# Week 3

# OpenGL Shading Lang. (GLSL)

- Part of OpenGL 2.0 onwards
- As of OpenGL 3.1, application programs must provide shaders (as no default shaders are available)
- High level C-like language
- New data types are provided
    - Matrices
    - Vectors

# Data Types

- C types: int, float, bool
- Vectors:
  - vec2, vec3, vec4;
    - Each element is a float
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2 (2x2), mat3, (3x3), mat4 (4x4)
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec2 a = vec2(3.0, 2.0);
  - vec3 b = vec3(a, 1.0);

# No Pointers

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types, they can be passed into and output from GLSL functions
    - mat3 func(mat3 a)

# GLSL Qualifiers

## GLSL:

- Has many qualifiers from C/C++ (e.g. const)
- We need other qualifiers
    - Due to the nature of the rendering pipeline
    - To modify the storage/behavior of global/local variables

We consider variables that can change

- Once per primitive
- Once per vertex
- Once per fragment
- At any time in the application

Reminder: Vertex attributes are interpolated by the rasterizer into fragment attributes

## Types

Qualifiers that can be used in shader programs (GLSL code) include:

- **Storage Qualifiers:** *const, attribute, uniform, varying*
- **Precision Qualifiers:** *highp, mediump, lowp, precision*
- **Parameter Qualifiers:** *in, out, inout*

## Storage Qualifiers

- *const*
- *attribute*
- *uniform*
- *varying*

const

## The qualifier **const**:

- Means the variable is **read only** (constant and cannot be changed)
- Means the variable must be initialized in its definition
- Used for compile-time constants or for read only function parameters

attribute

## The qualifier **`attribute`**

- Is used to declare variables that are shared between a vertex shader and the application program
- Makes the variable declared **read-only** in the vertex shader
- Means the variable must be **initialized in the init() function** the application program
    - They are in the scope of both

Typically, vertex coordinates passed to the vertex shader, e.g. vPosition:
**Example:**
```
attribute vec4 vPosition;
```

Vertex attributes are used to specify per vertex data. They typically provide data such as the object space position, the normal direction and the texture coordinates of a vertex.

## The qualifier **`uniform`**

- Is used to declare variables that are <u>shared between a shader and the application program.</u>
- Denotes a variable that appears in **both the fragment and vertex shaders**
  - Declaration must be identical in both
  - Global scope
- Describe **global properties** that affect the scene to be rendered (e.g. projection matrix, light source position, object properties (e.g., colour, materials))

  ```
  uniform mat4 projection;
  uniform float temperature;
  ```
- Make variables **not changeable** within the vertex shader or the fragment shader.
  - But their values can change in the application program
  - We pass new values inside display callback function, giving them to the shaders each frame

# varying

The qualifier **`varying`**:

- Is for variables that are **shared between the vertex and fragment shaders** (not with application program)
  - Must be declared identically in both shaders.
- Can only be used with **floating point** scalar/vector/matrices/arrays
- Is for variables used to store data calculated in the vertex shader and *pass it down to the fragment shader.*
  - Example:
    - The vertex shader can compute the colour of the incoming vertex and then pass the value to the fragment shader for interpolation.
    - Both shaders would have: `varying vec4 colour;`

## **Precision Qualifiers:** *highp, mediump, lowp, precision*

- Specify the level of precision available for a variable (high, med, low)
- Can appear in both the shaders
- In the fragment shader:
  - "precision" must precede the level qualifier (unless default precision exists for the datatype)
    - The default precision for int, float, and vectors of these types is highp
  - `precision lowp vec3 indices;`
- In the vertex shader
  - the use of a precision qualifier is optional
  - When none is given, the highest is the default

## Effect

The actual range corresponding to a precision qualifier is dependent on the specific application.

Using a lower precision might have a positive effect on performance (frame rates) and power efficiency but might also cause a loss in rendering quality.

The appropriate trade-off can only be determined by testing different precision configurations.

