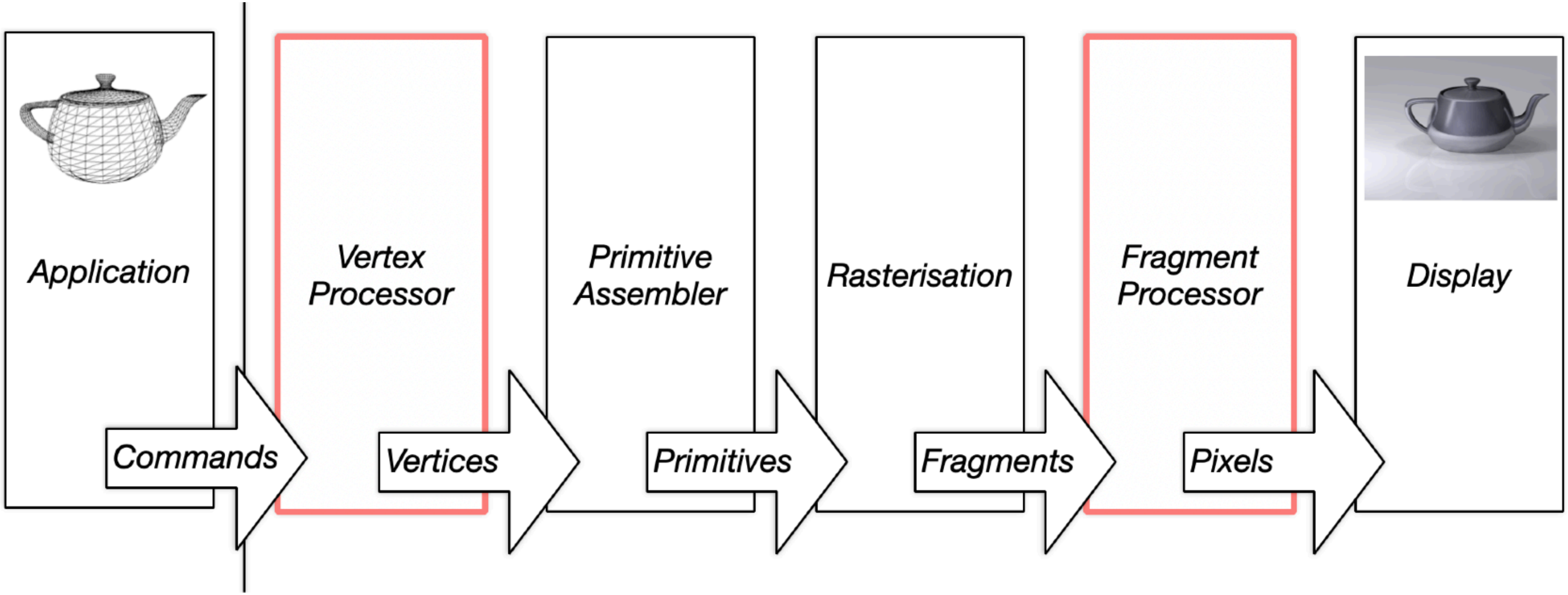
Visual Computing I:

Interactive Computer Graphics and Vision



OpenGL/Shader Programming

Stefanie Zollmann and Tobias Langlotz

Let’s Get Practical

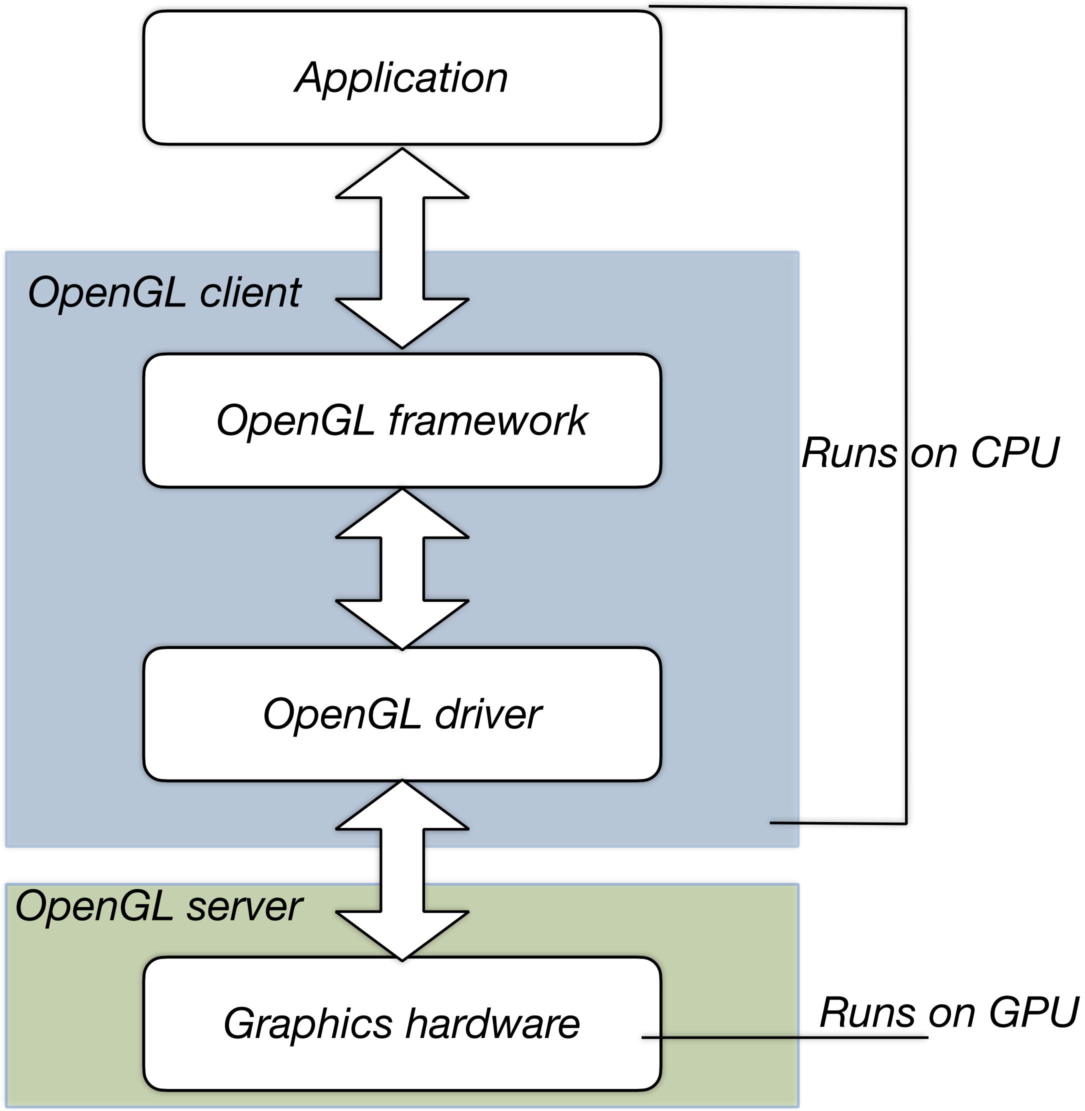
How to talk to the Graphics Hardware?

Graphics APIs

OpenGL®

OpenGL

- Cross-language, cross-platform application programming interface (API)
  - Interface is platform-independent
  - Implementation is platform-dependent.
- API for interacting with graphics processing unit (GPU) to render 2D and 3D graphics
- Works using a client-server model
  - Client (application) creates commands
  - Server processes commands



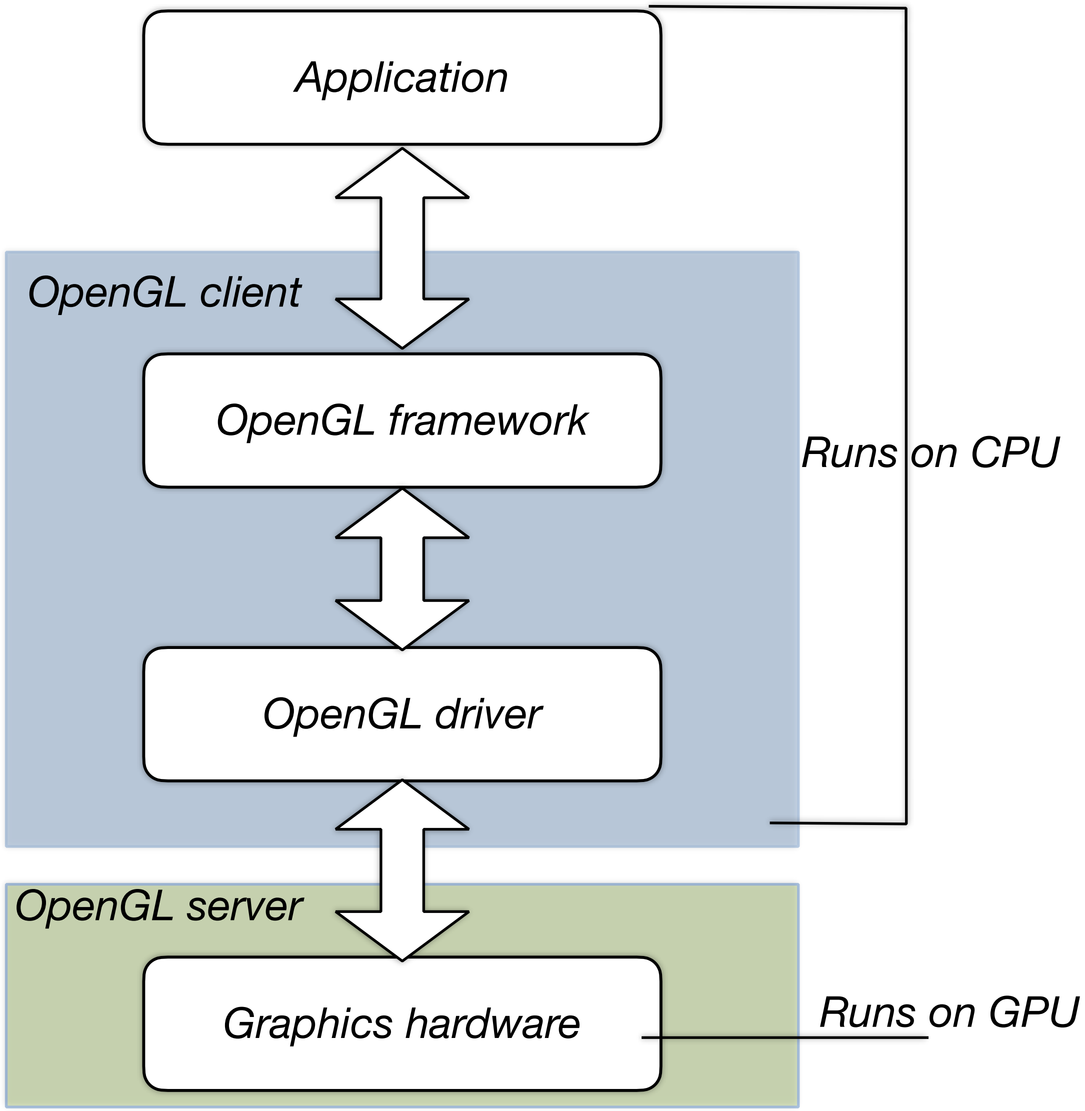
OpenGL

Important to note:

