

COMP220: Graphics & Simulation

2: Shader programs

Learning outcomes

By the end of this session, you should be able to:

- ▶ **Explain** the role of shaders in graphics programming
- ▶ **Distinguish** the roles of the vertex shader and the fragment shader
- ▶ **Write** simple shader programs in GLSL

Agenda

- ▶ Lecture / live coding: experimenting with shaders
- ▶ Exercise - Working with OpenGL and Shaders

Assignments Reminder

Assignment 1

- ▶ Part A - Friday 5pm Week 3 - (feedback given in class)
- ▶ Worksheet A - Friday 5pm Week 4 (pull request)
- ▶ Worksheet B - Friday 5pm Week 7 (pull request)
- ▶ Worksheet C - Friday 5pm Week 9 (pull request)
- ▶ Worksheet D - Friday 5pm Week 12 (pull request)
- ▶ Final Handin - 7th of January 5pm (Learning Space)

Assignment 2

- ▶ Part A - Week 3 Workshop session (In class)
- ▶ Part B - 16th November 5pm (Learning Space)
- ▶ Part C - Viva 8th of January 5pm (Viva session)

GLSL

OpenGL Shading Language (GLSL)

- ▶ Used for writing **shaders** for OpenGL applications
- ▶ C-like syntax
- ▶ GLSL compiler is part of the **graphics driver** on the **end user's machine**
 - ▶ Yes, you need to ship your shader source code with your game!

Basic vertex shader

```
#version 330 core

layout(location = 0) in vec3 vertexPos;

void main()
{
    gl_Position.xyz = vertexPos;
    gl_Position.w = 1.0;
}
```

Basic vertex shader

```
#version 330 core
```

- Tells the compiler to use OpenGL 3.3 core functionality

Basic vertex shader

```
layout(location = 0) in vec3 vertexPos;
```

- ▶ Specifies **input values** to the vertex shader
- ▶ Corresponds with layout of **vertex buffers** in C++ program

Basic vertex shader

```
void main()
```

- ▶ Every shader program must define a **void** `main()` function

Basic vertex shader

```
gl_Position.xyz = vertexPos;  
gl_Position.w = 1.0;
```

- ▶ `gl_Position` is one of many **built-in** variables with special meaning
- ▶ See [https://www.opengl.org/wiki/Built-in_Variable_\(GLSL\)](https://www.opengl.org/wiki/Built-in_Variable_(GLSL))

Basic fragment shader

```
#version 330 core

out vec3 color;

void main()
{
    color = vec3(1, 1, 0);
}
```

Basic fragment shader

```
out vec3 color;
```

- ▶ By convention, fragment shader should have **one output**, namely the fragment colour
- ▶ Doesn't have to be named `color` — could be any other non-reserved identifier

Programming in GLSL

- ▶ `if` statements, `for` loops, `while` loops, `do while` loops, `switch` statements, `break`, `continue`, `return` all work the same as C++
- ▶ `//Single-line comments` and `/*Multi-line comments */` work the same too
- ▶ Function definitions and declarations are similar to C++, except that parameters must be declared as `in`, `out` Or `inout`
- ▶ Recursion is **forbidden**
- ▶ No `#include` — splitting a shader into multiple files is not easy...
- ▶ No `class`

Data types in GLSL

- ▶ `bool`, `int`, `float`: just like in C++
- ▶ `vec2`, `vec3`, `vec4`: **vectors** Of `floats`
 - ▶ Vectors in the mathematical sense, not the `std::vector` sense
- ▶ `mat2`, `mat3`, `mat4`: **square matrices** Of `floats`
- ▶ `mat2x3`, `mat3x2`, `mat4x2` etc: **rectangular matrices** of `floats`
- ▶ **Arrays** of constant size e.g. `float myArray[10]`
- ▶ There's no such thing as **pointers** in GLSL (hooray!)