# Parameter Qualifiers

**Parameter Qualifiers:** *in, out, inout*

- In
  - Marks a parameter as **read-only** when a function is declared
  - This is the default
- Out
  - Marks a parameter as write-only when a function is declared
- InOut
  - Marks a parameter as *read-write* when a function is declared

Example

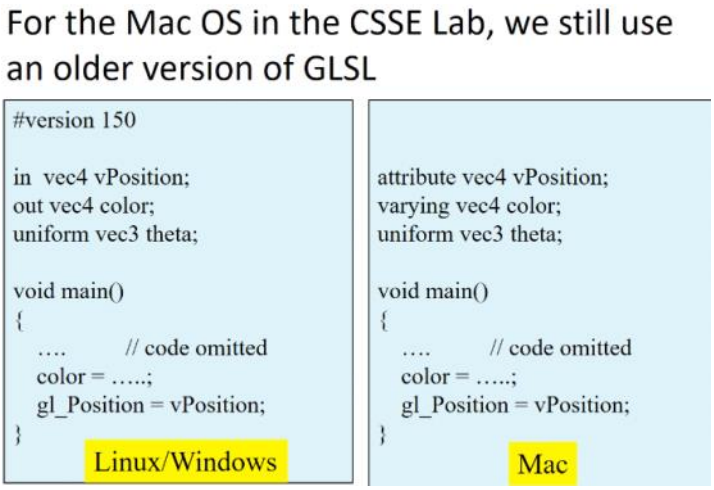
```
int newFunction(in bvec4 aBvec4, // read-only
                out vec3 aVec3, // write-only
                inout int aInt); // read-write
```

The usage of the read-only qualifier is not necessary since this is the default if no qualifier is specified.

## Replacements

The `in/out` qualifiers **replace** the `attribute` and `varying` qualifiers in GLSL V4.20 onward:

- `attribute` is replaced by `in` in the vertex shader
- `varying` in the vertex shader is replaced by `out`
- `varying` in the fragment shader is replace by `in`



# GLSL Built-In Functionalities

Wednesday, November 20, 2019 3:19 PM

Built In Variables

# gl_Position

# gl_FragColor

# gl_Position

## gl_Position

- is already known/declared, but **must be defined** in vertex shader
- is the position that will be passed to the rasterizer
- must be output by every vertex shader

**Example:**

- `in vec4 vPosition;`
    - `void main() {`
        - `gl_Position = vPosition;`
    - `}`
- The input vertex's location is given by the 4D vector 'vPosition'
- The keyword 'in' is to signify that its value is input to the shader when the shader is initialized

## gl_FragColor

- Used only on the lab Macs
- Value must be defined in the fragment shader
- Each invocation of the vertex shader **outputs a vertex**
- Each fragment invokes an *execution of the fragment shader*.
- Each execution of the fragment shader **must output a color for the fragment**
- **Example:**
    - ```
      void main()
      {
      gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
      }                     (R,  G, B, Opacity)
      ```
      ```

## Selection and Swizzling

Can refer to array elements by their indices using [] or by selection operator (.)

- x, y, z, w (indices 0, 1, 2, 3)
- r, g, b, a
- s, t, p, q
- `vec4 m;`
- `m[2]`, `m.b`, `m.z`, and `m.p` are the same

The **Swizzling Operator** lets you initialize/swap components easily

```
vec4 a;
a.yz = vec2(1.0, 2.0);
vec4 newColour = v.bgra; // swap red and blue
```

## Features

### Overloading of vector and matrix types

- ```
  mat4 A;
  vec4 b, c, d;
  c = b*A; // not implemented in Angel.h
  d = A*b; // a column vector stored as a 1d array
  ```