- The API is defined as a set of functions
  - Drawing commands

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

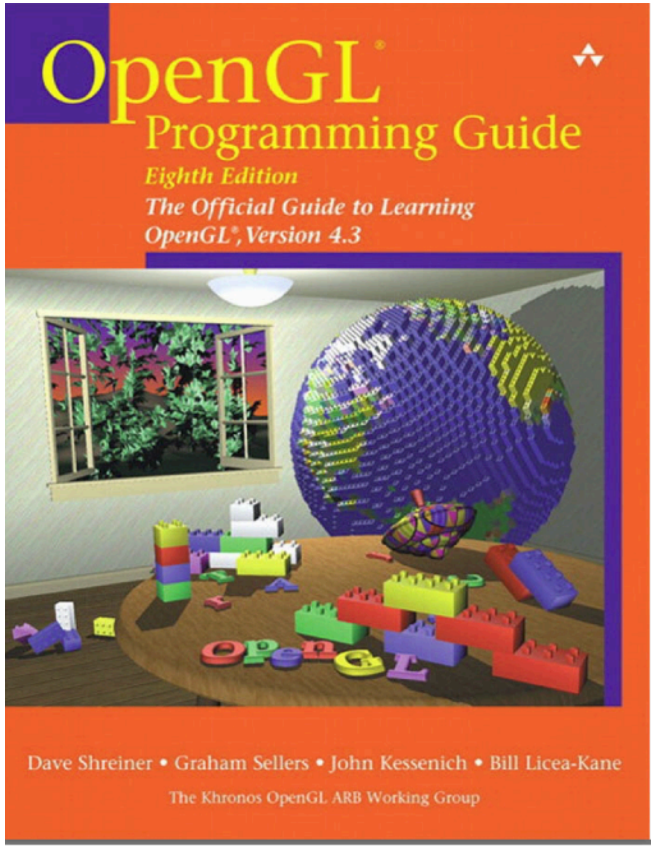
  - Working with identifier: no concept of permanent objects

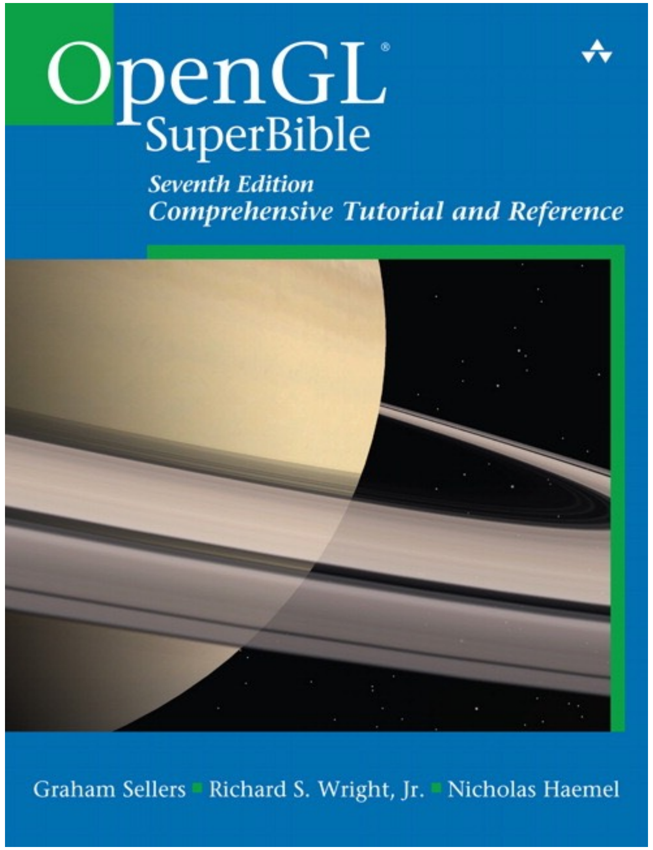


OpenGL - History

- 1992 – Originally released by Silicon Graphics Inc. (SGI) as a platform-independent graphics API for professional 3D applications
- 2006 – Management transferred to the Khronos Group, a non-profit industry consortium that also maintains Vulkan, WebGL, and OpenXR
- 2004 – OpenGL 2.0: Introduced the OpenGL Shading Language (GLSL), allowing programmable vertex and fragment processing
- 2008 – OpenGL 3.0: Major revision; deprecated the fixed-function pipeline and immediate mode (glBegin/glEnd) in favour of the programmable pipeline (using shaders and buffer objects)
- 2017 – OpenGL 4.6: The last version released by Khronos
- Now: OpenGL is stable but no longer actively developed. Khronos and GPU vendors continue to provide driver support, but new features and performance improvements are being developed under Vulkan, which is intended as its successor

OpenGL - Ressources





https://learnopengl.com/

OpenGL Concepts

- OpenGL Context:
  - OpenGL operates within a context, the environment that links the application to the GPU)
- OpenGL State:
  - Current configuration controlling rendering behaviour.
  - OpenGL functions modify or query this global state inside the active context.
  - OpenGL State is stored in context
- OpenGL Object Model:
  - Organizes GPU resources (buffers, textures, shaders))

—> Open Context holds the state and objects that define how and what OpenGL renders.

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

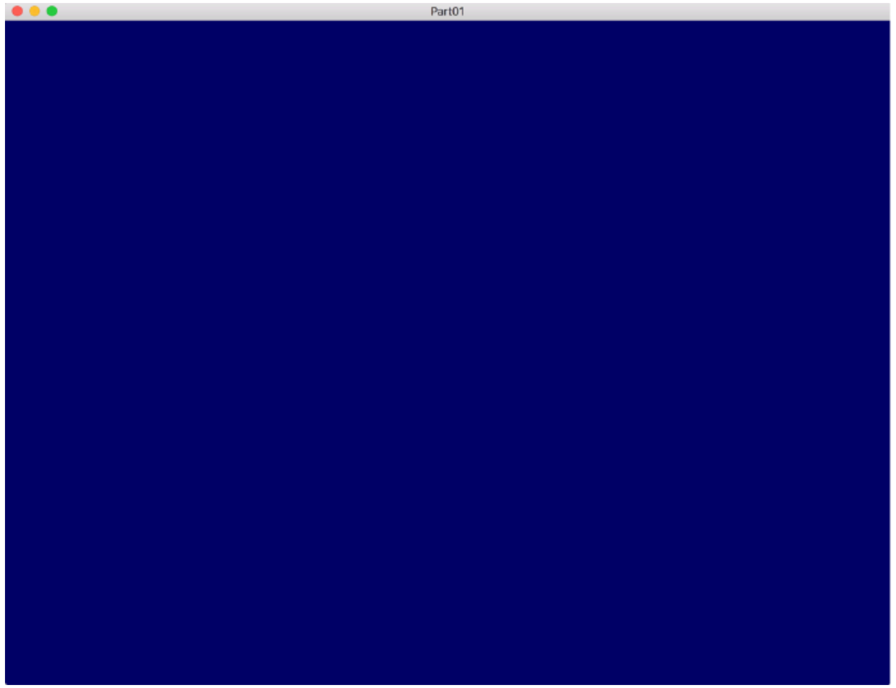
OpenGL Context

Context creation with GLFW (Graphics Library Framework: 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 (Graphics Library Framework: 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 blending
glEnable(GL_BLEND);

// Disable blending
glDisable(GL_BLEND);
```

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

```
//specifies textures
glTexImage2D(GL_TEXTURE_2D, 0,
GL_RGB, width, height, 0, GL_RGB,
        GL_UNSIGNED_BYTE, image);
```

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?

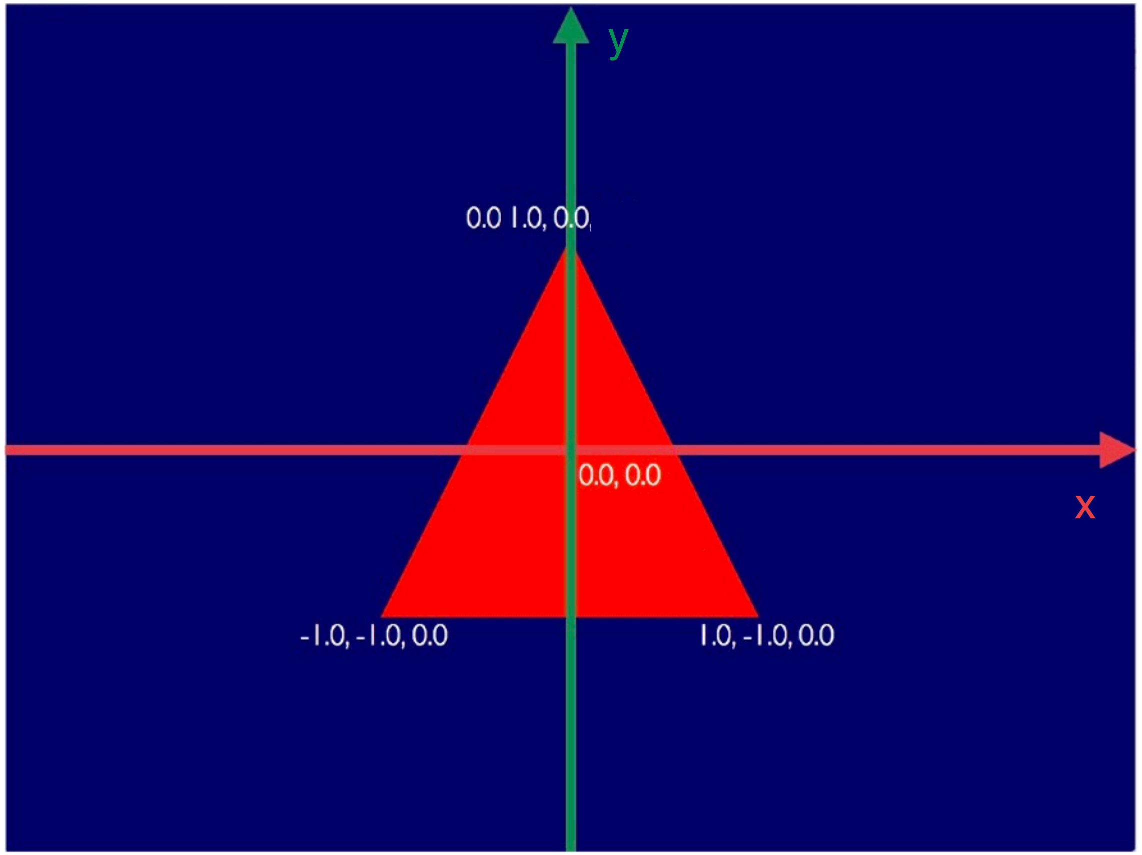
- target⇔type
- commands⇔methods

OpenGL Objects

- Act as
  - Sources of input
  - Sinks for output
- Examples:
  - Buffer 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
```

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)