Vectors

- ▶ An n -**dimensional vector** is formed of n numbers
- ▶ E.g. 2-dimensional vectors:

$$(1, 2) \quad (-2.7, 0) \quad (3.4, -12.7)$$

- ▶ E.g. 3-dimensional vectors:

$$(1, 2, 0) \quad (-9, 6, 3.7) \quad (2.1, 2.1, 2.1)$$

- ▶ Used to represent **points** or **directions** in n dimensions
- ▶ Also used to represent e.g. colours in RGB(A) space

Constructing vectors in GLSL

```
vec2 a = vec2(1.2, 3.4);  
vec3 b = vec3(1); // same as vec3(1, 1, 1)  
vec3 c = vec3(a, 5.6); // same as vec3(1.2, 3.4, 5.6)
```

Vector maths

Most operations work **component-wise**:

```
vec2 a = vec2(1, 2);  
vec2 b = vec2(3, 4);  
vec2 c = a + b; // c == vec2(4, 6);  
vec2 d = a * b; // d == vec2(3, 8);
```

Can also multiply a **vector** by a **scalar**:

```
vec2 e = 3.1 * a; // e == vec2(3.1, 6.2)
```

Accessing components

Can access the components of a vector as `.x`, `.y`, `.z`, `.w`:

```
vec4 a = vec4(1, 2, 3, 4);  
float b = a.y; // b == 2  
float c = a.z; // c == 3  
a.x = 5;       // a == vec4(5, 2, 3, 4)  
a.w = a.y;     // a == vec4(5, 2, 3, 2)
```

Can also use `r g b a` (for colours) and `s t p q` (for texture coordinates)

Swizzling

Can access multiple components in one go:

```
vec4 a = vec4(1, 2, 3, 4);  
vec2 b = a.xy;      // b == vec2(1, 2)  
vec3 c = a.zyz;     // c == vec3(3, 2, 3)  
a.xw = vec2(5,6);   // a == vec4(5, 2, 3, 6)  
a.xyzw = a.wzyx;    // a == vec4(6, 3, 2, 5)
```

- ▶ **Can** use the same component twice in the **right-hand side** of an assignment
- ▶ **Cannot** use the same component twice in the **left-hand side** of an assignment
- ▶ Swizzling is generally **faster** than the equivalent code without swizzling
- ▶ Can also use `r g b a` or `s t p q`, but can't mix them (e.g. `.gbr` is valid but `.gzx` is not)

Variables in GLSL

Passing values from the application to the shader

There are two ways:

- ▶ **Vertex attributes**

- ▶ Different values for each vertex
- ▶ More on this later in the module

- ▶ **Uniform variables**

- ▶ Constant across one `glDraw...` call

Uniform variables

In GLSL (outside `main()`):

```
uniform vec3 myVariable;
```

In C++:

```
GLuint location  
    = glGetUniformLocation(programID, "myVariable");
```

and then:

```
glUniform3f(location, 1, 2, 3);
```

Uniform variables

- ▶ `glGetUniformLocation` is expensive — do it on initialisation, not in the main loop
- ▶ Uniforms can be any GLSL type...
- ▶ ... but you must use the `glUniform...` function that matches the type

Passing values from the vertex shader to the fragment shader

Define an **out** variable in the vertex shader:

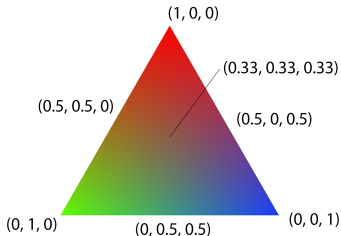
```
out vec4 myVariable;
```

Define an **in** variable **of the same name** in the fragment shader:

```
in vec4 myVariable;
```

Interpolation

- ▶ The vertex shader sets a value for each vertex
- ▶ So what is the value in the middle of the triangle?
- ▶ The GPU **interpolates** the value across the triangle



Exercise 1 - Shaders

- ▶ Make sure you can compile and run the demos from last week
- ▶ Bring in the shader loading code from the following -
`http://www.opengl-tutorial.org/
beginners-tutorials/
tutorial-2-the-first-triangle/`
- ▶ Add in a basic Vertex and Fragment shader based on the above link
- ▶ Compile and run the application

Exercise 2 - Working with Uniform Variables

- ▶ Add in a Uniform variable to the fragment shader, this should be a vec4 representing the colour of the triangle
- ▶ Send the colour across from the Application side (C++), you can use an array of floats or GLM Library to represent the colour on the C++ side
- ▶ Compile and run the application