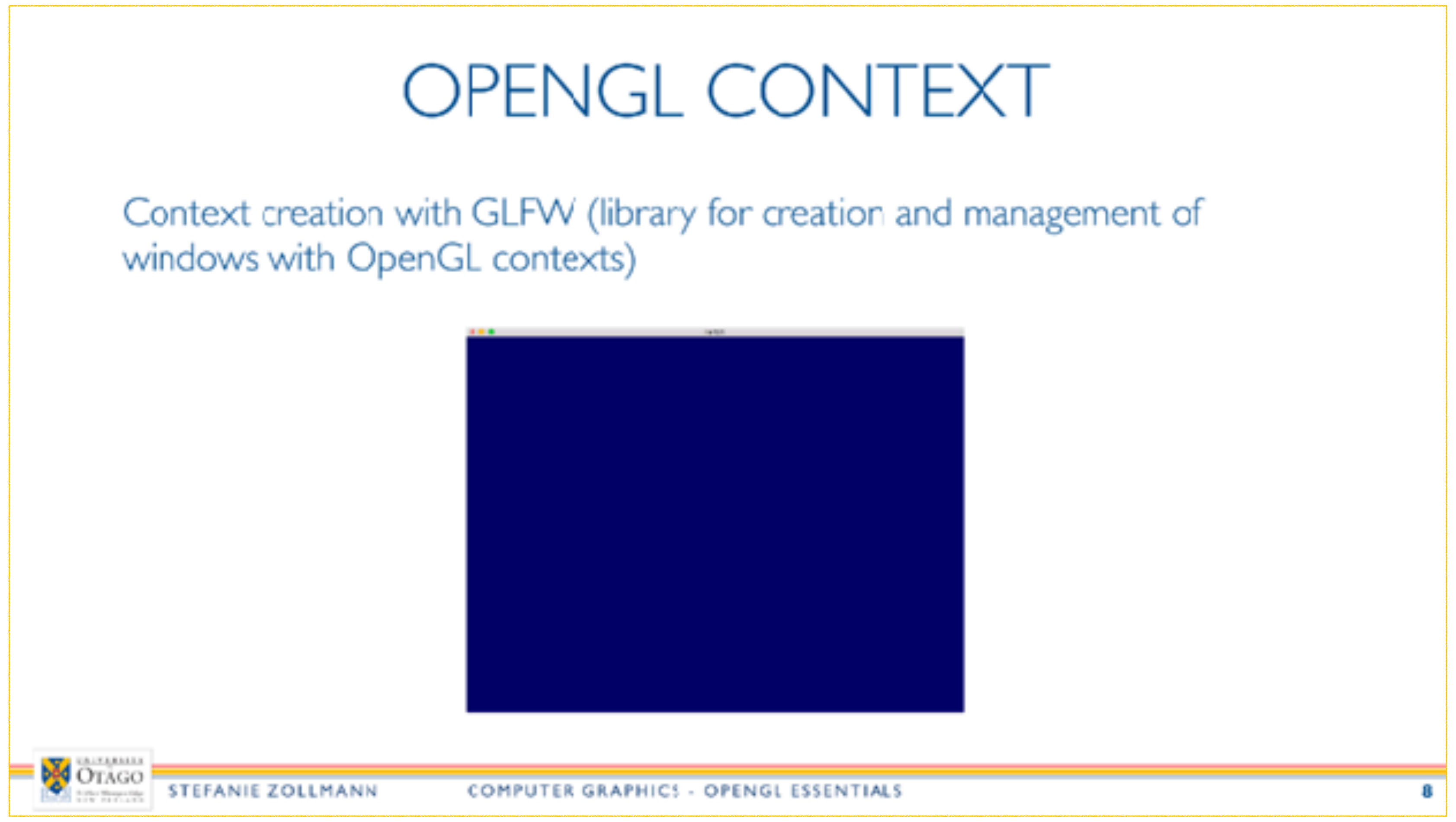


The Graphics Pipeline

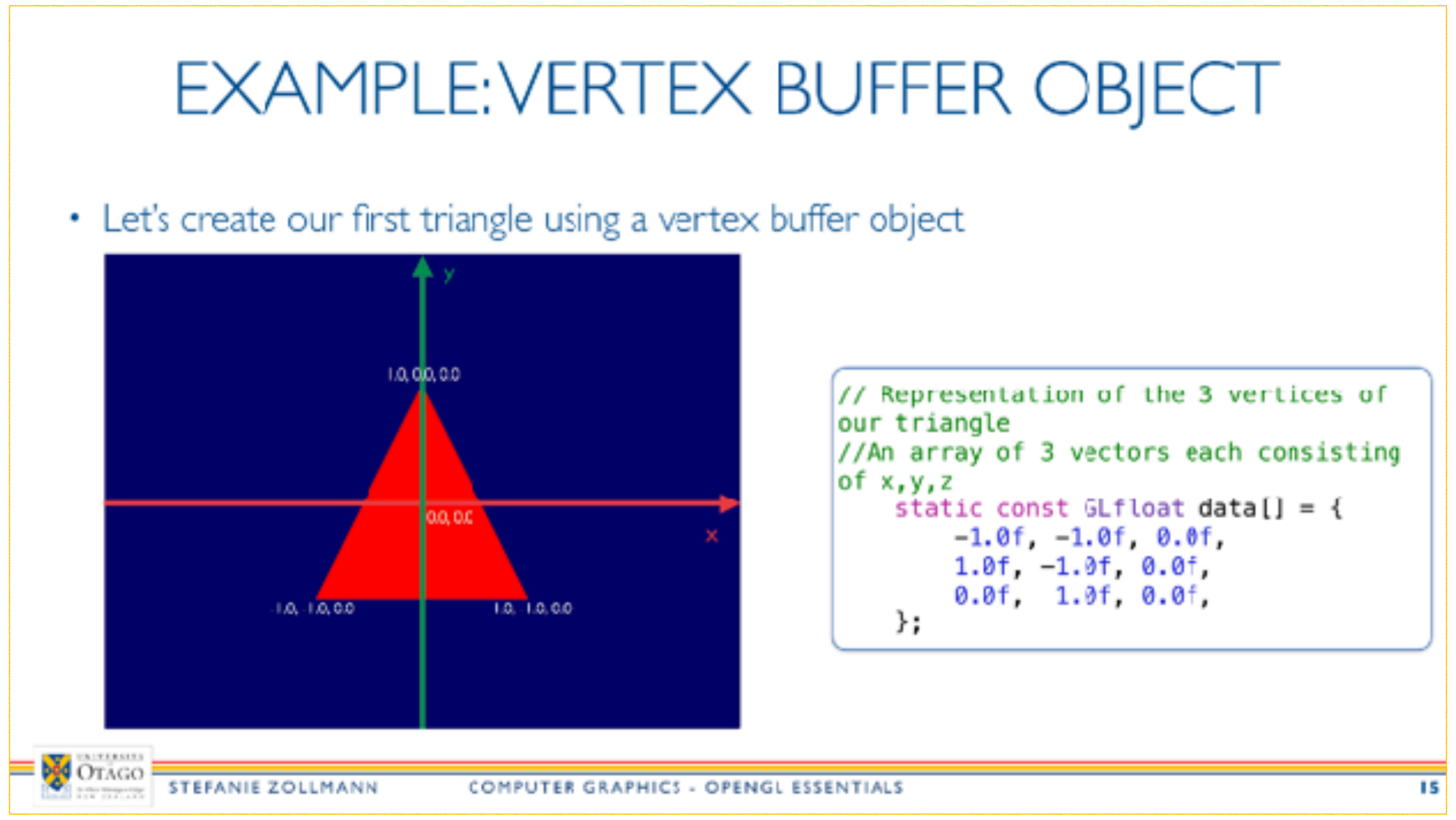


OPENGL

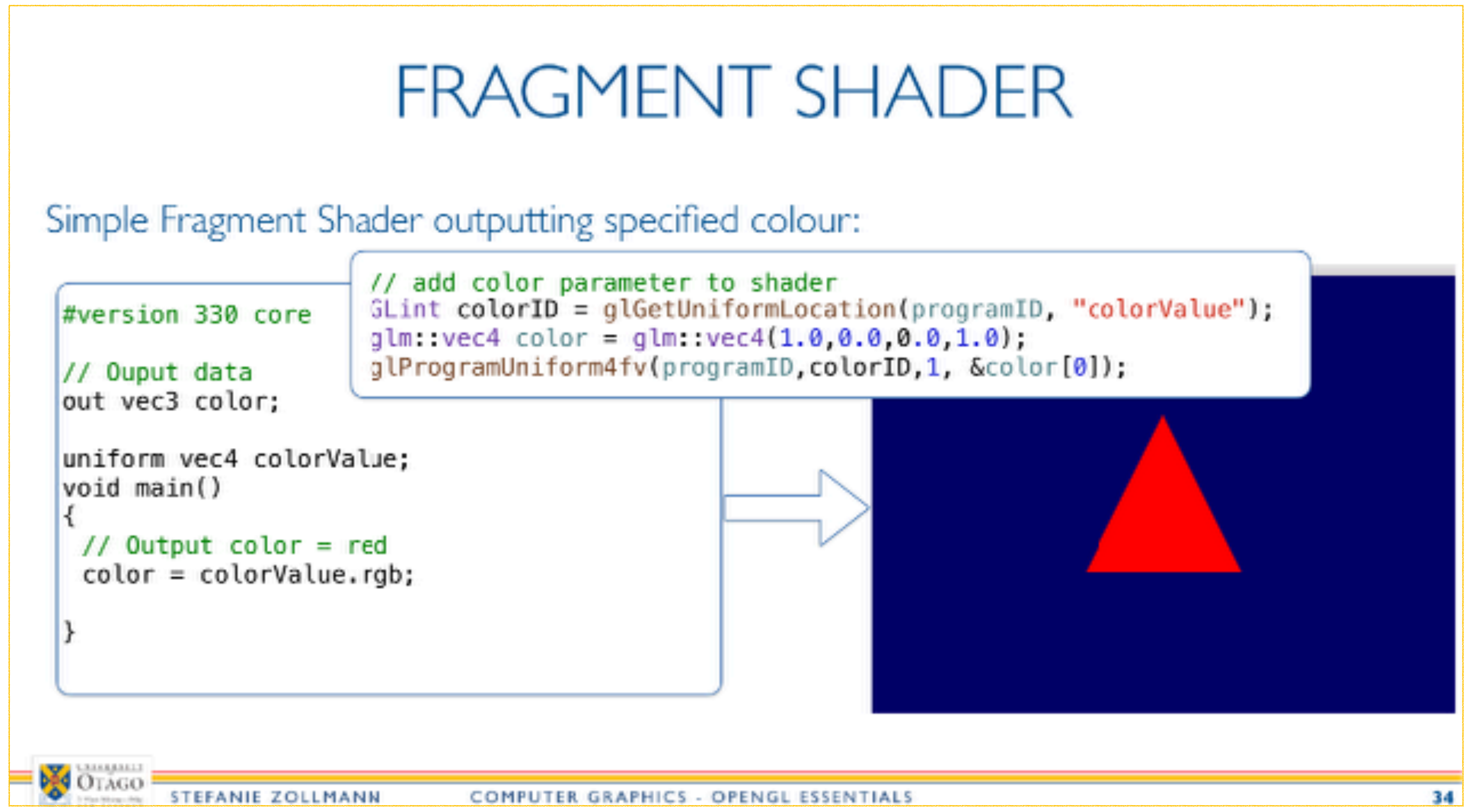
TODAY



OpenGL Context



OpenGL Objects



Shaders

OPENGL - HISTORY

- Originally released by Silicon Graphics Inc. (SGI) in 1992
- Now managed by non-profit technology consortium Khronos Group
- OpenGL has been through a number of revisions
- Significant changes:
  - **OpenGL 2.0** incorporates the significant addition of the OpenGL Shading Language (also called GLSL)
  - **OpenGL 3.0** first major API revision deprecated fixed-function vertex and fragment processing and direct-mode rendering, using glBegin and glEnd

Important:

- This lecture uses "modern" OpenGL (version 3.x and higher, latest is 4.5)

OPENGL CONCEPTS

- OpenGL Context
- State
- OpenGL Object Model

OPENGL CONTEXT

- Represents an instance of OpenGL
- Context stores all of the state associated with this instance of OpenGL
- A process can have multiple contexts
  - Each represent separate viewable surface (e.g. a window)
  - Each has own OpenGL Objects
  - Multiple contexts can share resources.