  - 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		

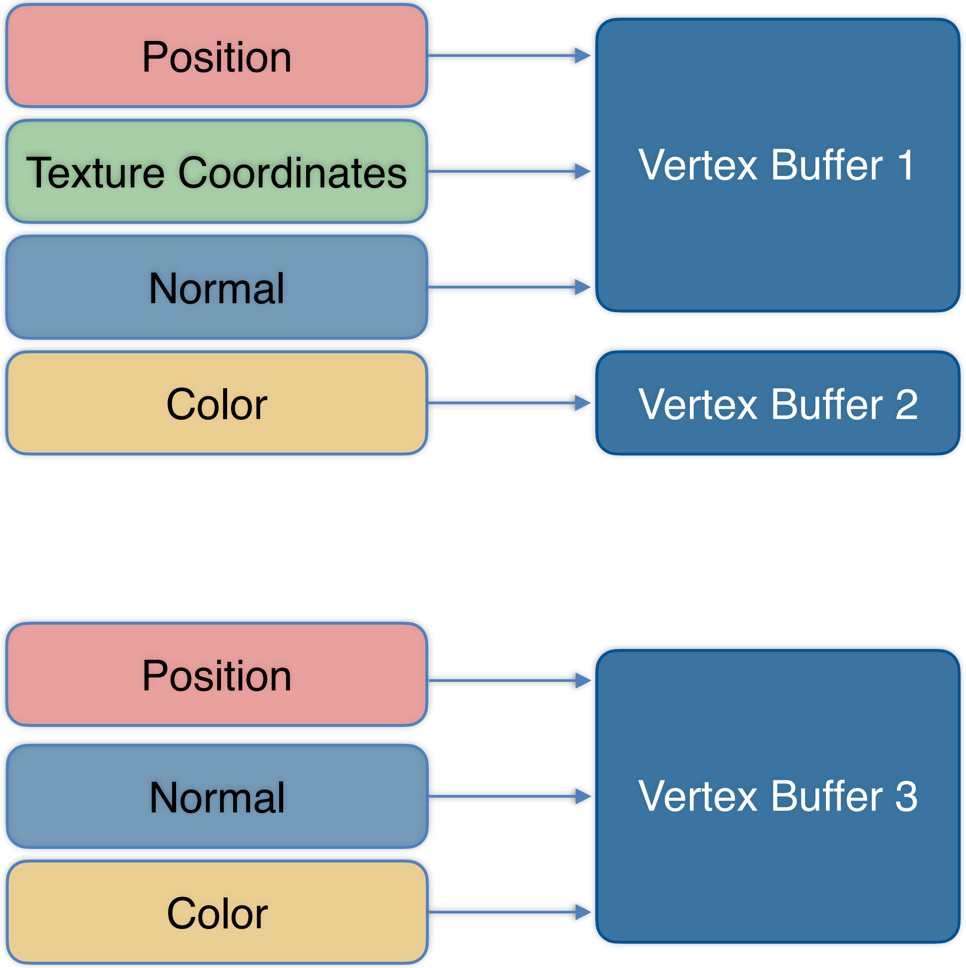
Stride = 3x4bytes + 3x4bytes = 24

Example: Interleaved Vertex Buffer Object

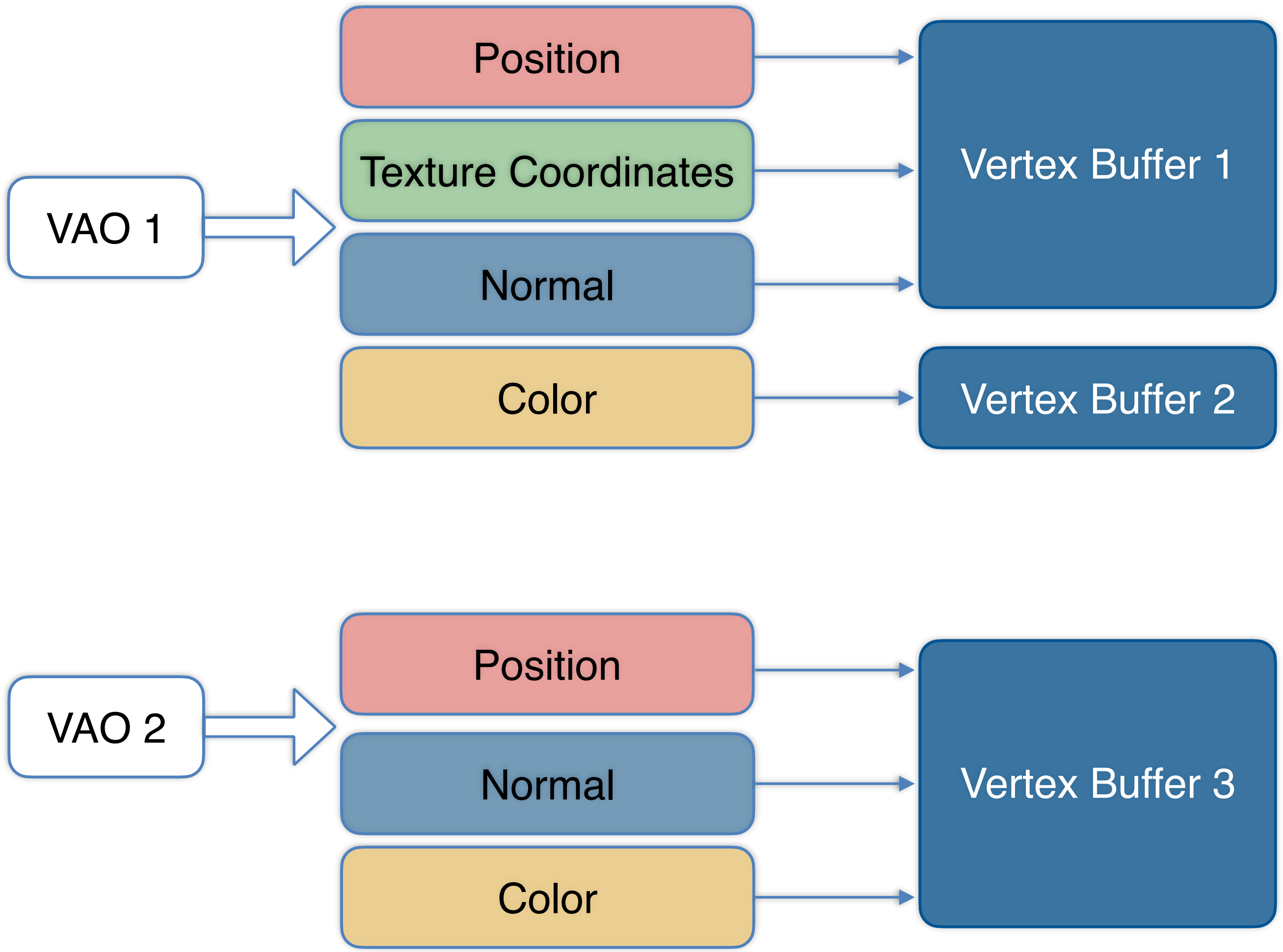
```
//  - - - - - - - - - - - - - - - 1. Step: Creating the data - - - - - - - - - - - - - - - - - - - - -
  std::vector<glm::vec3> m_vertices; // for more flexibility we use a std vector here - and fill it
// Load it into a VBO
    glGenBuffers(1, &m_vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, 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?
                      24,                   // interleaved data
                      (void*)0             // array buffer offset is 0 for vertices
                      );
//  - - - - - - - - - - - - - - - Actual drawing call - - - - - - - - -
  glDrawArrays(GL_TRIANGLES, 0, m_vertices.size()); //draw x triangles
```

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



VERTEX ARRAY OBJECT



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

VERTEX ARRAY OBJECT Example

```
GLuint vao2;
glGenVertexArrays(1, &vao2);
glBindVertexArray(vao2);

// Create and bind buffer object for vertex data
GLuint vbuffer;
glGenBuffers(1, &vbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vbuffer);

// copy data into the vertex buffer object
glBufferData(GL_ARRAY_BUFFER, NUM_VERTS * sizeof(Vertex), vertexdata, GL_STATIC_DRAW);


// set up vertex attributes
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, position));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, normal));
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, colour));

glBindVertexArray(vao2);
```

Draw Call

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

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

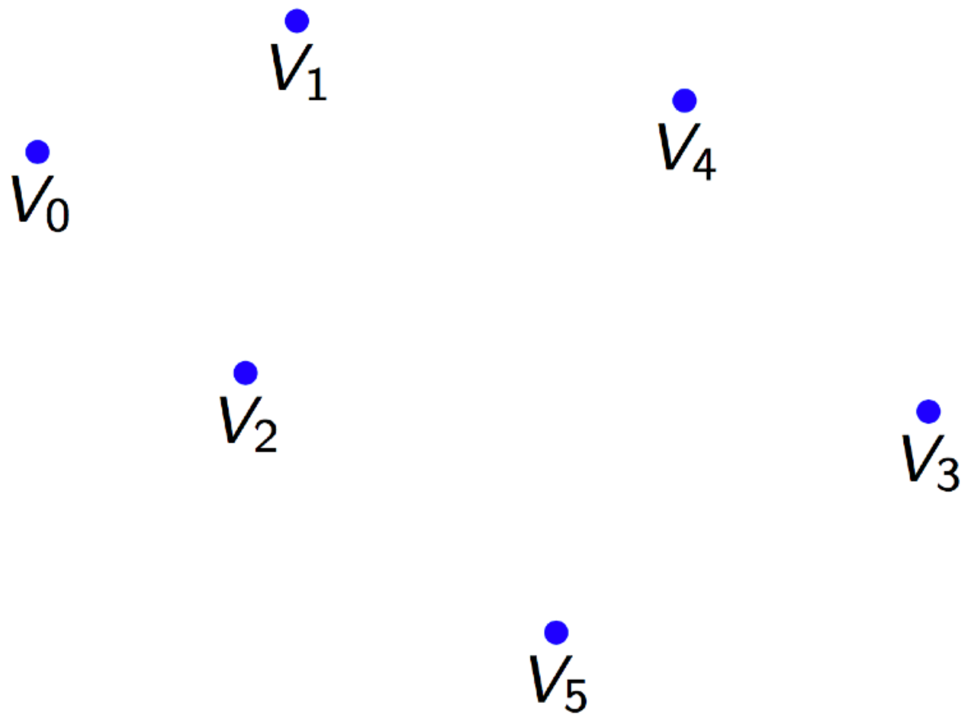
```
mode specifies what kind of primitives to render. e.g. GL_TRIANGLES
```

```
specifies the starting index in the enabled arrays.
```

```
count specifies the number of indices to be rendered.
```

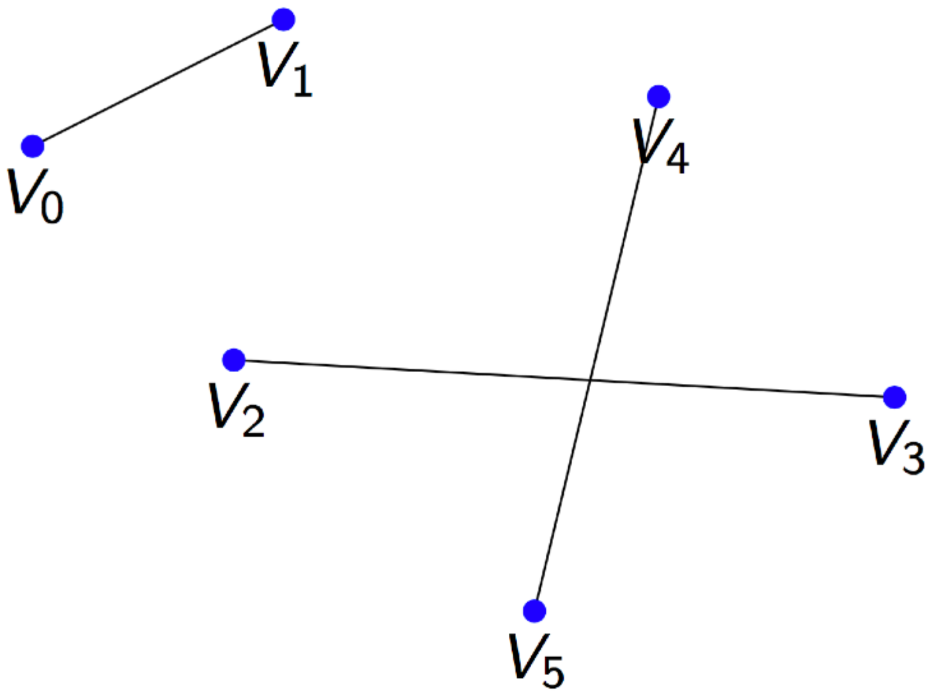
Primitive Types

- Points
- Lines
- Triangles
- Line strips
- Line loops
- Triangle strips
- Triangle fans



