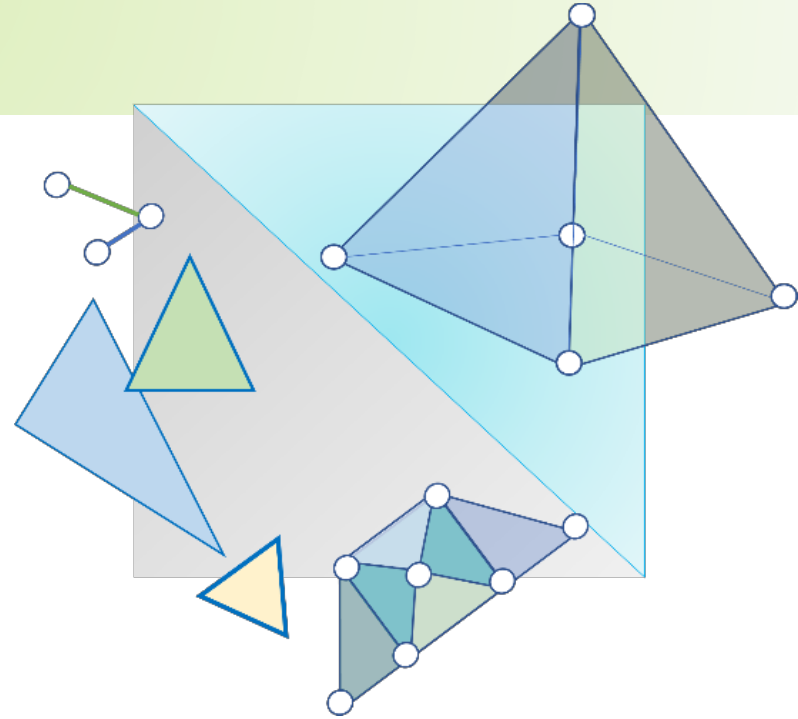


CITS3003 Graphics & Animation

Lecture 5: Vertex and Fragment Shaders-1



Content

- The rendering pipeline and the shaders
- GLSL
 - Data types, qualifiers, built-in variables, and functions in shaders
 - Swizzling & selection

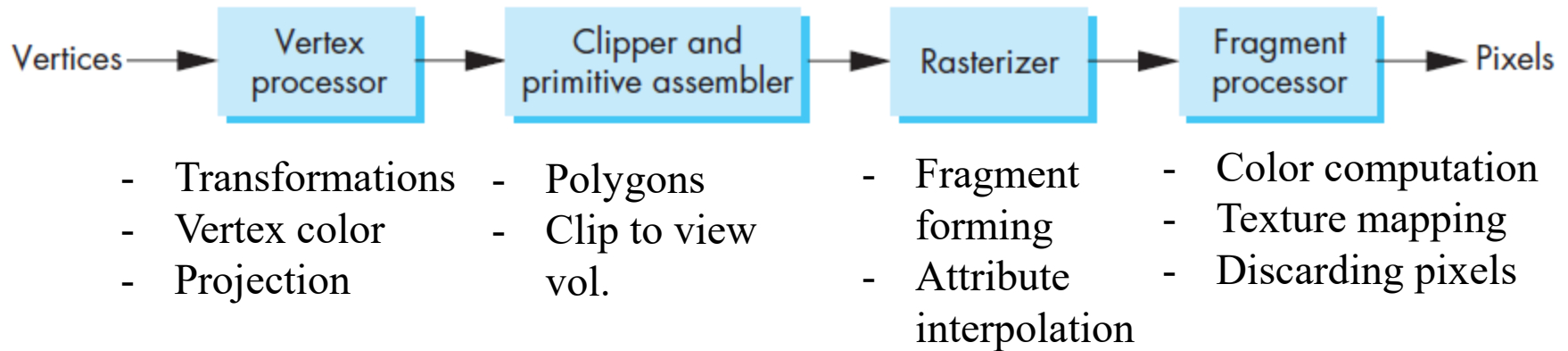
The Rendering Pipeline and the Shaders

Where are the vertex and fragment shaders on the rendering pipeline:

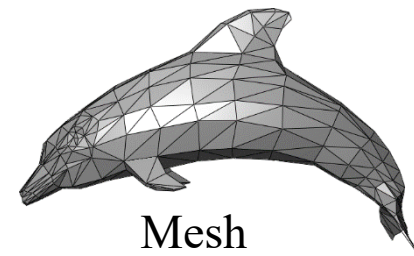
application

program (Vertex shader)

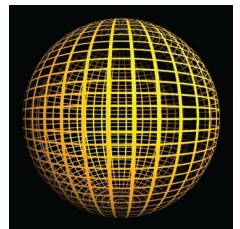
(Fragment shader) *display*



- The goal of the **vertex shader** is to provide the final transformation of mesh vertices to the rendering pipeline.
- The goal of the **fragment shader** is to provide the colour to each pixel in the frame buffer.



Mesh



A sphere with some parts made invisible by discarding pixels in the fragment shader.