### Matrice initialization

```
mat3 theMatrix;
theMatrix[1] = vec3(3.0, 3.0, 3.0); // Sets the 2nd column
theMatrix[2][0] = 16.0; // Sets the 1st entry of 3rd  column
```
```

# Functions

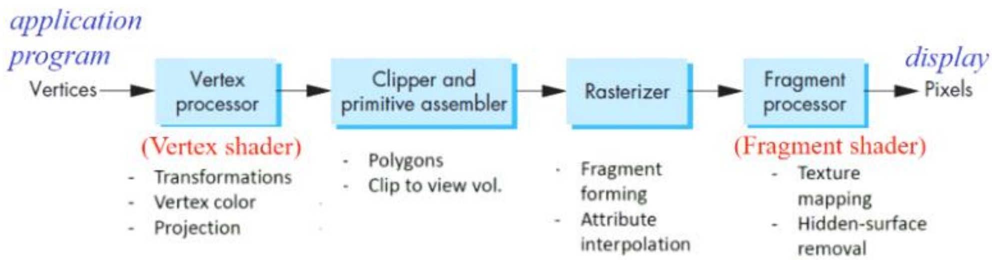
GLSL has the Standard C functions

- Trigonometric
- Arithmetic
- Normalize, reflect, length

**Shaders** are small programs that are ***compiled and run on the GPU***

Multiple shader programs can be invoked and run in parallel to render complex scenes in real-time.

## Role in Rendering Pipeline




**Vertex Shader**

- Comes first in pipeline
- Purpose is to provide the *final transformation of mesh vertices* to the rendering pipeline.

**Fragment Shader**

- Comes last in pipeline
- Provides the colour for each pixel in the frame buffer and decide which ones get displayed

**Clipping** is then performed on a primitive by primitive basis rather than a point-by-point basis. Clipping is the process of removing parts of objects that are outside the viewing volume.

**Primitive Assembler** collects/groups vertices into geometric objects such as line segments, polygons, curves and surfaces.

# Rasterizer

**Rasterizer** produces a set of fragments for each object that is not clipped out.

Fragments are potential pixels which have a location (in the frame buffer), colour, depth and alpha attributes.

Rasterizer interpolates vertex attributes (colour, transparency) over the object.

## The Vertex Shader

In the rendering pipeline, each vertex is processed independently.

The vertex shader ***processes one vertex***

The vertex shader takes in one vertex from the vertex stream as input and ***generates the transformed vertex*** (optionally with attributes) to the output vertex stream.

# Vertex Shader Uses

The **Vertex Shader** can be used for **per vertex operations**:

- Geometric transformations
  - Change relative location, rotation, scale of objects/camera
  - Apply 3D perspective transformation — make far objects smaller
- Moving vertices
  - Performing morphing (smoothly moving vertices to form a new object)
  - Compute wave motion and deformation due to physical forces
  - Simulate particle effects (fire, smoke, rain, waterfalls)
  - Compute fractals (with loops not recursion)
- Lighting
  - Calculate shading color using light and surface properties
  - Calculate cartoon shading (for special effects)

## Simple Example

```glsl
// GLSL Version 1.50
#version 150

// in = Input from application
// vPosition must be mentioned in application
in vec4 vPosition;

void main(void)
{
    // Built in variable
    // We assign the vertex position to the built in variable
    gl_Position = vPosition;
}
```

## More Complex Shaders

```
#version 150
// Input vertex position
in vec4 vPosition;
// Vertex shader can produce output for the rasterizer and fragment shader further down the pipeline
out vec4 color;
// Uniform
uniform vec3 theta;

void main() {
    // Code omitted
    color = ....., // Compute the out variable color
    gl_Position = vPosition; // May be a more complex expression
}
```

## Wave Motion Vertex Shader

```
in vec4 vPosition;
uniform float h, xs, zs; // Height scale, Frequencies

void main ( ) {
   vec4 t = vPosition; // Temporary variable

   t.y = vPosition.y // Y component being changed
   + h*sin(time + xs*vPosition . x)
   + h*sin(time + zs*vPosition.z);

   gl_Position = t;
}
```

## Particle System Vertex Shader

| | |
|---|---|
| `in vec3 vPosition;`<br>`uniform mat4 ModelViewProjectionMatrix;`<br>`uniform vec3 vel;`<br>`uniform float g, m, t;`<br>`void main() {`<br>`    vec3 object_pos;`<br>`    object_pos.x = vPosition.x + vel.x*t;`<br>`    object_pos.y = vPosition.y + vel.y*t`<br>`                   + g/(2.0*m)*t*t;`<br>`    object_pos.z = vPosition.z + vel.z*t;`<br>`    gl_Position = ModelViewProjectionMatrix *`<br>`                  vec4(object_pos,1);`<br>`}` | In = Will be initialized in init() function<br>Uniform = varying<br>Gravity, mass and time variables<br><br>We make a temporary variable (object_pos) and assign x and y values<br><br>Final value is multiplied by project matrix |

# The Fragment Shader

Tuesday, 3 March 2020 9:22 AM

## Fragment Shader Uses

**The Fragment Shader** is used for **per fragment** operations:

- Recall that a fragment is a potential pixel that not only has location coordinates but also has colour, depth, and alpha values
- **Lighting Calculations**
  - 

    per vertex lighting per fragment lighting
- **Texture Mapping**
  - 

    smooth shading environment mapping bump mapping

Simple Example

```
#version 150

// fragcolor is an output variable of the shader
out vec4 fragcolor

void main()
{
   // Fragcolor must be computed and output
   // Mac version would use inbuilt gl_FragColor
   fragcolor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