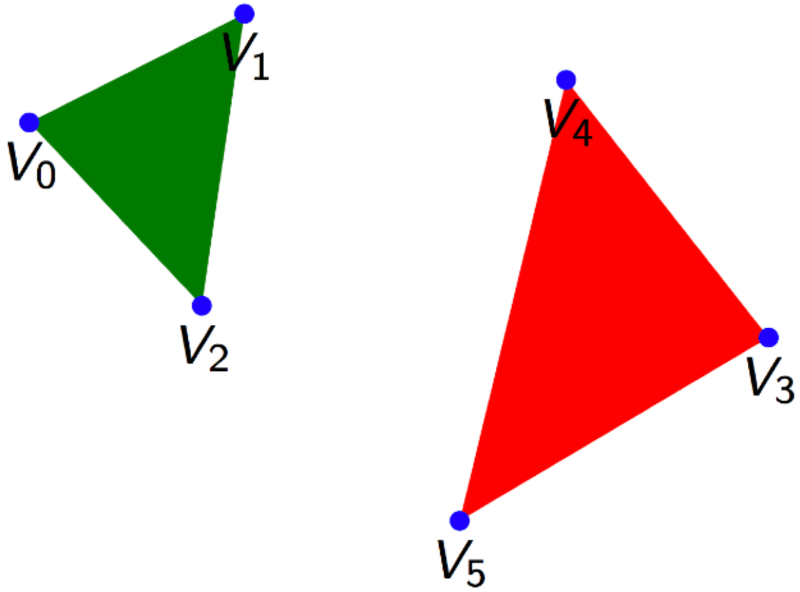
Primitive Types

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



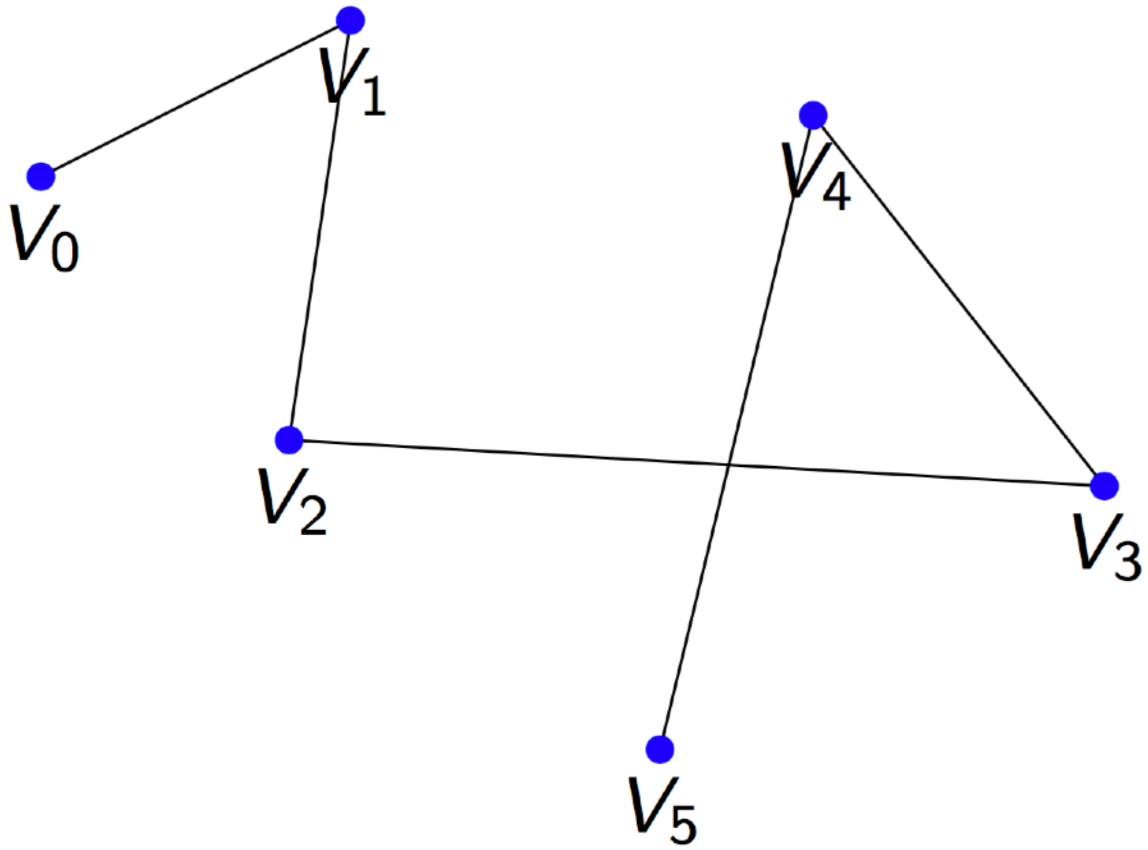
Primitive Types

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



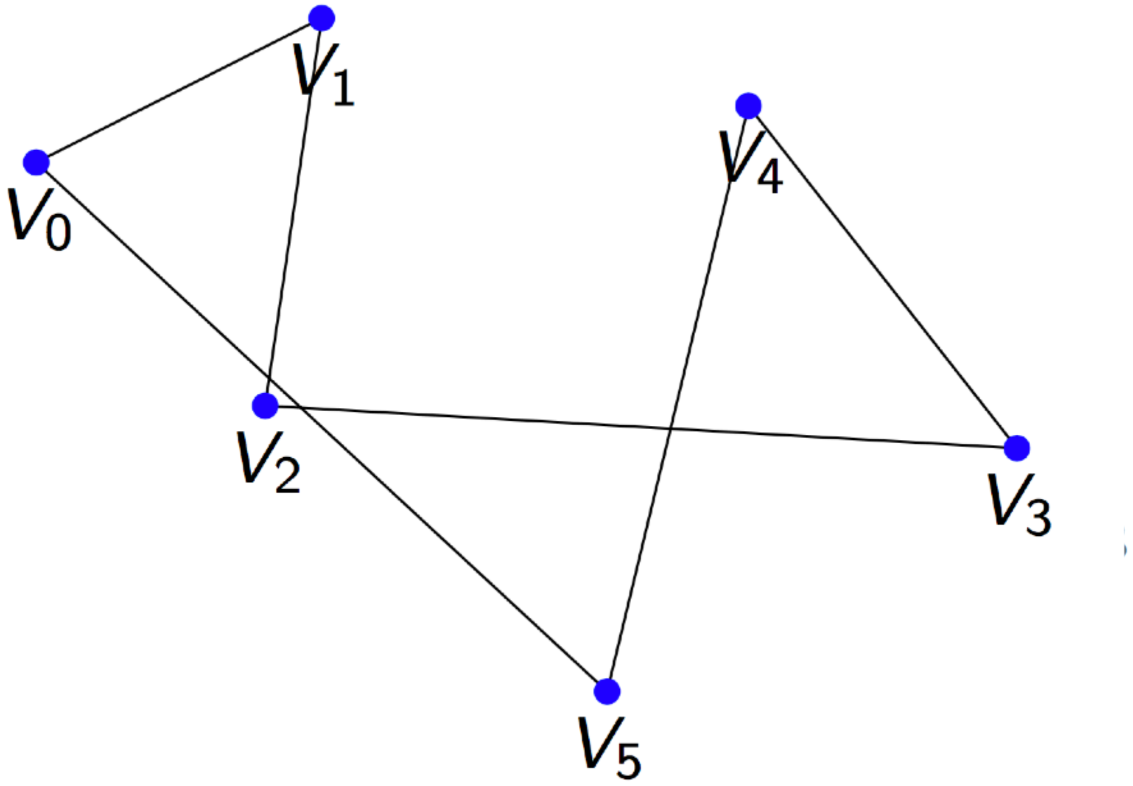
Primitive Types

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



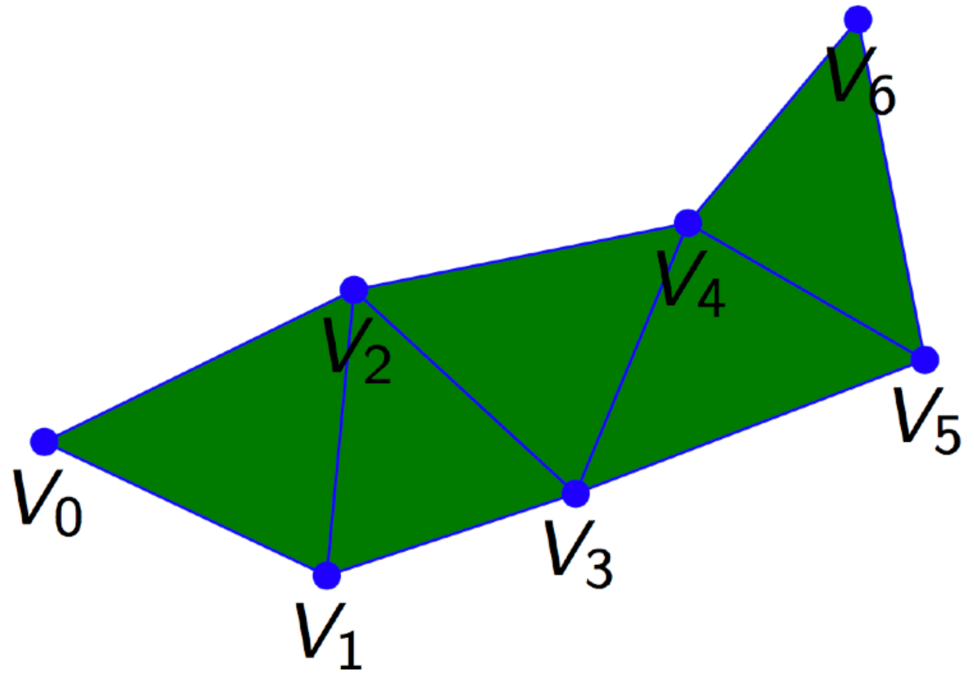
Primitive Types

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



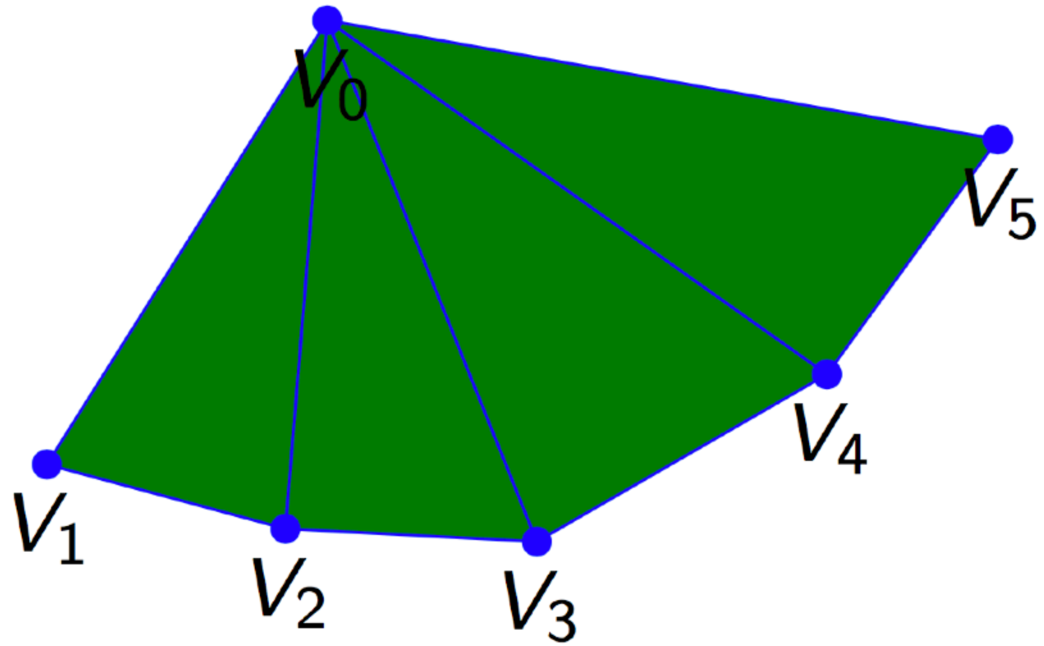
Primitive Types

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