OPENGL CONTEXT

- A context can be current for a given thread.
- One to one mapping
  - Only one current context per thread
  - Context only current in one thread at the same time
- OpenGL operations work on the current context

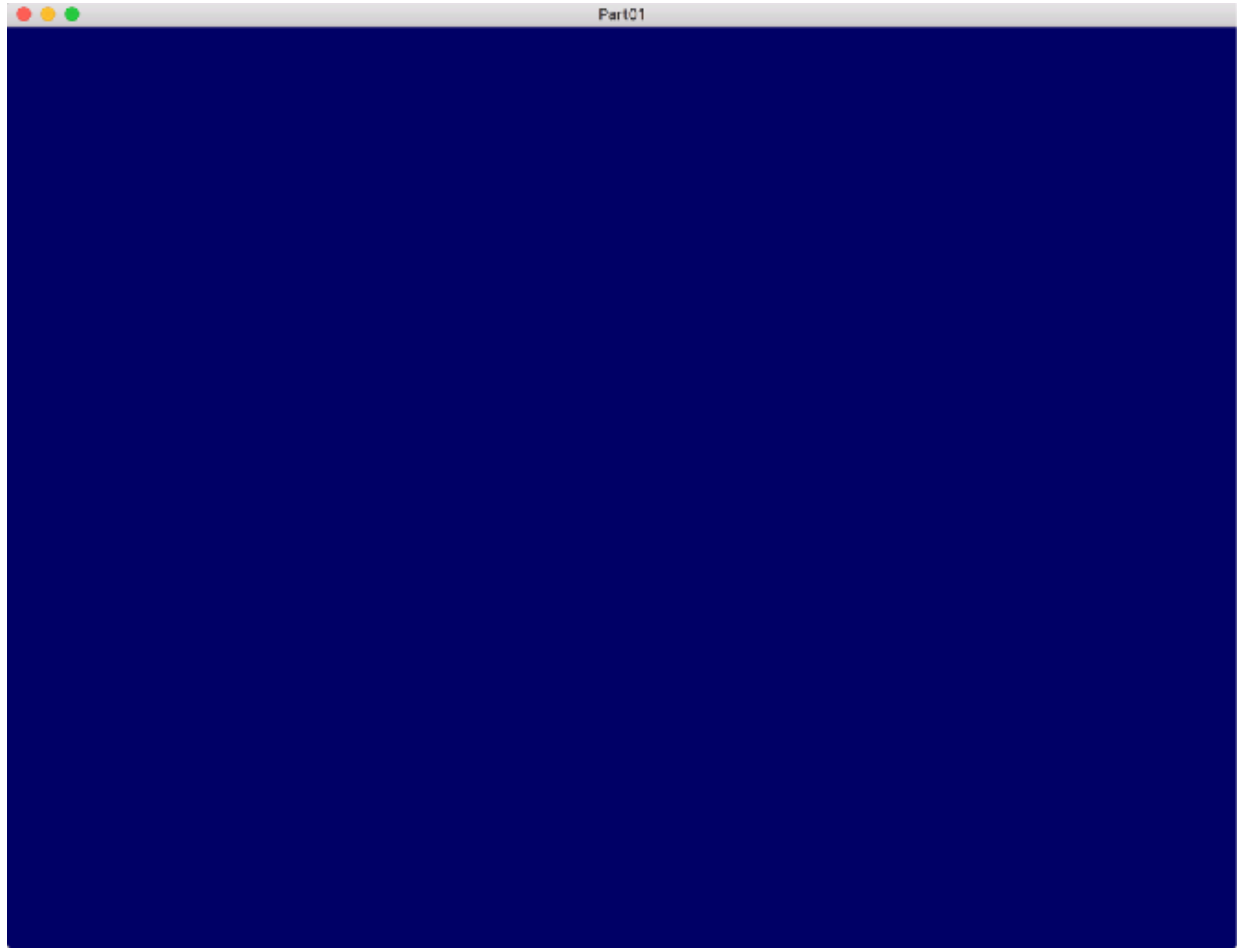
OPENGL CONTEXT

Context creation with GLFW (library for creation and management of windows with OpenGL contexts)

```
 // Open a window and create its OpenGL context
window = glfwCreateWindow( 1024, 768, windowName.c_str(), NULL, NULL);
if( window == NULL ){
    fprintf( stderr, "Failed to open GLFW window. \n” );
    getchar();
    glfwTerminate();
    return false;
}
// set the context as current
glfwMakeContextCurrent(window);
```

OPENGL CONTEXT

Context creation with GLFW (library for creation and management of windows with OpenGL contexts)



OPENGL STATE

- Information that the context contains and that is used by the rendering system
- A piece of state is simply some value stored in the OpenGL context
- OpenGL as "state machine"
- When a context is created, state is initialised to default values

```
// Enable depth test
glEnable(GL_DEPTH_TEST);
// Enable blending
glEnable(GL_BLEND);

// Disable depth test
glDisable(GL_DEPTH_TEST);
```

Examples

OBJECT MODEL

- OpenGL is "object oriented"
- Object instances are identified by a name
  - Unsigned integer handle (GLuint)
  - References that identify an object (no pointers)
- Commands work on targets
  - Each target has an object currently bound to the target
  - To modify objects, you must first bind them to the OpenGL context, then execute command

```
GLuint m_textureID;
// Create texture
glGenTextures(1, &m_textureID);

// "Bind" texture
glBindTexture(GL_TEXTURE_2D,
m_textureID);
```

GLuint

OBJECT MODEL

- OpenGL is "object oriented"
- Object instances are identified by a name
  - Unsigned integer handle (GLuint)
  - References that identify an object (no pointers)
- Commands work on targets
  - Each target has an object currently bound to the target
  - To modify objects, you must first bind them to the OpenGL context, then execute command

```
GLuint m_textureID;
// Create texture
glGenTextures(1, &m_textureID);

// "Bind" texture
glBindTexture(GL_TEXTURE_2D,
m_textureID);
```

GLuint

Object oriented?

- target⇔type
- commands⇔methods

OPENGL OBJECTS

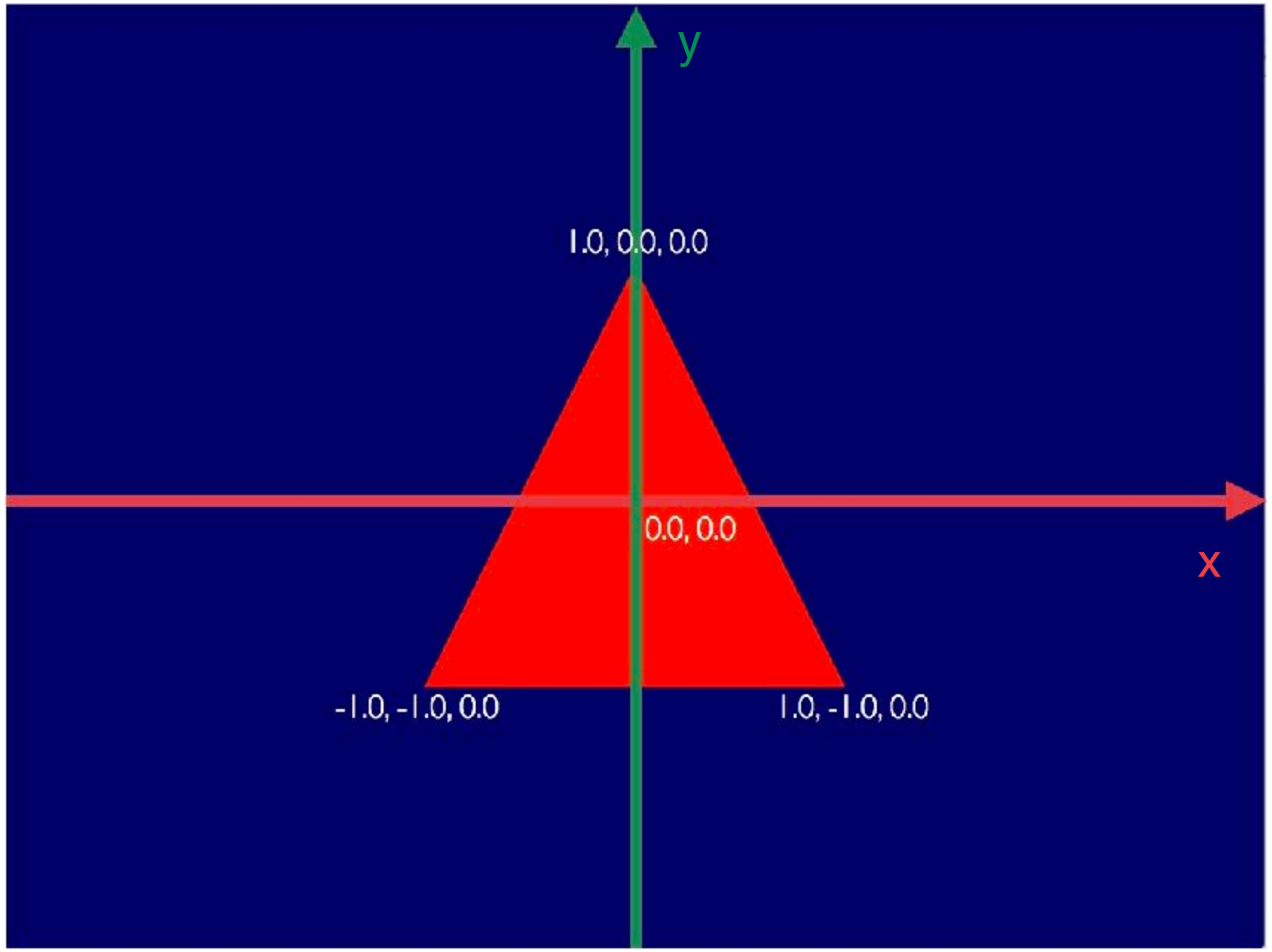
- **Act as**
  - Sources of input
  - Sinks for output
- **Examples**:
  - Buffer
  - Images
  - State objects

OPENGL OBJECTS

- **Buffer objects**
  - Unformatted chunks of memory
  - Can store vertex data (VBO) or pixel data, etc.
- **Textures**
  - 1D, 2D, or 3D arrays of texels
  - Can be used as input for texture sampling
- **Vertex Array Objects**
  - Stores all of the state needed to supply vertex data (vertex data + format)
- **Framebuffer Objects**
  - User-defined framebuffers that can be rendered to

EXAMPLE: VERTEX BUFFER OBJECT

- Let's create our first triangle using a vertex buffer object




```
// Representation of the 3 vertices of our triangle
// An array of 3 vectors each consisting of x,y,z
    static const GLfloat data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        0.0f,  1.0f, 0.0f,
    };
```

EXAMPLE: VERTEX BUFFER OBJECT

```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
 std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                      0,                    // attribute
                      3,                    // size
                      GL_FLOAT,             // type
                      GL_FALSE,             // normalized?
                      0,                    // stride - 0= tightly packed
                      (void*)0              // array buffer offset
                      );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
 glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

STRIDE

- Specifies the byte offset between consecutive generic vertex attributes (if stride equals 0 -> means tightly packed)

```
GLfloat data[] = { -1.0f, -1.0f, 0.0f, 1.0f, -1.0f, 0.0f, 0.0f,  1.0f, 0.0f};
```

- Tightly packed:

X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
Vertex 0			Vertex 1			Vertex 2			Vertex 3		
Stride = 3x4bytes = 12											

- Interleaved:

X	Y	Z	R	G	B	X	Y	Z	R	G	B
Vertex 0			Color 0			Vertex 1			Color 1		
Stride = 3x4bytes + 3x4bytes = 24											

EXAMPLE: VERTEX BUFFER OBJECT

```
//  - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - -
 std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                      0,                    // attribute
                      3,                    // size
                      GL_FLOAT,             // type
                      GL_FALSE,             // normalized?
                      0,                    // stride - 0= tightly packed
                      (void*)0              // array buffer offset
                      );
//  - - - - - - - - - - - - - Actual drawing call - - - - - - - -
 glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

DRAW CALL

- After creating and loading data:
  - We use the draw call to actually draw something

```
glDrawArrays(GL_TRIANGLES, 0, m_vertices.size());
```