## Shaders Working Together

**Vertex Shader**

```
#version 150

const vec4 red =
   vec4(1.0, 0.0, 0.0, 1.0);
in vec4 vPosition;
out vec4 color_out;

void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

**Fragment Shader**

```
#version 150


in vec4 color_out;
out vec4 fragcolor;

void main(void) {
  fragcolor = color_out;
}
// in pre-OpenGL 3.2
// versions, use built-in:
// gl_FragColor = color_out;
```

**Out variables** declared in the **vertex shader** must be <u>In variables</u> in the <u>fragment shader</u>

These are for Linux/Windows

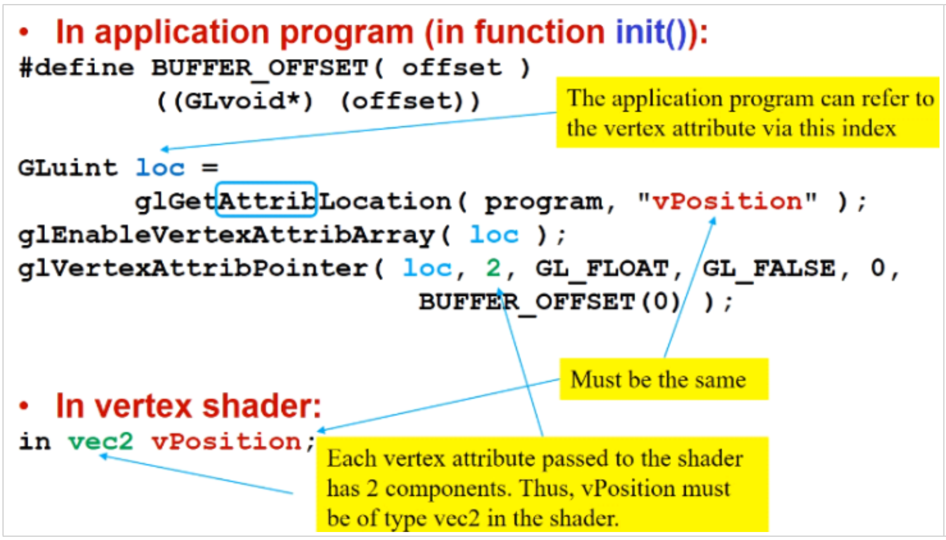
For Mac:

- Varying variables in the vertex shader must be varying in the fragment shader
- Inbuilt frag color is used

For each variable with an **`attribute/in/uniform`** qualifier in the **vertex shader**, its name is stored in a table.

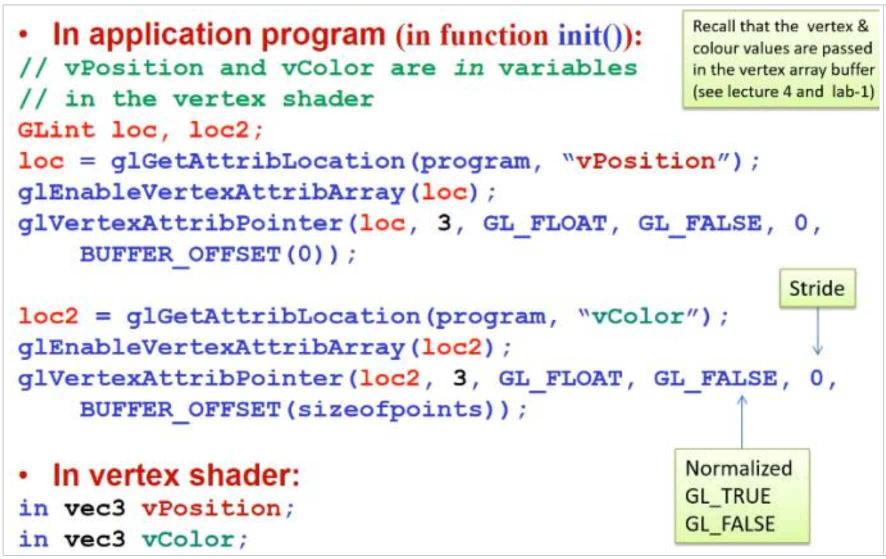
The application program can get an index for each variable from the table.

## Example with One `in` Variable




2nd parameter = 2 means that each vertex is 2 dimensional (vec2)

## Example with Two `in` Variables




We utilize the **offset function** here

## Example with `uniform` Variable