Primitive Types

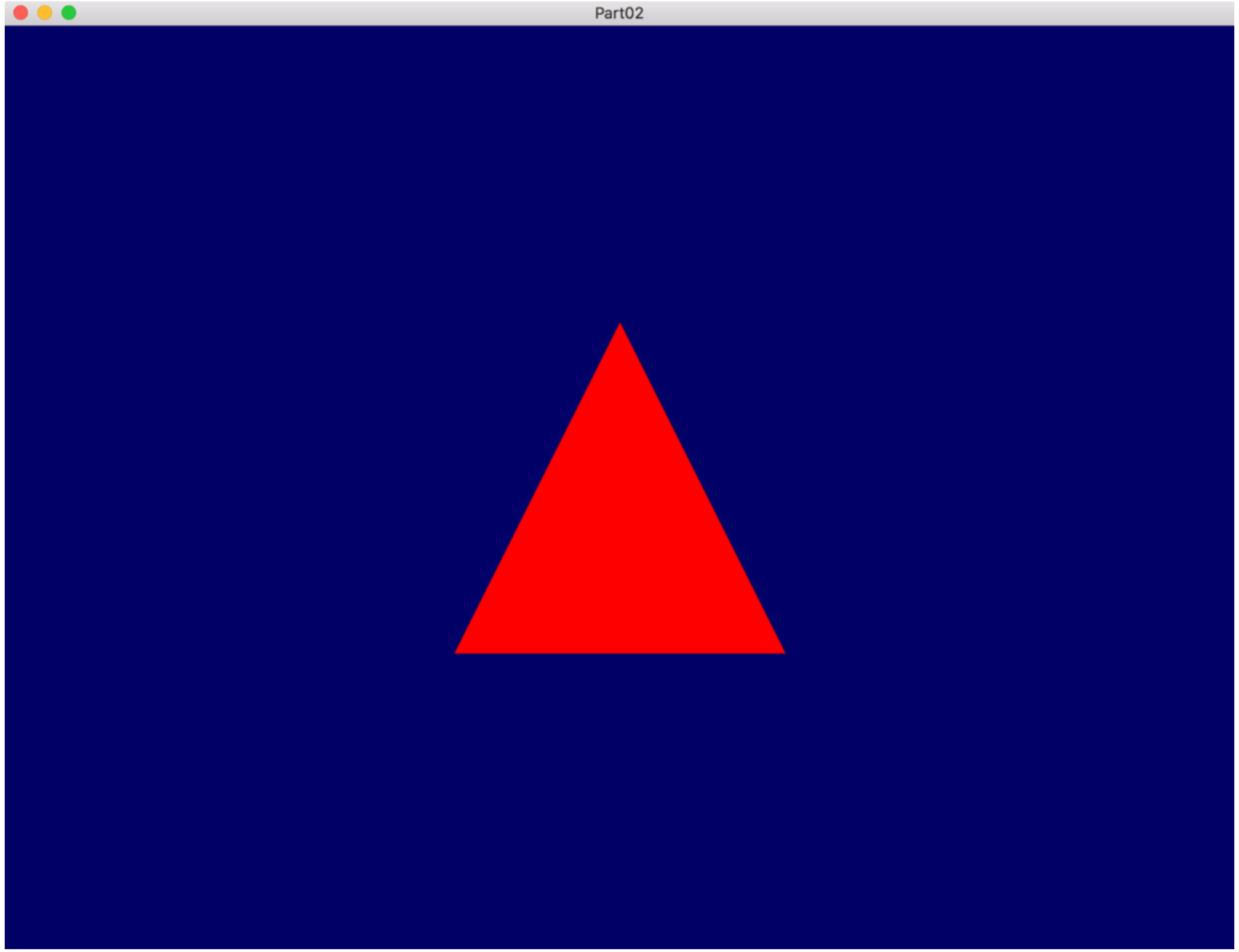
- Points
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans



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



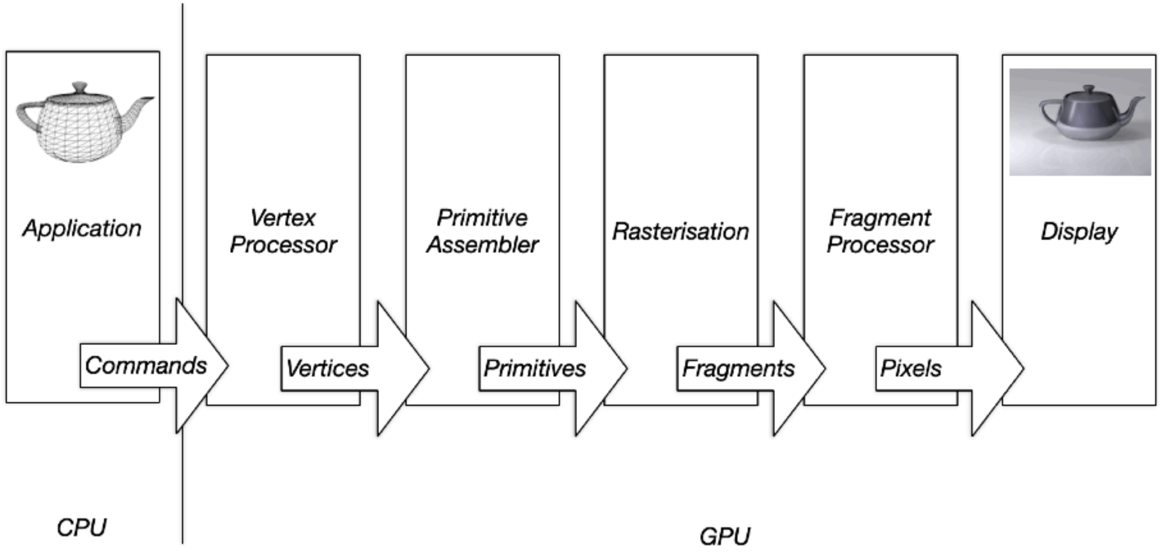
First Triangle

```
//  - - - - - - - - - - - - - 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
```

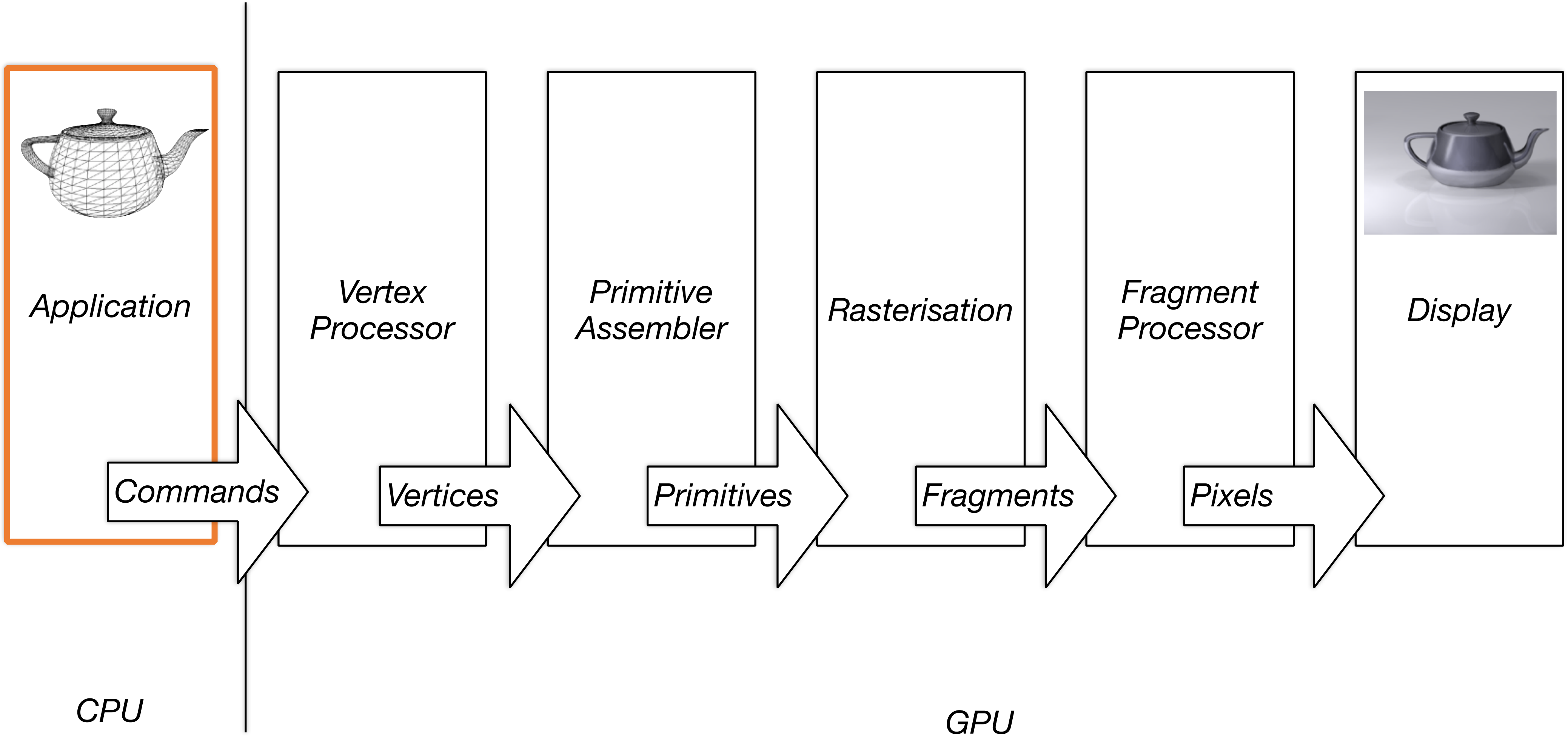
Example in Triangle.cpp

3-Min Discussion:

So Far - Where are we in the graphics pipeline?



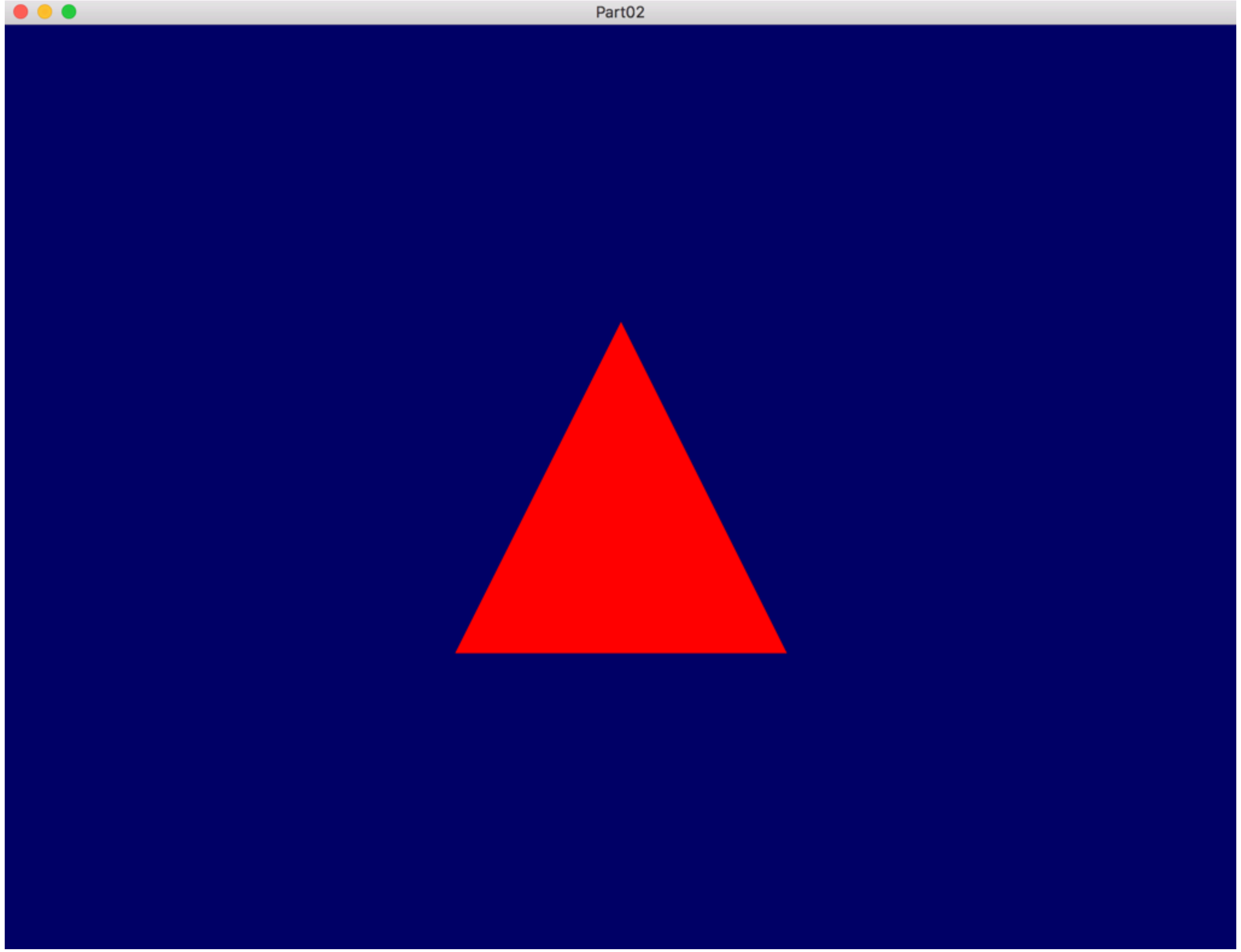
So Far - Where are we?