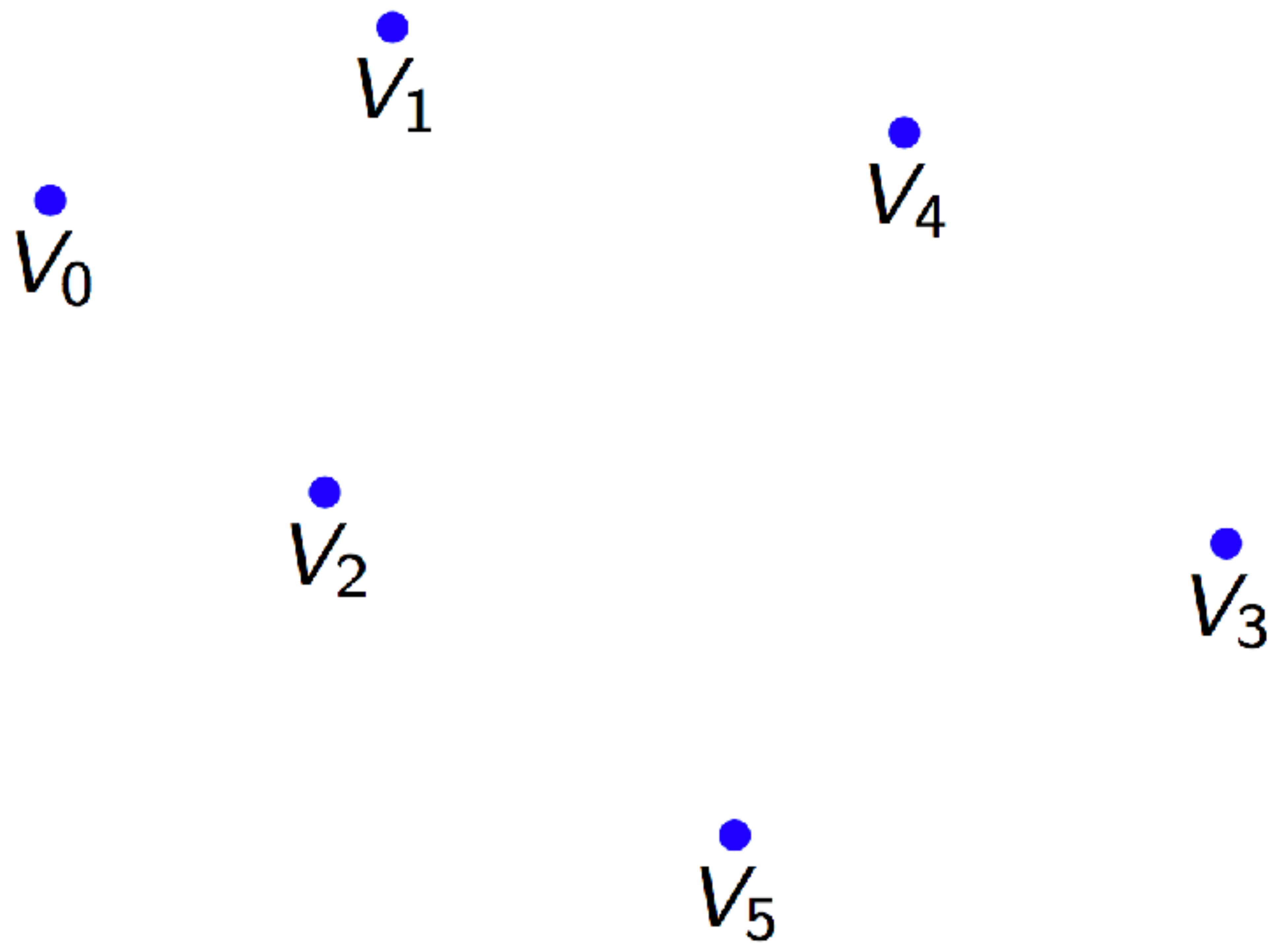
mode specifies what kind of primitives to render. e.g. GL_TRIANGLES

specifies the starting index in the enabled arrays.

count specifies the number of indices to be rendered.