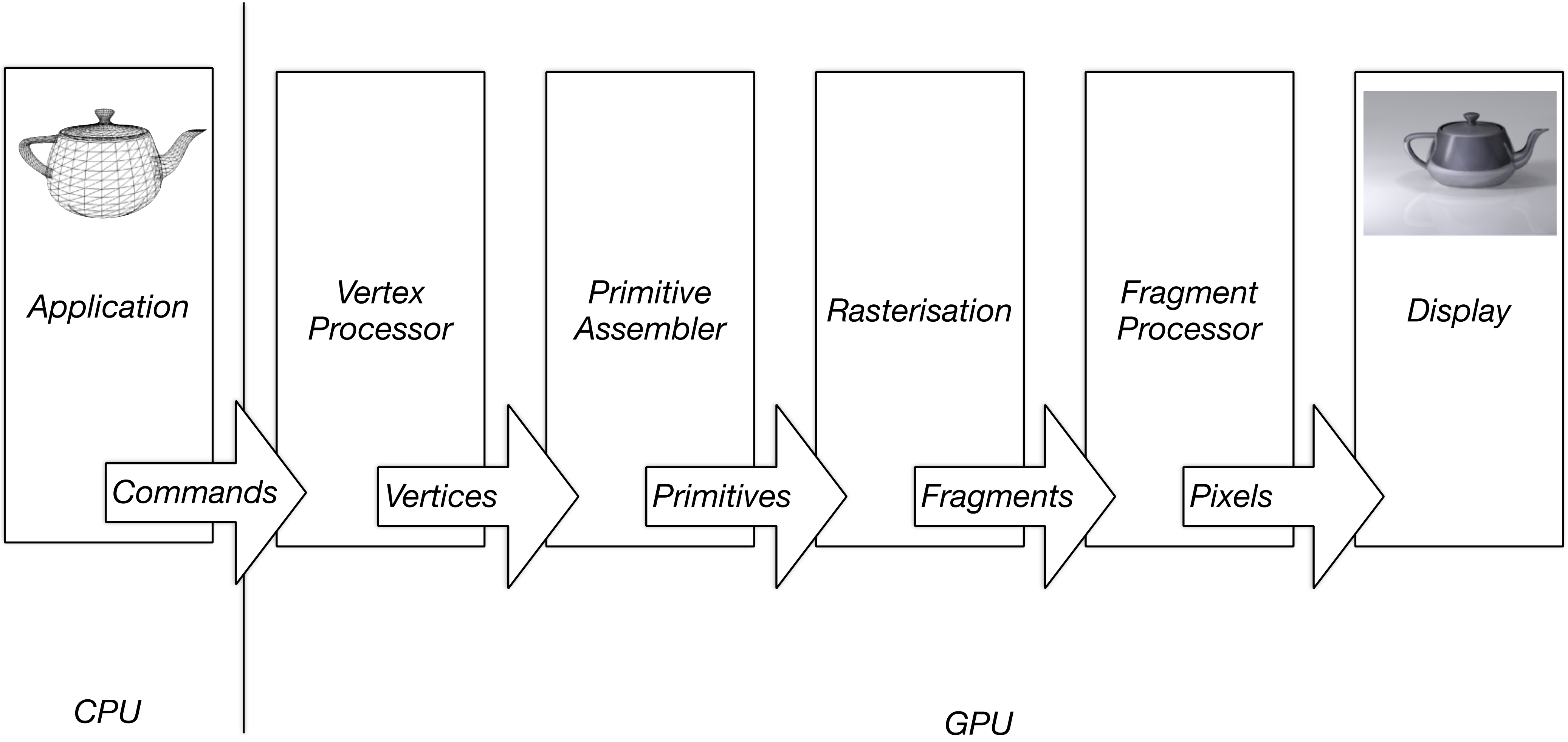
Shader

- 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

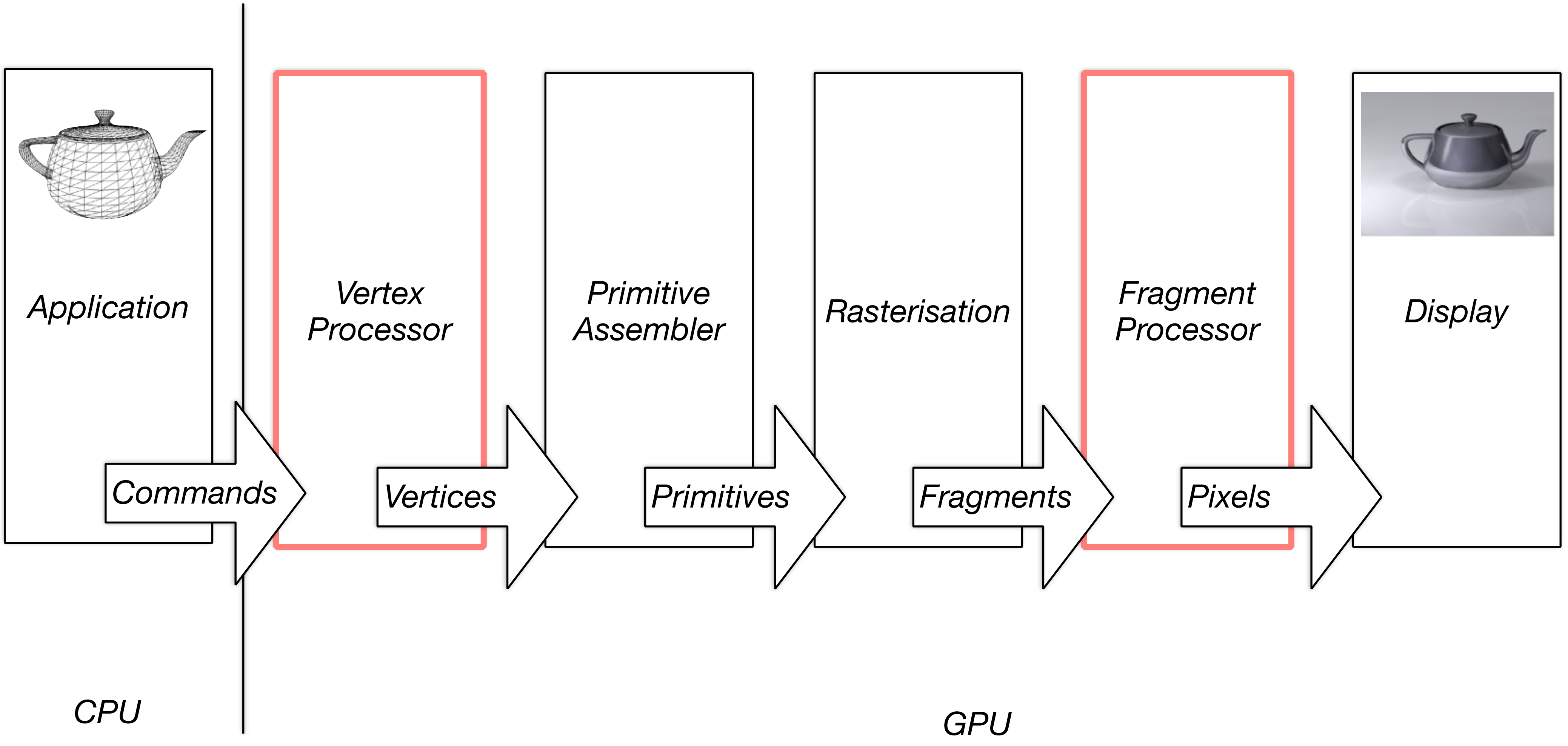
First Triangle



So Far - Where are we?



So Far - Where are we?



Shader

- Programs implementing the programmable parts of the pipeline
- Parts of 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

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)

Example:

- 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]);
```

Note: Example uses OpenGL Mathematics library (glm) - Documentation https://glm.g-truc.net/

Example in Shader.cpp

Passing parameters to Shader

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

Example:

- 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]);
```

Example code for ColourShader.cpp

Vertex Shader

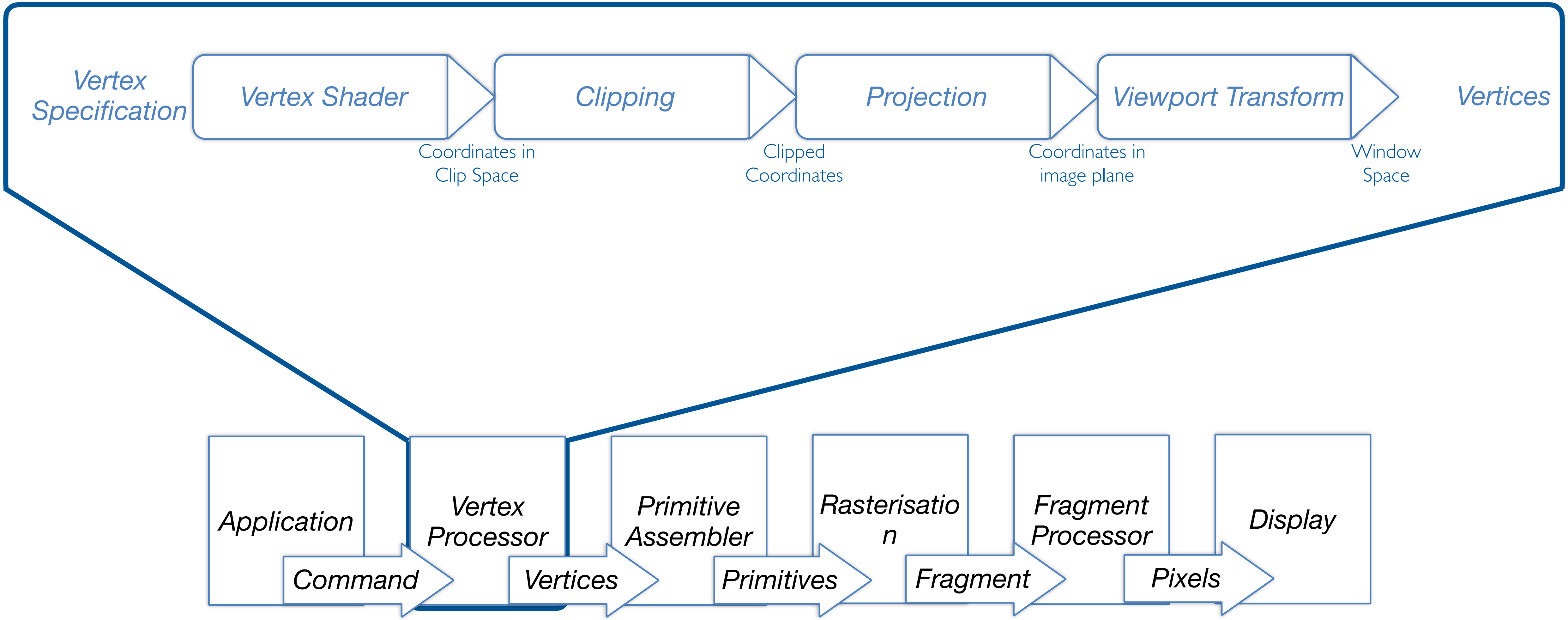
Remember from the rasterisation pipeline:

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

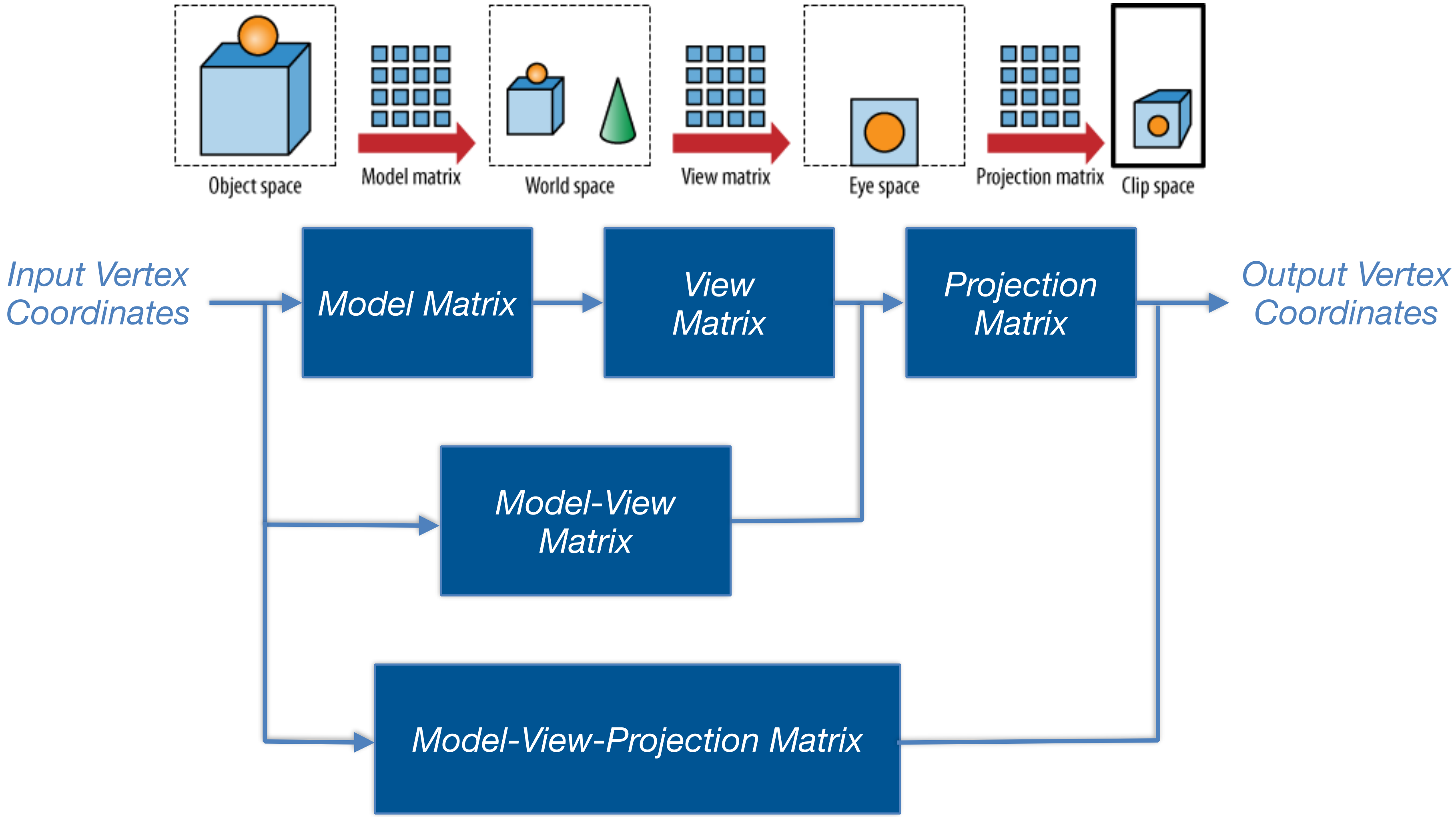
Interface to fixed-function parts of the pipeline:

- in int gl_VertexID;
- out vec4 gl_Position;

Vertex Processing



Vertex Shader: Transformations



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
}
```

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);
}
```

Example in basicShader.vert

Vertex Shader

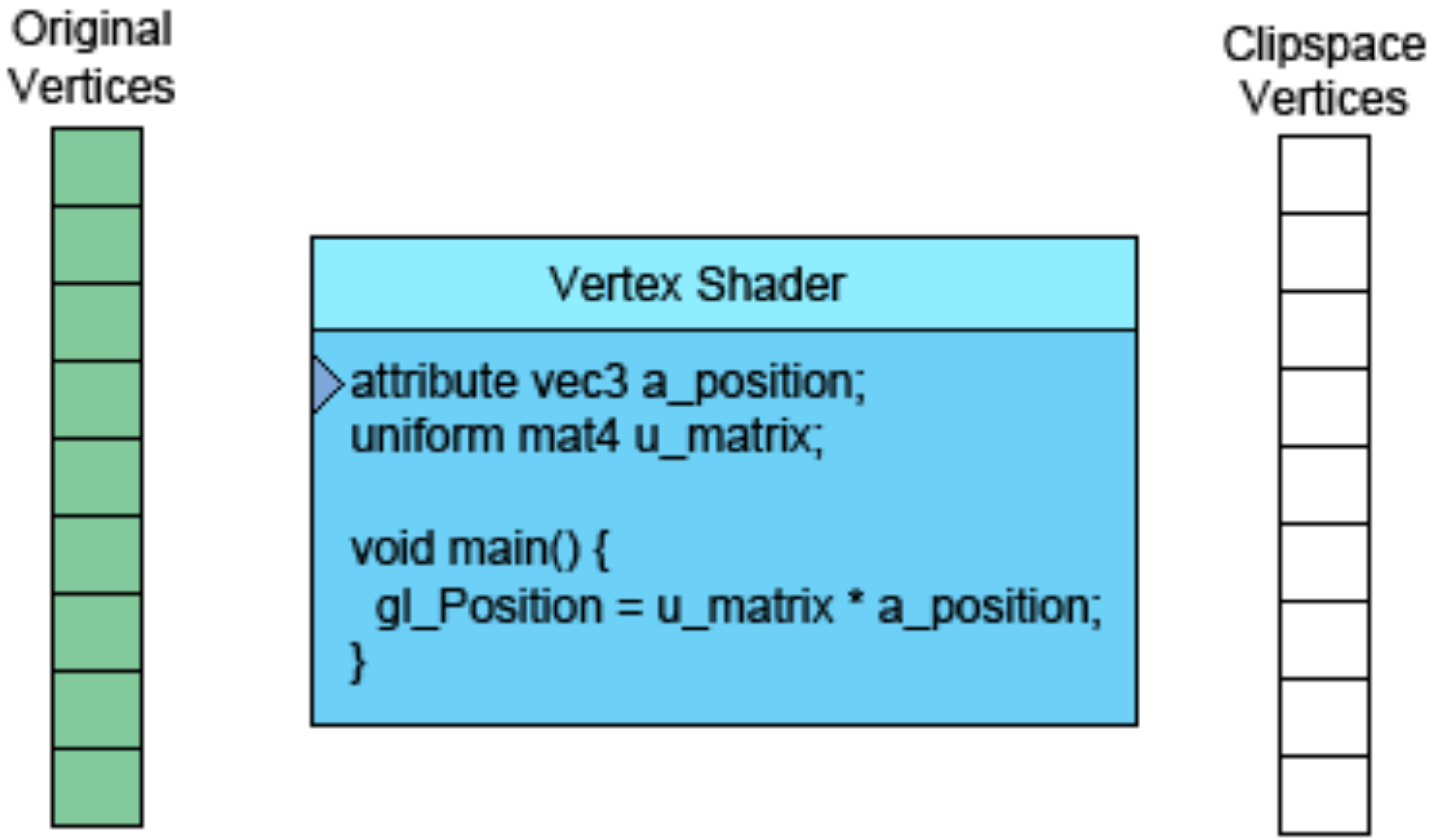
Simple vertex shader apply the model view project transformation:

```
// Pass MVP to shader - in cpp file
glm::mat4 MVP;
GLint m_MVPID = glGetUniformLocation(programID, "MVP");
glUniformMatrix4fv(m_MVPID, 1, GL_FALSE, &MVP[0][0]);
```

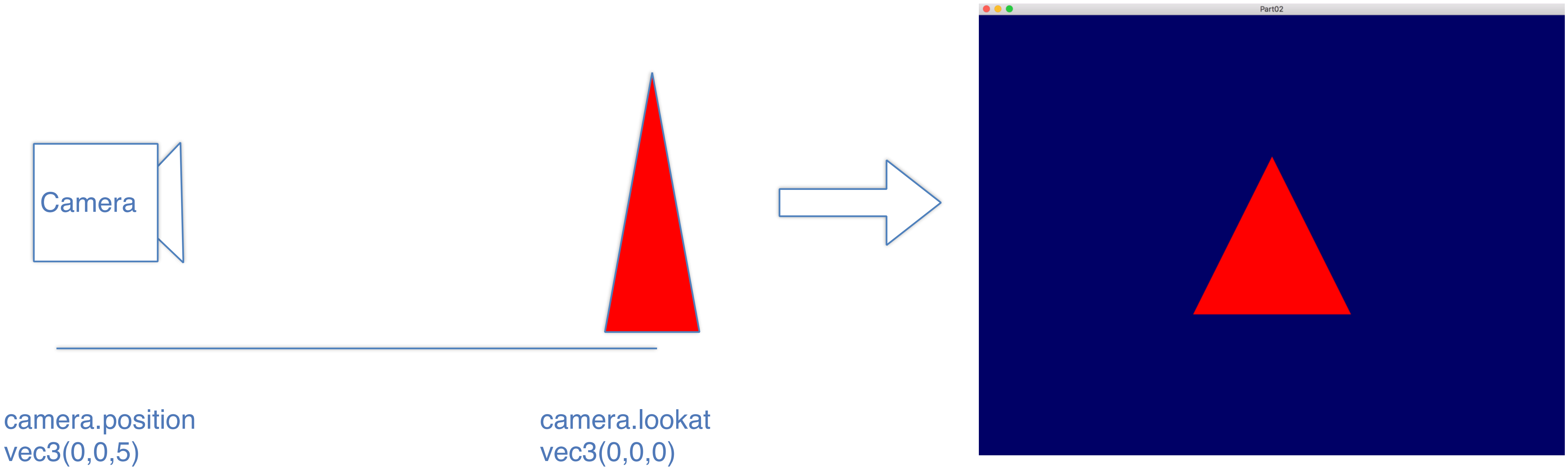
```
#version 330 core

// 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 MVP;
void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
}
```

Example combining Shader.cpp and basicShader.vert

Vertex Shader



Vertex Shader



Vertex Shader

```
//position
glm::vec3 position = m_camera->getPosition();
// Up vector
glm::vec3 up = glm::cross( right, direction );
// set camera's lookat
m_camera->setLookAt(position,position+direction,up );

//in camera class definition
m_viewMatrix       = glm::lookAt(
                                   m_position,           // Camera is here
                                   m_lookat, // and looks here : at the same position, plus "direction"
                                   m_up                  // Head is up (set to 0,-1,0 to look upside-down)
                                   );

//in shader class - passing to vertex shader
void Shader::updateMVP(glm::mat4 MVP){
    glUniformMatrix4fv(m_MVPID, 1, GL_FALSE, &MVP[0][0]);
}
```