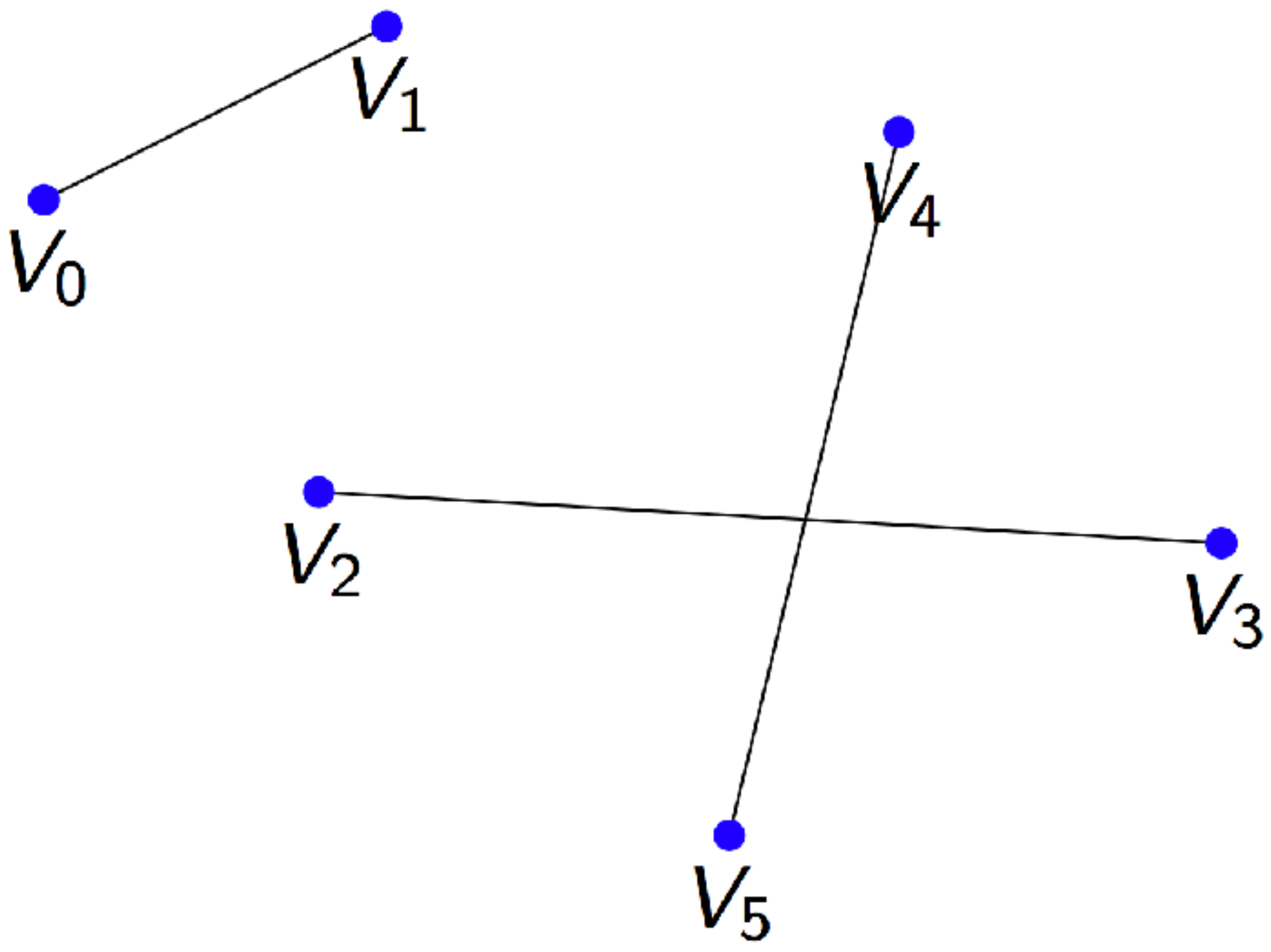
PRIMITIVE TYPES

- **Points**
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



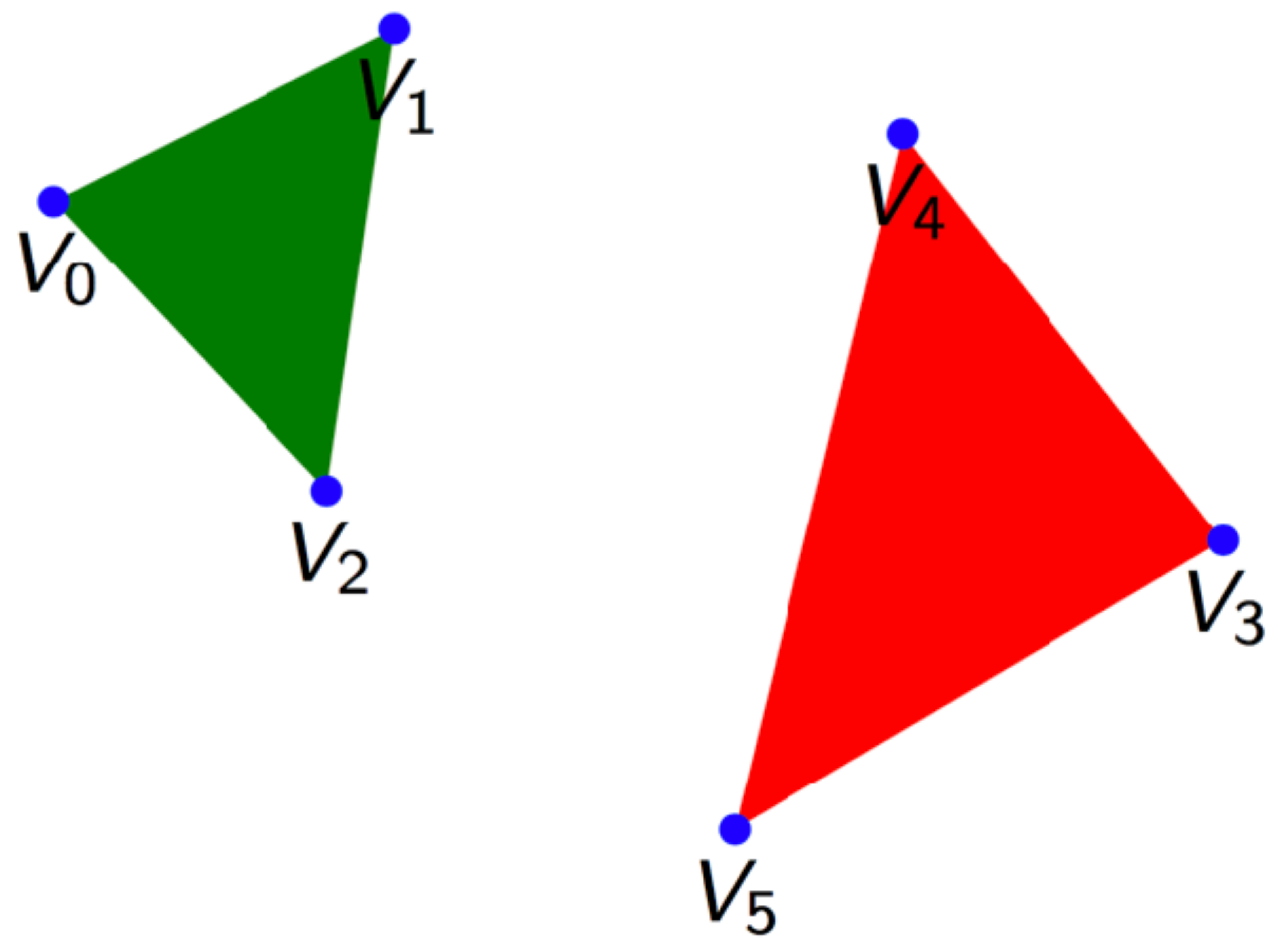
PRIMITIVE TYPES

- Points
- **Lines**
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



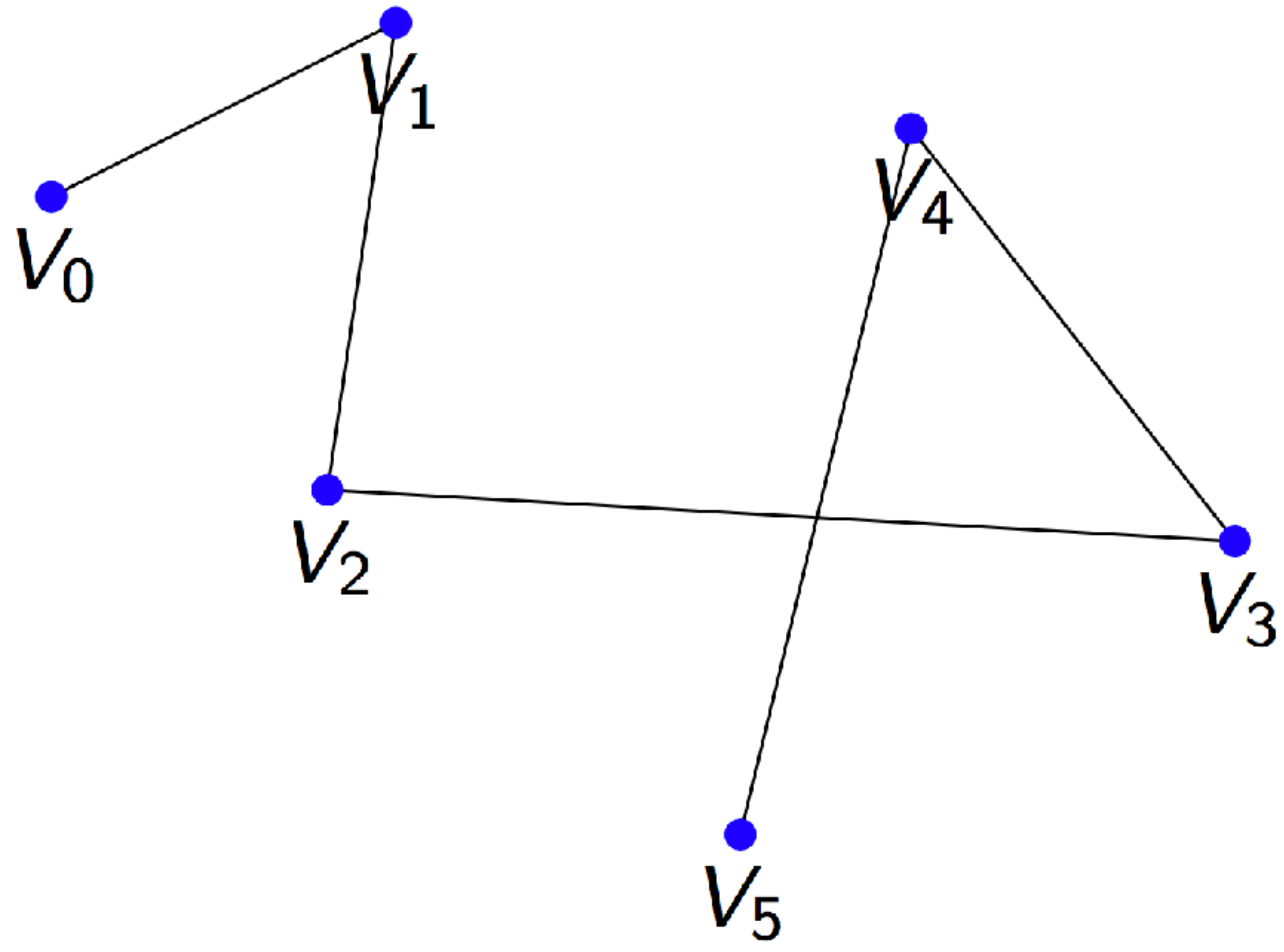
PRIMITIVE TYPES

- Points
- Lines
- **Triangles**
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



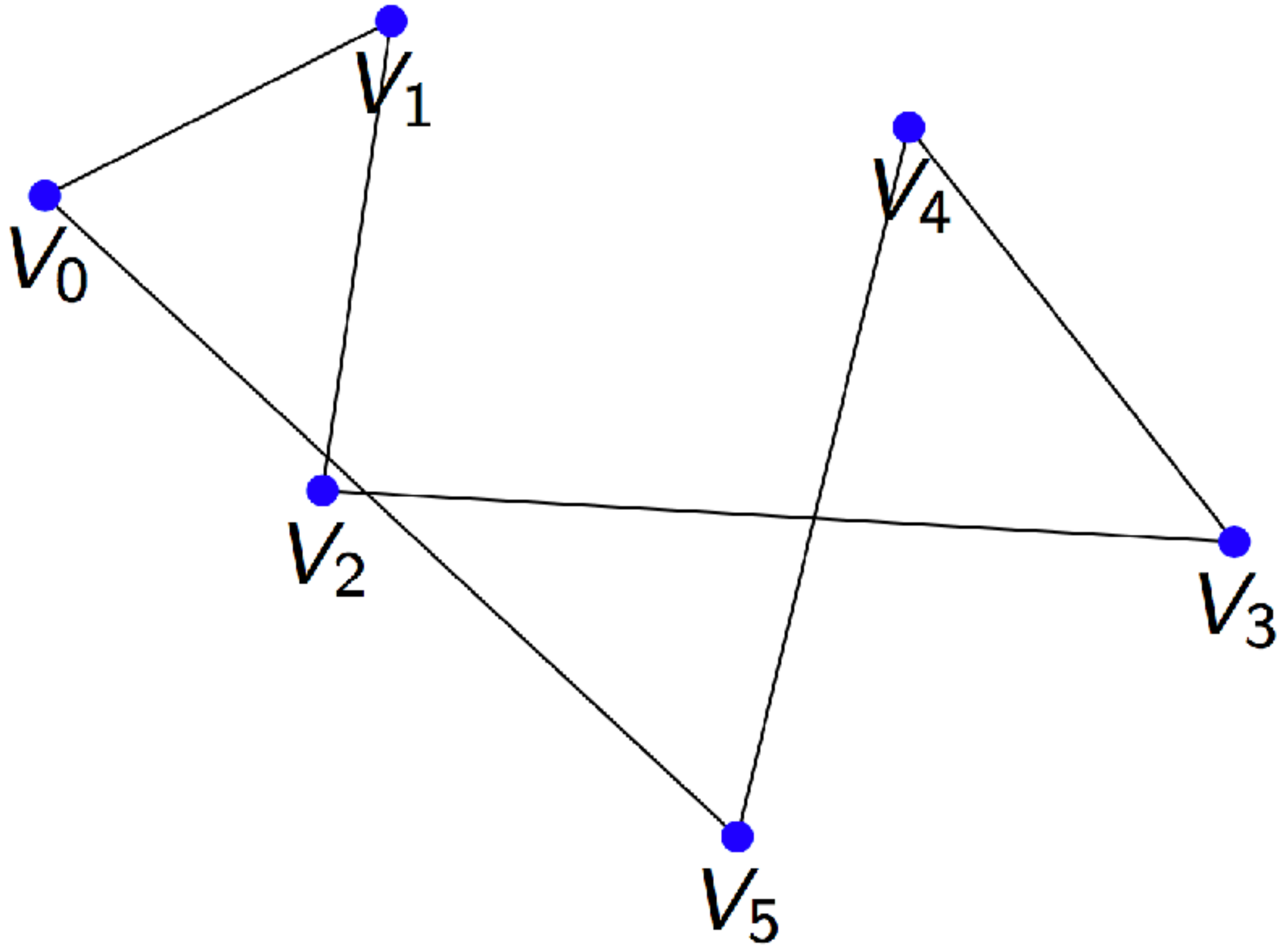
PRIMITIVE TYPES

- Points
- Lines
- Triangles
- **Line strips**
- Lineloops
- Triangle strips
- Triangle fans



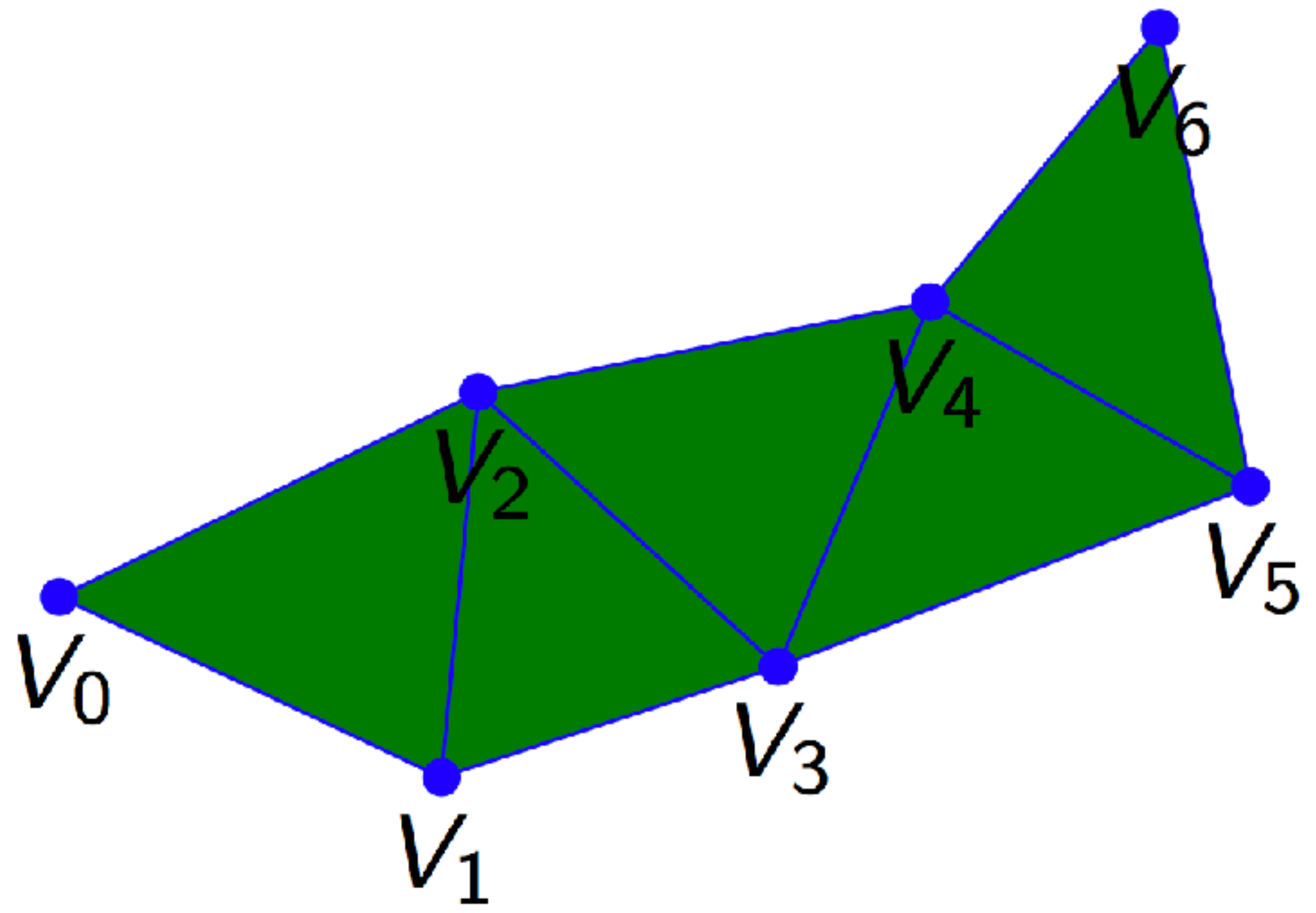
PRIMITIVE TYPES

- Points
- Lines
- Triangles
- Line strips
- **Lineloops**
- Triangle strips
- Triangle fans