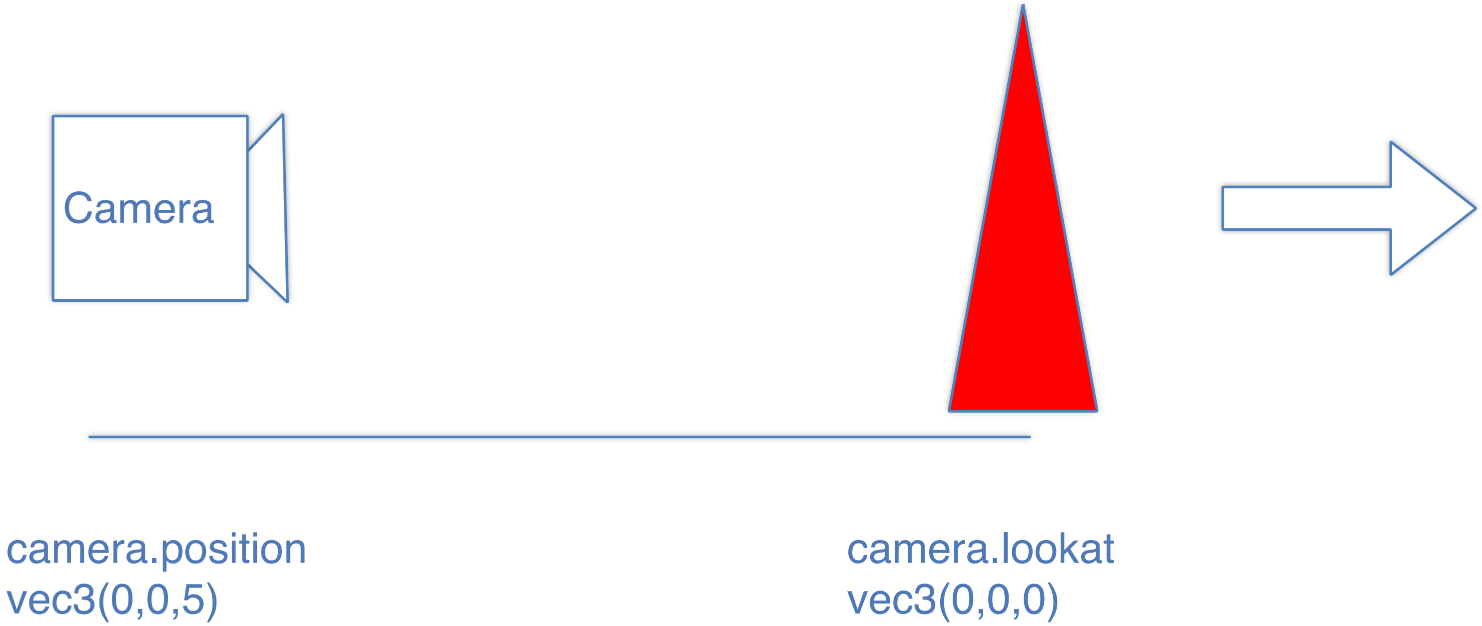
Camer

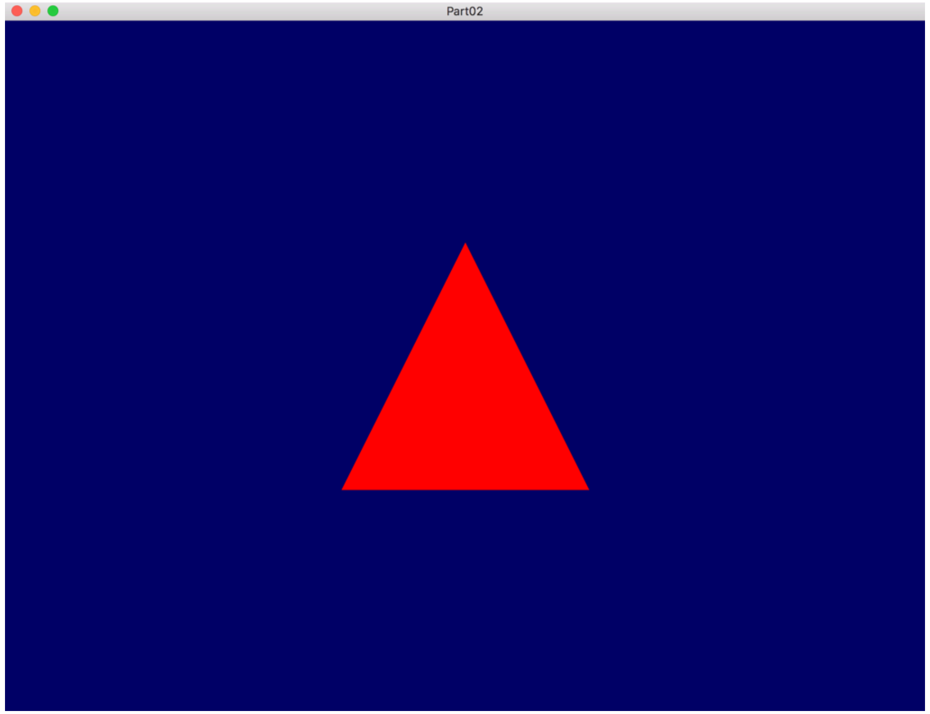
camera.position
vec3(0,0,5)

camera.lookat
vec3(0,0,0)

Example combining Shader.cpp and redTriangle.cpp

Vertex Shader





Fragment Shader

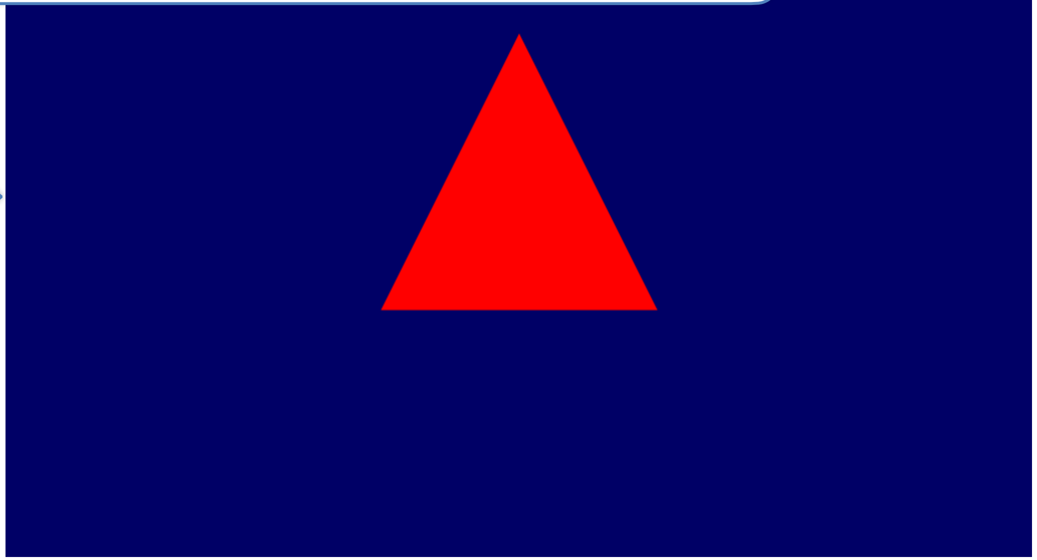
Example combining ColourShader.cpp and basicShader.frag

Simple Fragment Shader outputting gl_FragCoord:

```
#version 330 core

// Ouput data
out vec3 color;

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

}
```

basicShader.frag

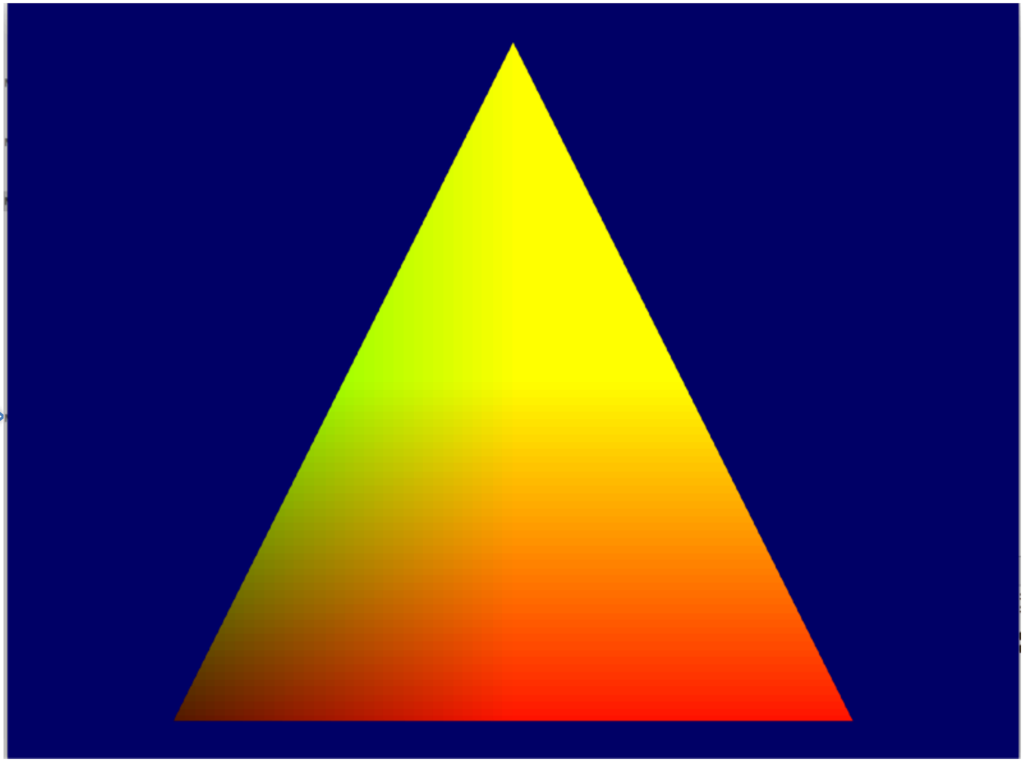
```
// add color parameter to shader - cpp file
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,0.0,0.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```



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);

}
```



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

Datatypes GLSL

- Basic Types
  - int, uint, float, bool: scalar numeric and logical types
  - ivec2, ivec3, ivec4: integer vectors
  - uvec2, uvec3, uvec4: unsigned integer vectors
  - bvec2, bvec3, bvec4: boolean vectors
- Floating-Point Vectors
  - vec2, vec3, vec4: 2D/3D/4D float vectors
  - Commonly used for positions, colors, and directions
- Matrices
  - mat2, mat3, mat4: 2×2, 3×3, 4×4 float matrices
  - Mixed forms also exist: mat2x3, mat3x4, etc.
  - Used for transforms (model, view, projection)

Datatypes GLSL

- Sampler and Image Types
  - sampler2D, samplerCube, sampler2DShadow, etc.
  - Represent textures bound to shader units
  - Accessed via functions like texture()
- Special Types
  - struct: custom user-defined types
  - array: fixed-size arrays of any GLSL type
  - in, out, uniform qualifiers specify data flow between stages
- Precision Qualifiers (ES / optional in desktop)
  - highp, mediump, lowp: control numeric precision

Additional Shader Stages in the OpenGL Pipeline

- Beyond the basic Vertex Shader and Fragment Shader, OpenGL includes optional programmable stages for finer control over geometry

- Tessellation Control Shader (TCS):
    - Defines how much a patch should be subdivided, enabling adaptive surface detail.
- Tessellation Evaluation Shader (TES):
    - Computes vertex positions of the tessellated patch, allowing smooth curved surfaces
- Geometry Shader:
    - Operates after vertex processing and can generate, modify, or discard entire primitives (e.g., expand points into quads, create outlines).
- These stages provide greater flexibility for procedural geometry, LOD, and displacement mapping

- When unused, the pipeline runs directly from the vertex shader to the rasterizer for efficiency

The end!