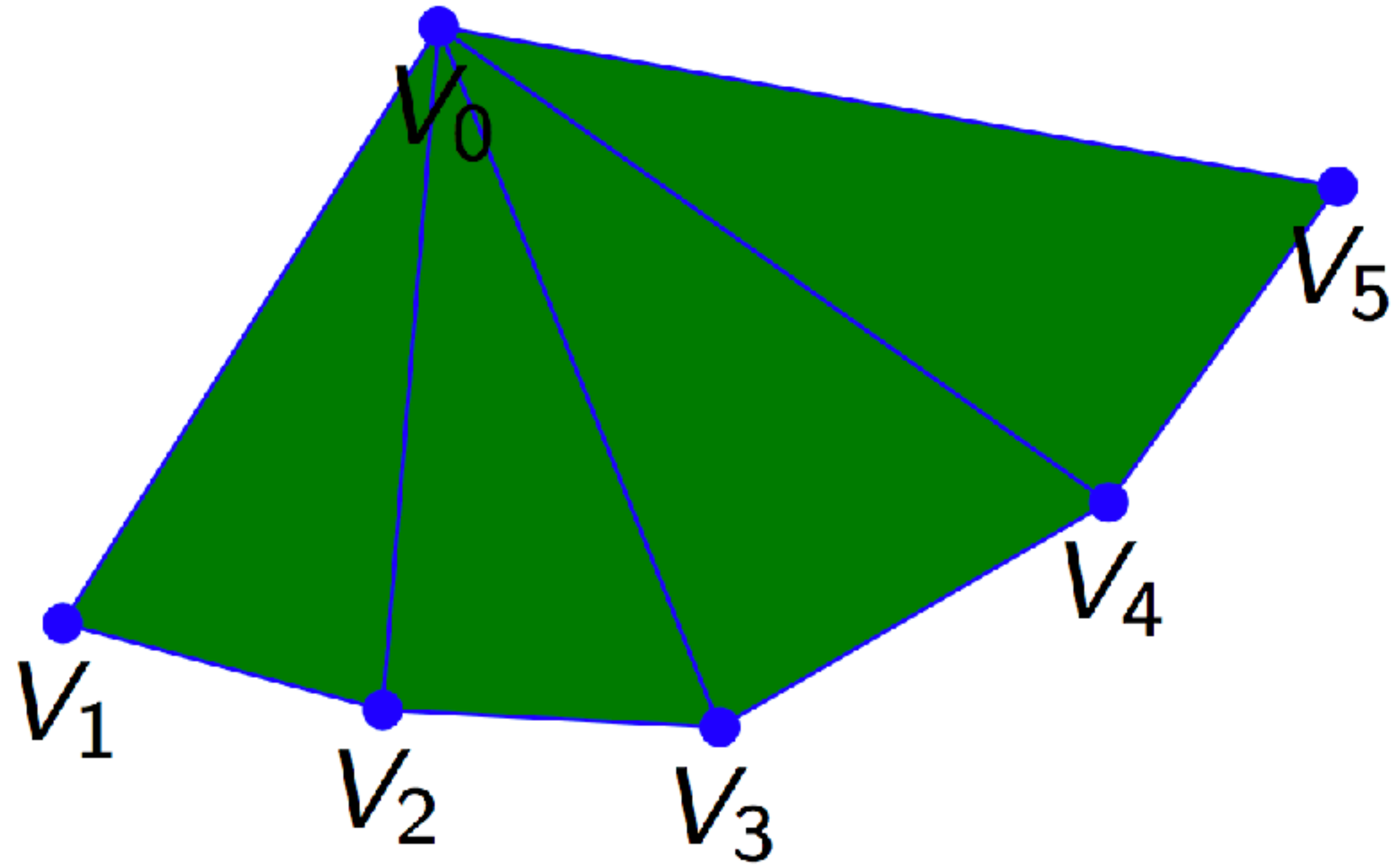
PRIMITIVE TYPES

- Points
- Lines
- Triangles
- Line strips
- Lineloops
- **Triangle strips**
- Triangle fans



PRIMITIVE TYPES

- Points
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- **Triangle fans**

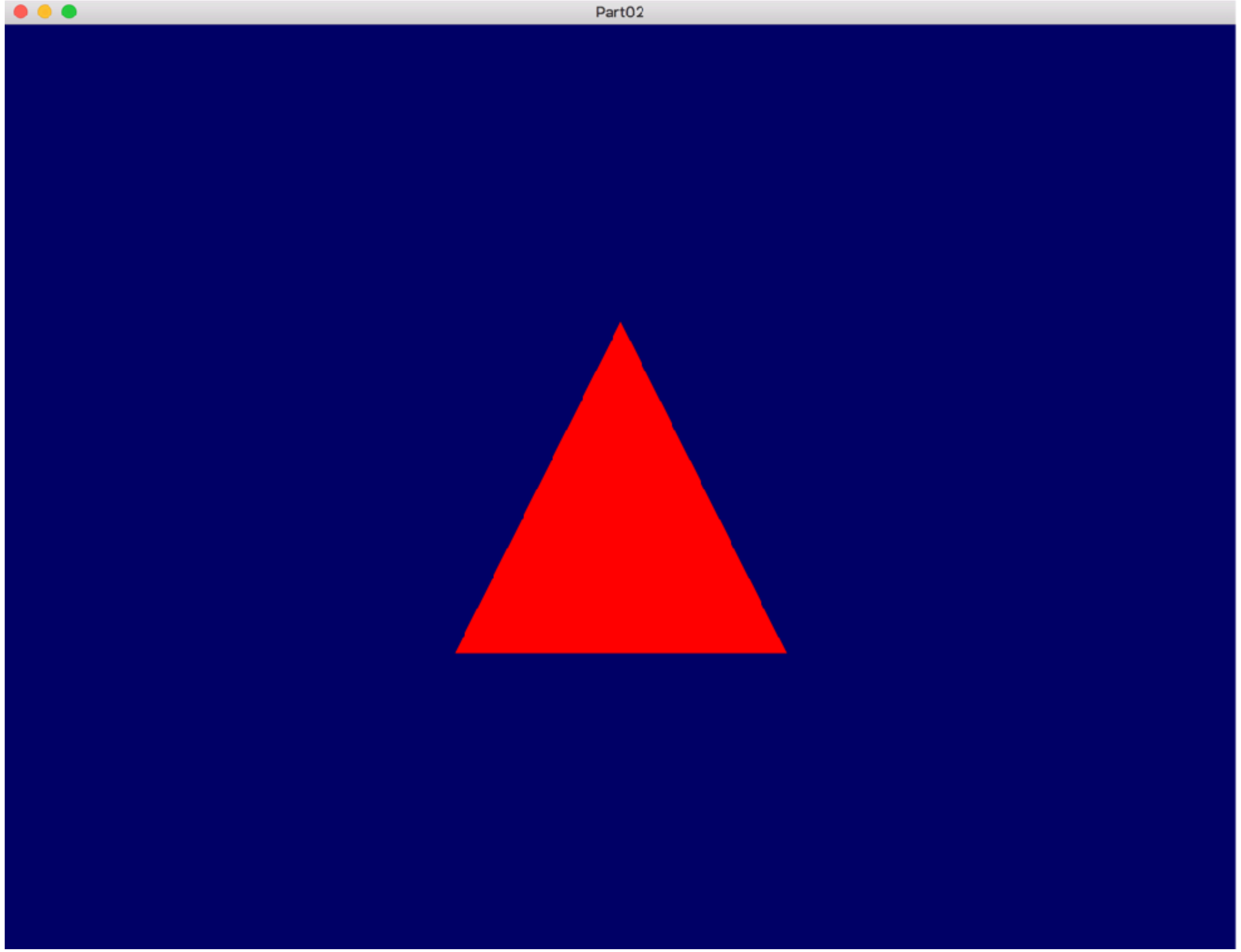


EXAMPLE: VERTEX BUFFER OBJECT

```
//  - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - -
 std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here
 // Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(glm::vec3), &m_vertices[0], GL_STATIC_DRAW);

//  - - - - - - - - - - - - 2.Step: Using the data for doing the rendering - - - - - - - - - - - - - - -
// 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBufferID);
    glVertexAttribPointer(
                          0,                    // attribute
                          3,                    // size
                          GL_FLOAT,             // type
                          GL_FALSE,             // normalized?
                          0,                    // stride
                          (void*)0              // array buffer offset
                          );
//  - - - - - - - - - - - - Actual drawing call - - - - - - - -
 glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

RESULT: VERTEX BUFFER OBJECT



SHADER

- Programs implementing the programmable parts of the pipeline
- Parts of a pipeline
  - Vertex Shader
  - Fragment Shader
  - Others (e.g. Geometry Shader, Tessellation Shader)
- Name originates from small programs used to calculate the shading of a surface

SHADER

Shading languages:

- **GLSL**
  - OpenGL Shading Language
  - C-like syntax
- HLSL
  - High-Level Shader Language
  - Developed by Microsoft for Direct3D
- CG
  - C for graphics
  - Developed by NVIDIA (not supported any more)

SHADER: STRUCTURE

```
#version 330 core

// Input data, different for all executions of this shader.
layout(location = 0) in vec3 some_input;
layout(location = 1) in vec2 some_other_input;

// Output data ;
out vec4 some_output;

// Values that stay constant for the whole mesh.
uniform vec4 someUniform;

void main(){

// run the computation
}
```

SHADER USAGE

1. Create Shader

```
GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);

GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
```

2. Load shader program into object

```
char const * shaderSource = someCodeString.c_str();
glShaderSource(VertexShaderID, 1, &shaderSource , NULL);
```

3. Compile Shader program

```
glCompileShader(VertexShaderID);
```

4. Use Shader program (bind)

```
glUseProgram(VertexShaderID);
```

PASSING PARAMETERS TO SHADER

For passing uniforms to shaders we need to create the location (glGetUniformLocation) and specify the value (glProgramUniformXX)

Examples:

- Add a colour value using a 4-dimensional vector:

```
// add color parameter to shader
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,1.0,1.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```

- Add a model-view-projection matrix using a 4x4 matrix:

```
glm::mat4 MVP;
GLint m_MVPID = glGetUniformLocation(programID, "MVP");
glUniformMatrix4fv(m_MVPID, 1, GL_FALSE, &MVP[0][0]);
```

VERTEX SHADER

Remember from the rasterisation pipeline:

- Handles the processing of individual vertices
- Input: vertex attributes (usually in model space)
- Output: vertex attributes (gl_Position is mandatory, usually in screen space)

Interface to fixed-function parts of the pipeline:

- in int gl_VertexID;
- out vec4 gl_Position;

VERTEX SHADER

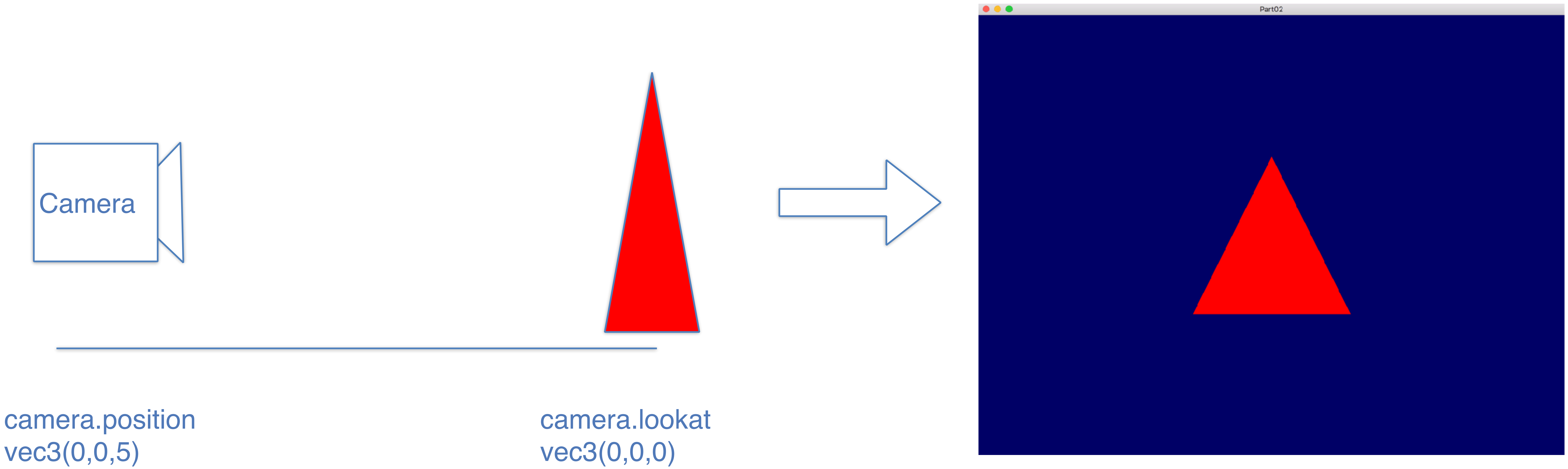
Simple vertex shader apply the model view project transformation:

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

// Values that stay constant for the whole mesh.
uniform mat4 ModelViewProjectionMatrix;
void main(){
    gl_Position =  ModelViewProjectionMatrix * vec4(vertexPosition_modelspace,1);
}
```

VERTEX SHADER



FRAGMENT SHADER

Remember from the rasterisation pipeline:

- Fragment shader processes each fragment
- Have control over the colour and depth values
- Input: interpolated vertex attributes
- Output: fragment color

Interface to fixed-function parts of the pipeline

- in vec4 gl_FragCoord;
- out float gl_FragDepth;

FRAGMENT SHADER

Simple Fragment Shader outputting specified colour:

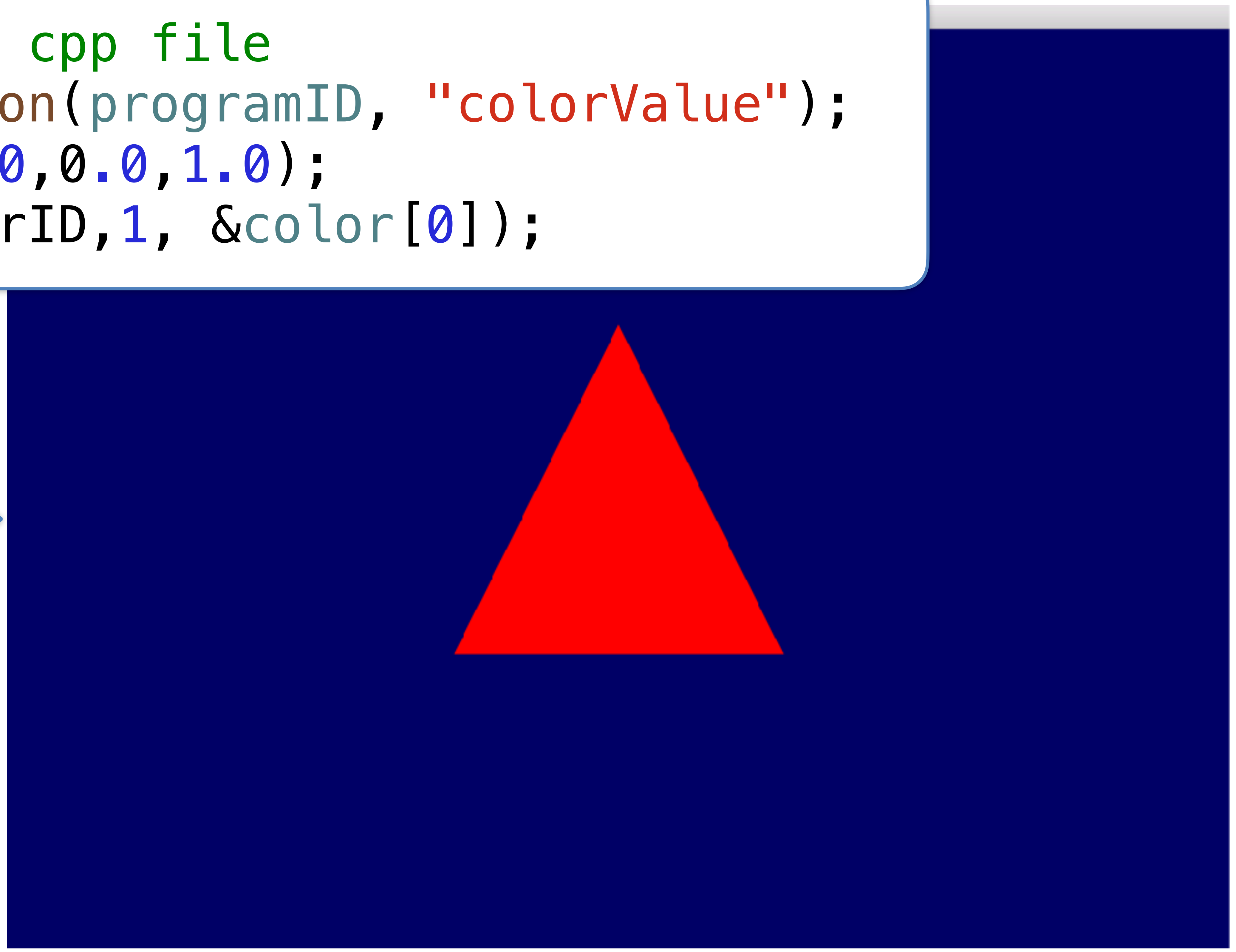
```
#version 330 core

// Ouput data
out vec3 color;

uniform vec4 colorValue;
void main()
{
  // Output color = red
  color = colorValue.rgb;

}
```

```
// add color parameter to shader - cpp file
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,0.0,0.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```



FRAGMENT SHADER

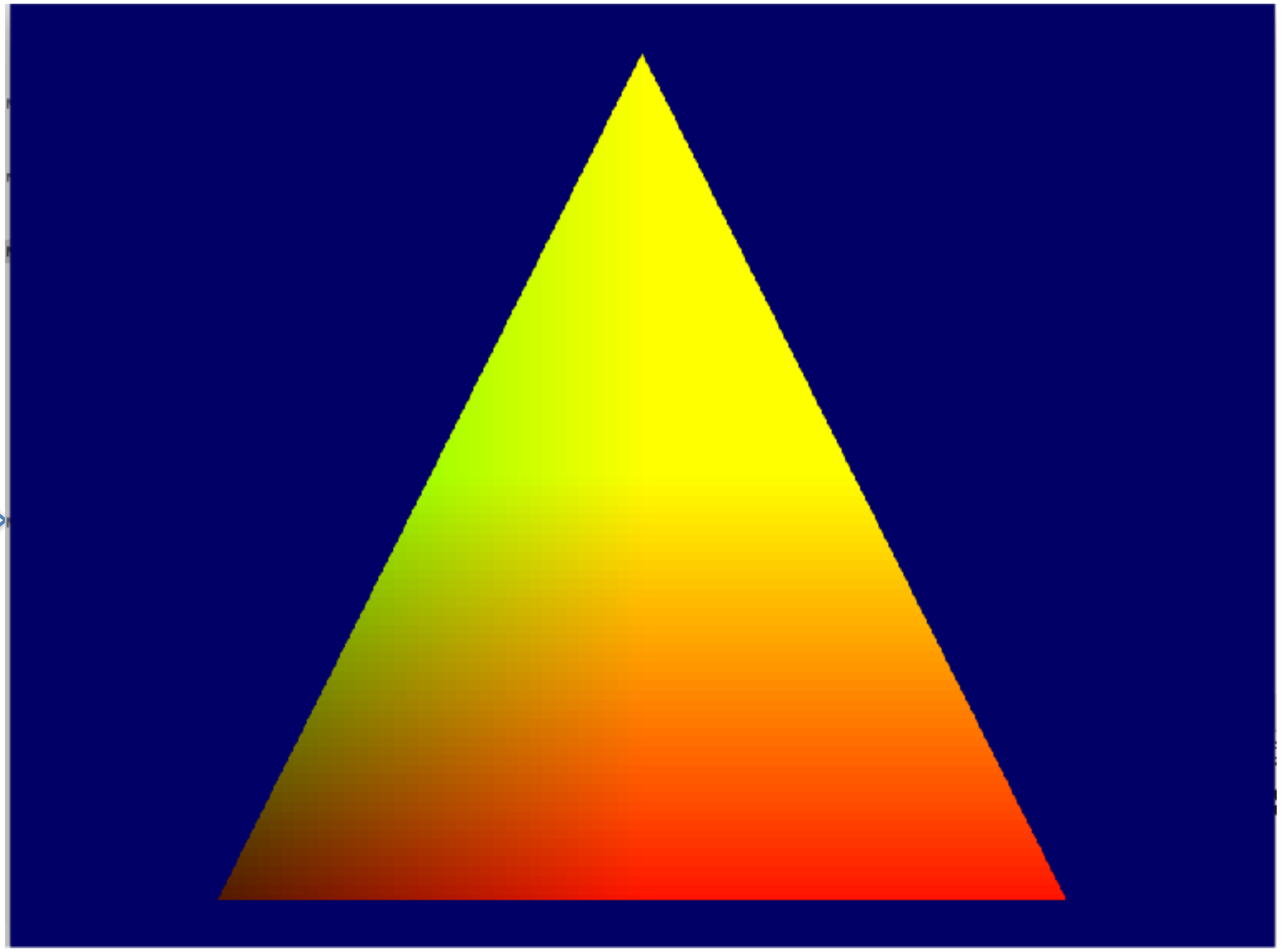
Simple Fragment Shader outputting gl_FragCoord:

```
#version 330 core

// Ouput data
out vec3 color;

void main()
{
  // Output color = screen coord
     color =  vec3(gl_FragCoord.r/1024,
     gl_FragCoord.g/768, 0.0);

}
```



VERTEX ARRAY OBJECT

- There are different ways how to configure vertex data:
  - Packed Arrays

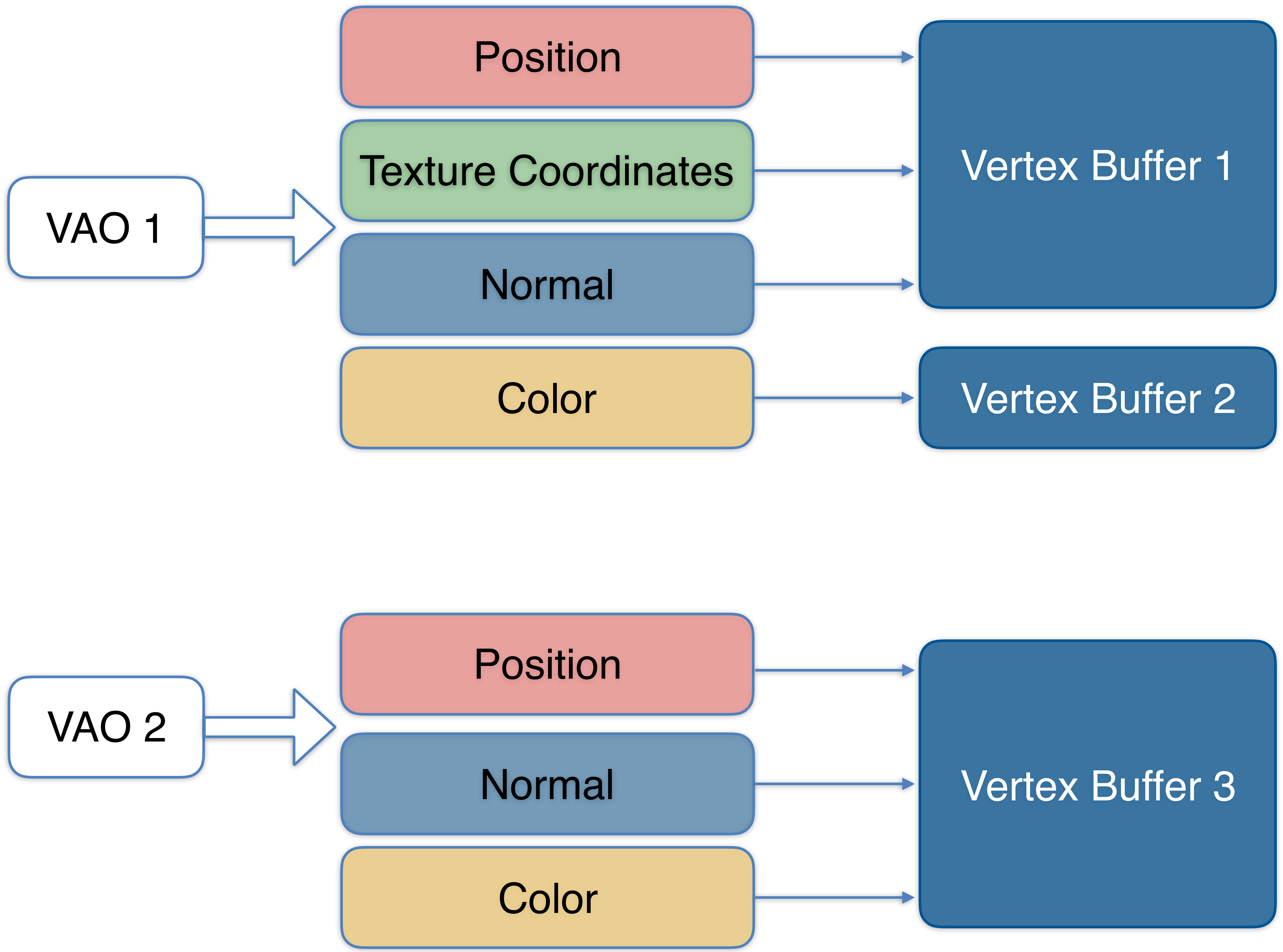
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
Vertex 0			Vertex 1			Vertex 2			Vertex 3		

R	G	B	R	G	B	R	G	B	R	G	B
Color 0			Color 1			Color 2			Color 3		

  - Interleaved Arrays

X	Y	Z	R	G	B	X	Y	Z	R	G	B
Vertex 0			Color 0			Vertex 1			Color 1		

VERTEX ARRAY OBJECT



VERTEX ARRAY OBJECT

- If an application renders a lot of different objects, we would need to reconfigure the buffers
- To avoid this -> Use vertex array objects
- **Vertex Array Object:**
  - Holds the state that configures the vertex specification stage:
    - Stores the data format of vertices
    - Buffer object bindings
    - Vertex attribute mapping

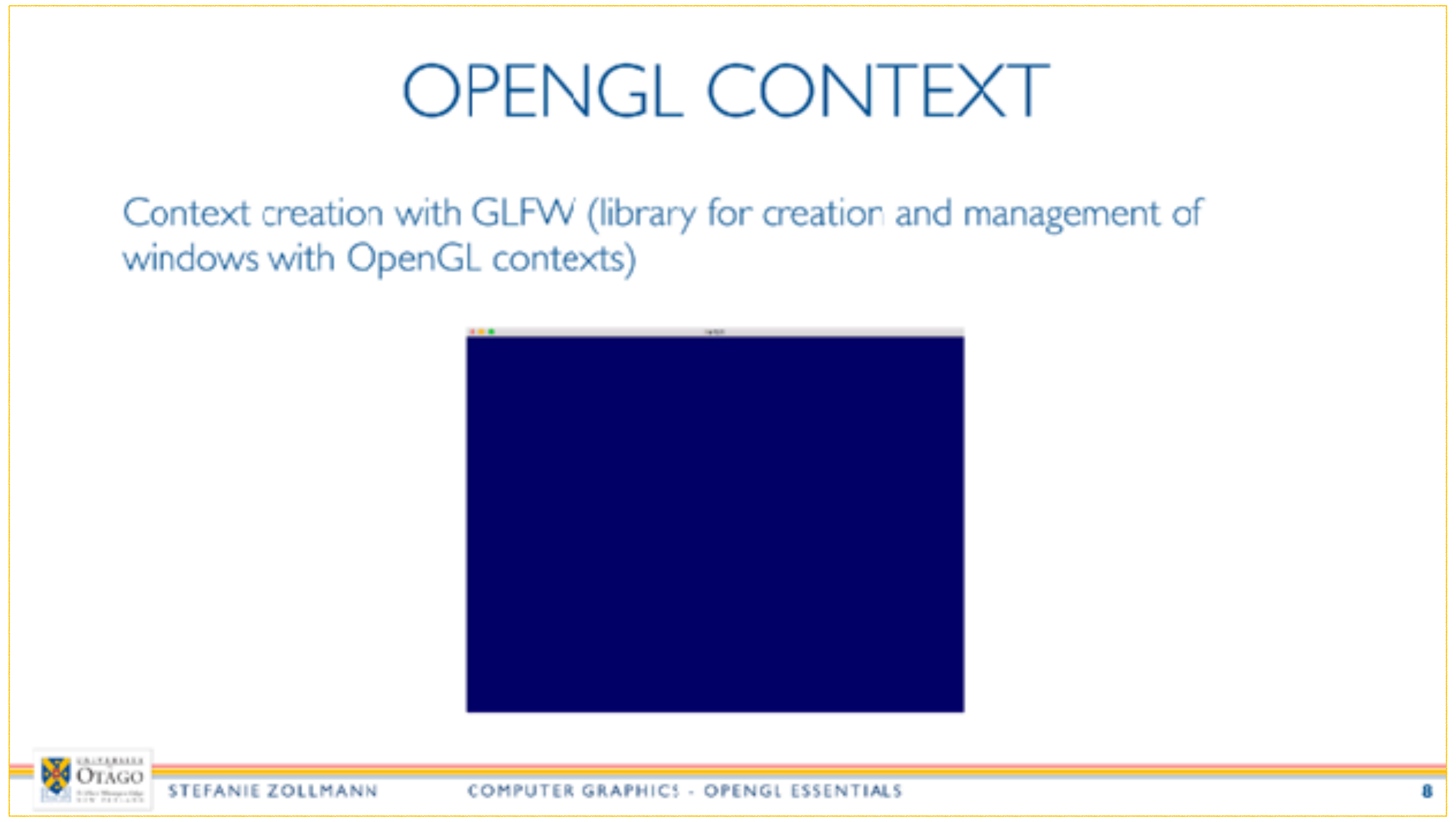
EXAMPLE: VERTEX ARRAY OBJECT

```
//create a Vertex Array Object and set it as the current one
 GLuint VertexArrayID;
 glGenVertexArrays(1, &VertexArrayID);
 glBindVertexArray(VertexArrayID);
```

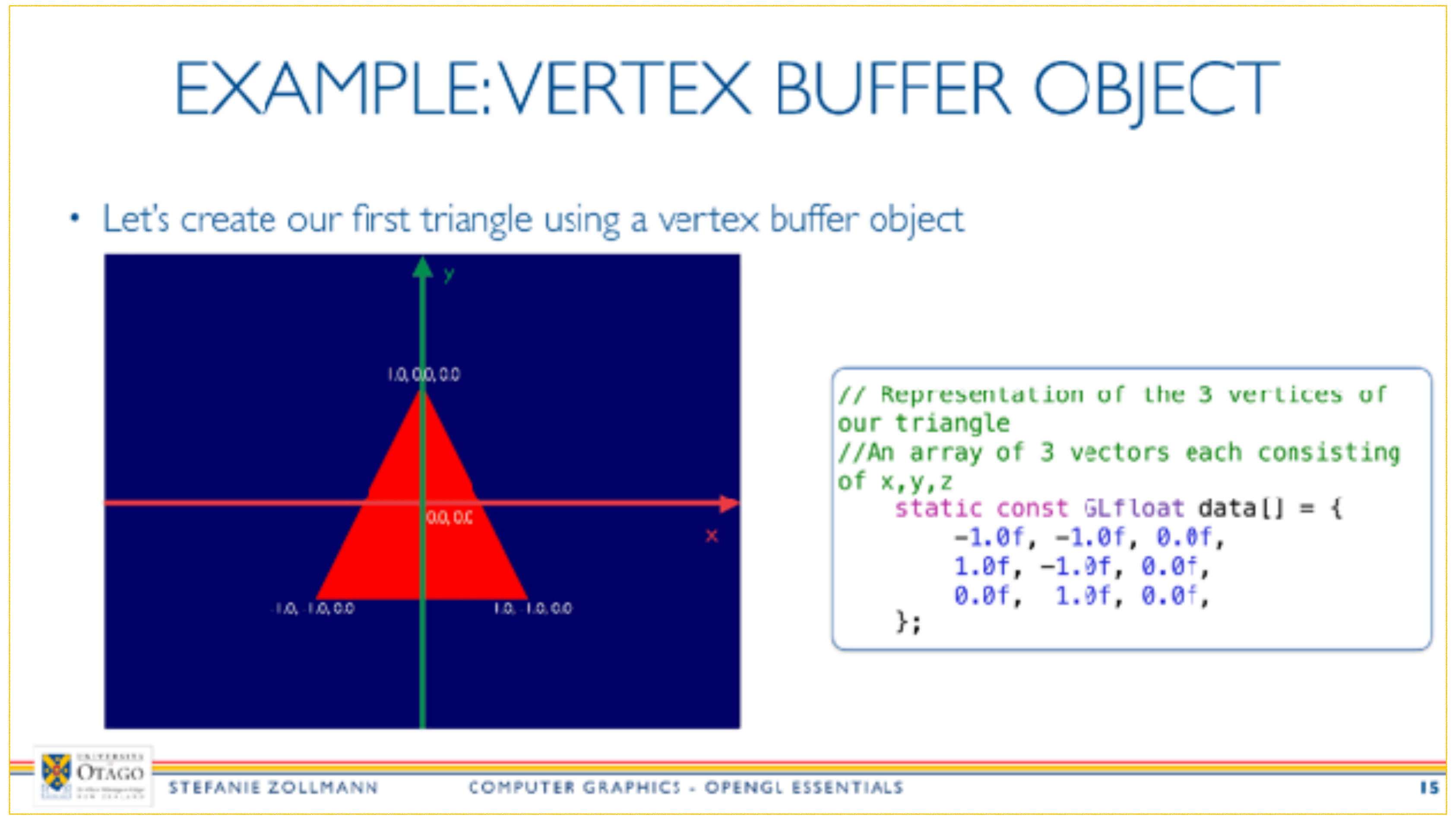
Important to note:

- We need at least on vertex array object in our application

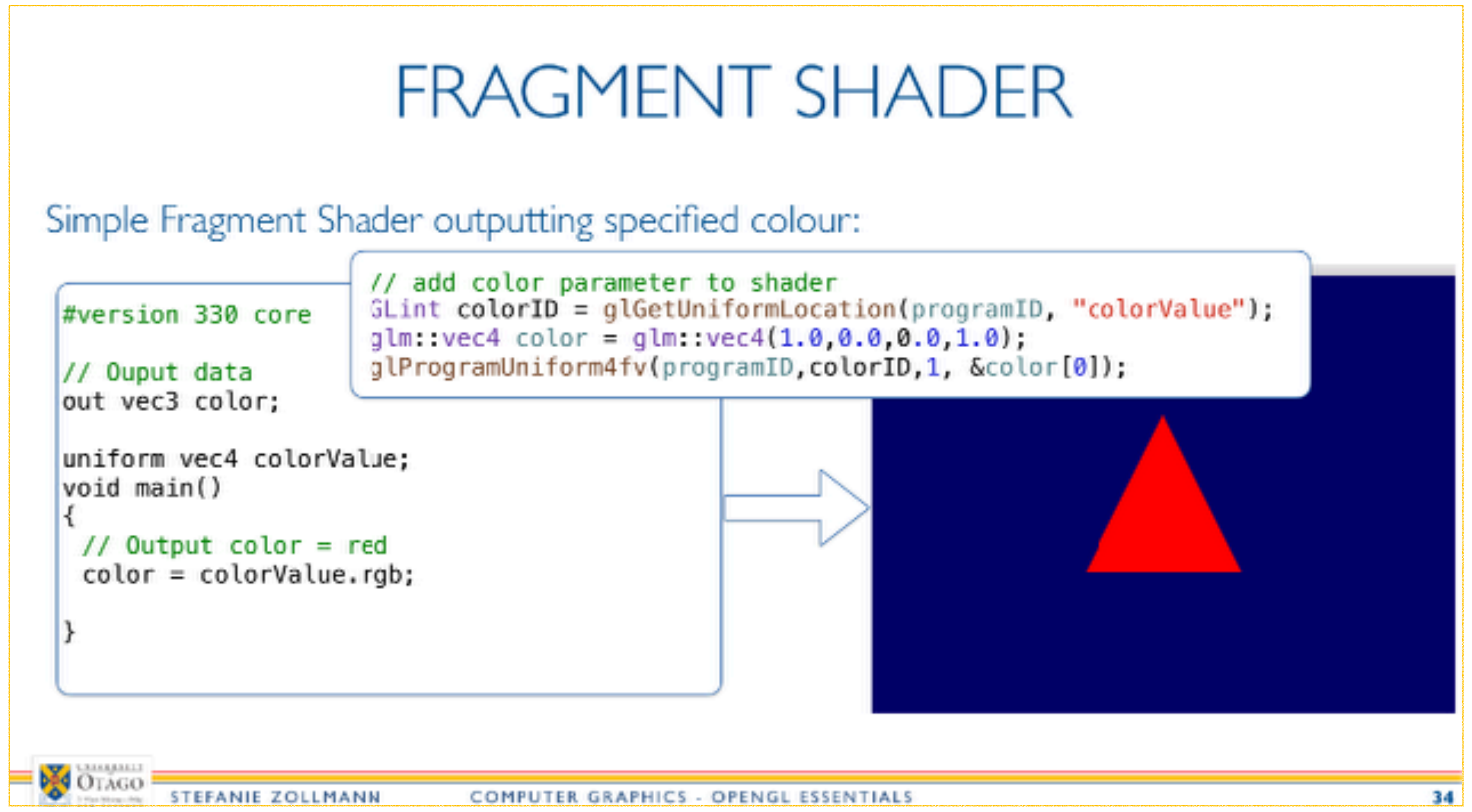
SUMMARY



OpenGL Context



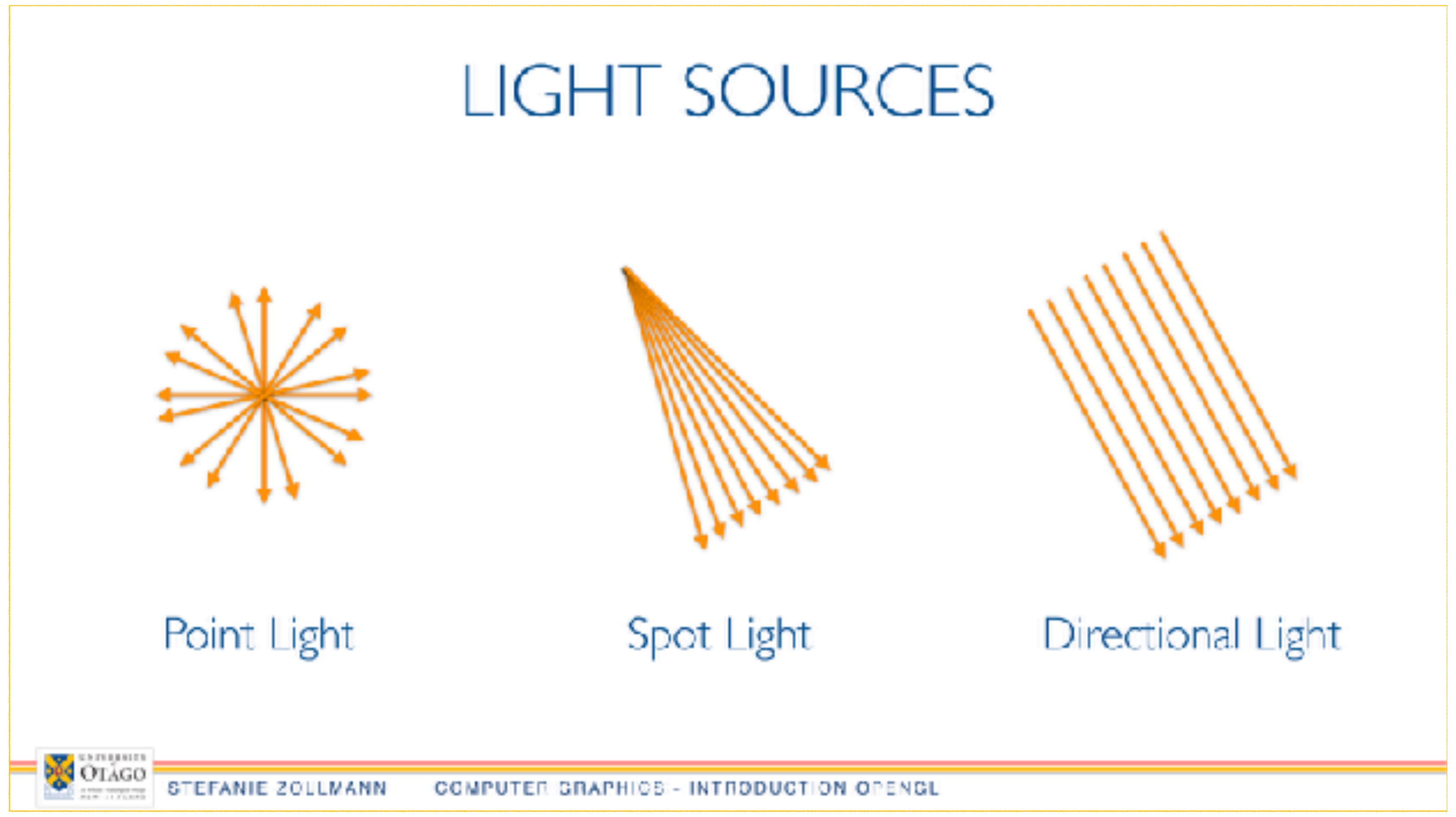
OpenGL Objects

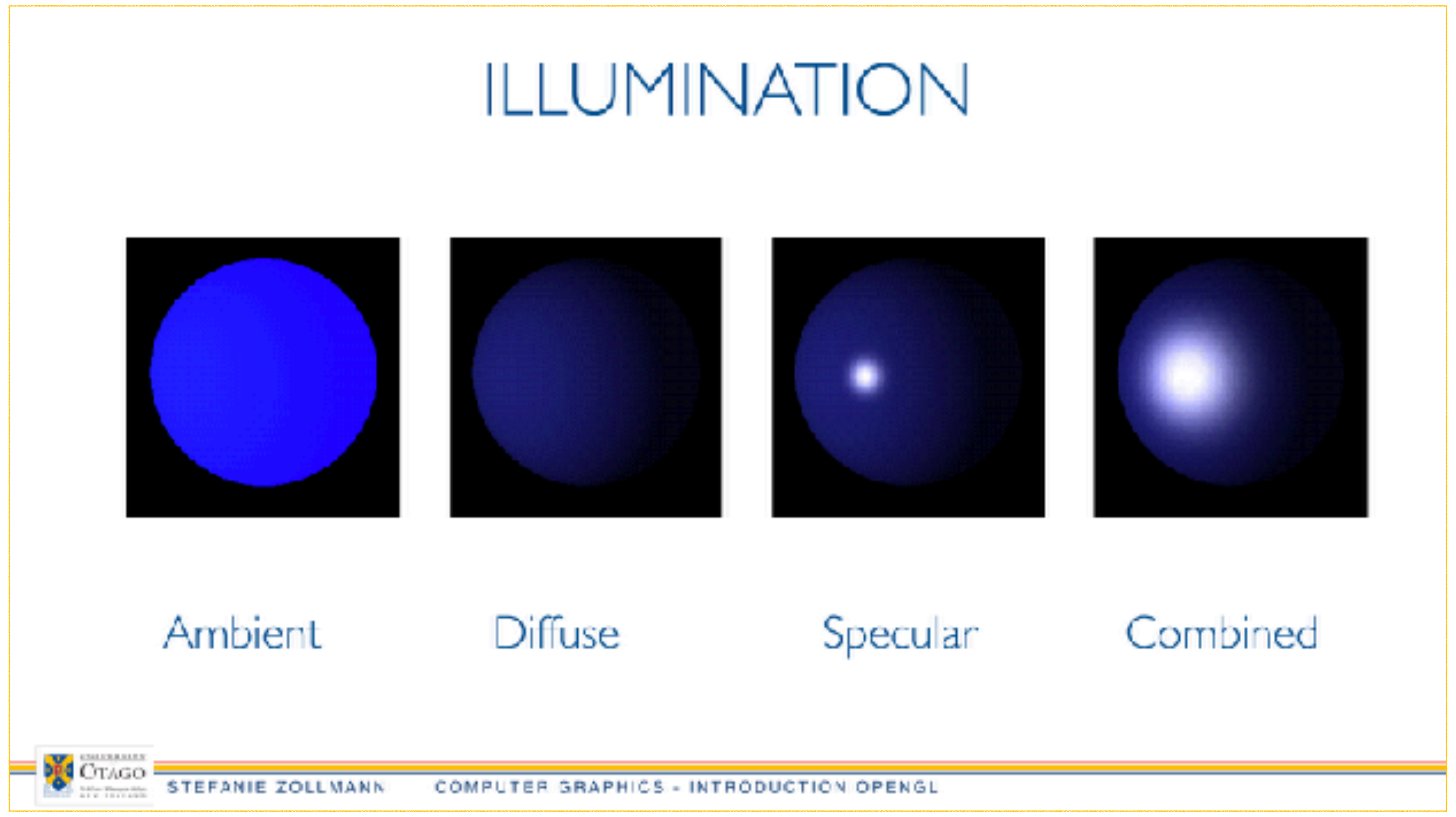


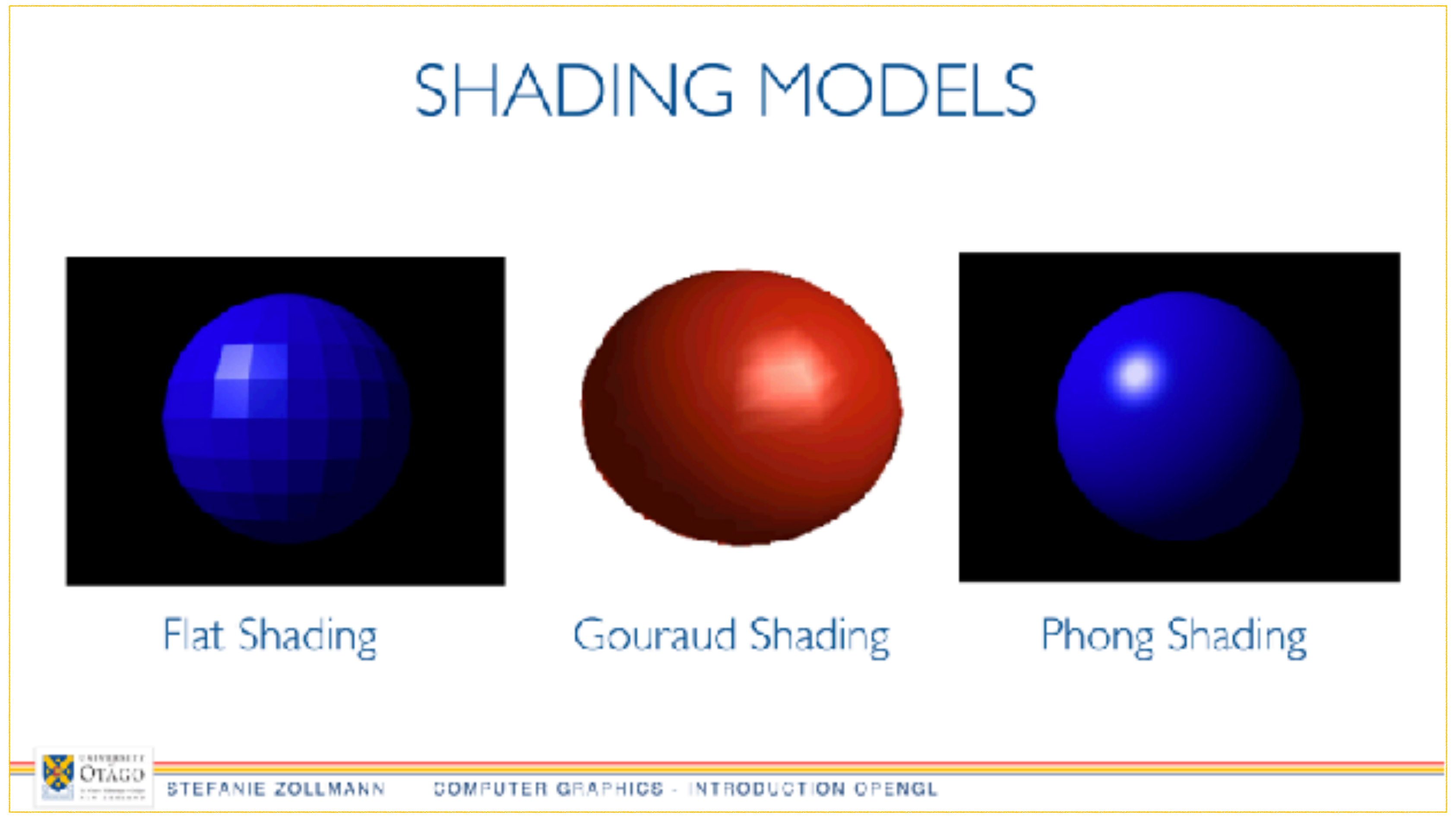
Shaders

WHAT'S NEXT

Lighting, illumination and shading models







Light Sources

Illumination

Shading

Thank You!

For more material visit

http://www.cs.otago.ac.nz/cosc342/