GLSL Shading Language

GLSL – A Quick Review

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types are provided, e.g.
 - Matrices
 - Vectors
- As of OpenGL 3.1, application programs must provide shaders (as no default shaders are available)

Data Types in GLSL

- **Scalar (non-vector) types:**
 - **bool** // e.g., bool var = true;
 - **Int** // e.g., int var = -10;
 - **uint** // e.g., uint var = 10;
 - **float** // e.g., float var = 0.5;
 - **double** // e.g., double var = 0.12567;
- **Vectors:** Each of the scalar types, *including booleans*, have 2, 3, and 4-component *vector equivalents*. The n digit below can be 2, 3, or 4:
 - **bvecn**: a vector of booleans
 - **ivec n**: a vector of signed integers //e.g., ivec2 myvar = ivec2(-1, 2);
 - **uvec n**: a vector of unsigned integers
 - **vec n**: a vector of single-precision floating-point numbers
 - **dvec n**: a vector of double-precision floating-point numbers

Data Types in GLSL

Matrices: All matrix types are floating-points

- **matn**: A matrix with n columns and n rows. Shorthand for $\text{mat}n \times n$
- **matnxm**: A matrix with n columns and m rows
- Double-precision matrices (OpenGL 4.0 and above) can be declared with a **dmat** instead of **mat**

mat3 Matrix;

Matrix[1] = vec3(1.0, 1.0, 1.0); // Sets the second column to all 1.0.

Matrix[2][0] = 15.0; // Sets the first entry of the third column to 15.0.

$$\text{Matrix} = \begin{pmatrix} 0.0 & 1.0 & 15.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{pmatrix}$$

Pointers

- There are **no pointers** in GLSL
- We can use C structs
- Because matrices and vectors are basic types, they can be passed into and output from GLSL functions, e.g.

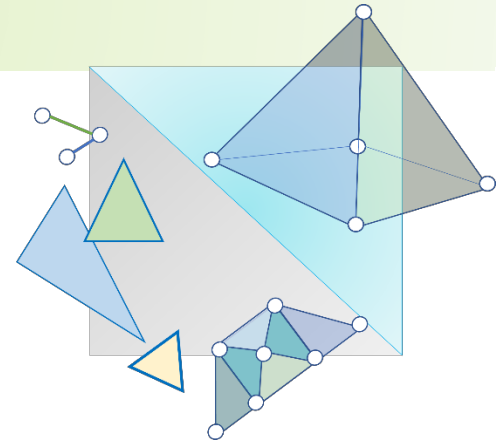
mat3 func(mat3 a);

GLSL Qualifiers

- A qualifier is used in GLSL to modify the storage or behaviour of variables. Qualifiers specify particular aspects of the variable, e.g., where they will get their data from?
- GLSL has some of the same qualifiers as C/C++, e.g., **const**. However, more are required due to the nature of the rendering pipeline
- Qualifiers that can be used in shader programs include:
 - **attribute, uniform, varying** (these are **storage qualifiers**)
 - **highp, mediump, lowp** (these are **precision qualifiers**)
 - **in, out, inout** (these are **parameter qualifiers**)

Precision qualifiers in GLSL are supported for compatibility with OpenGL ES

Storage Qualifiers



Qualifier *Constant*

- The qualifier **const** is used the same as in Java. It specifies that the value assigned to a variable is constant and cannot be changed. The variable is read only. Here are a legal and illegal statement.

// a vector assigned a value

```
const vec4 point = vec4(1.0, 2.0, 3.0, 1.0);
```

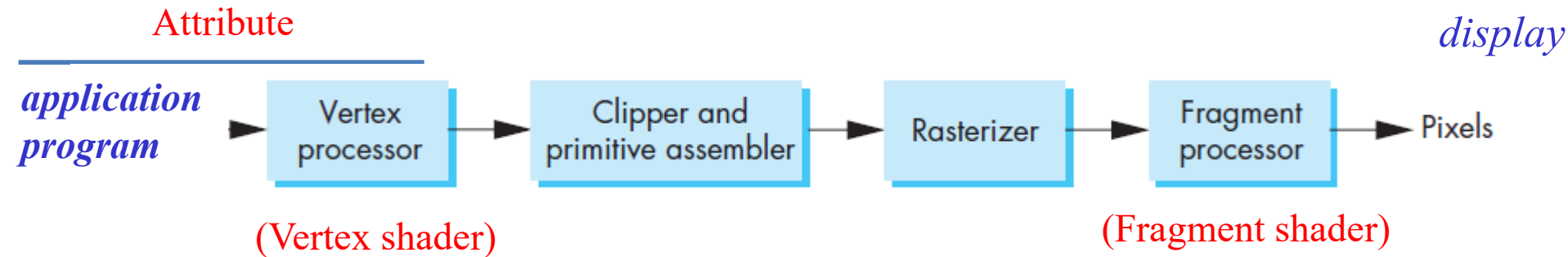
// illegal statement because the const variable must be initialized when declared

```
const float time;
```

- The qualifier **const** is used for variables that are compile-time constants or for function parameters that are read-only.
 - must be initialized when declared
- These variables:
 - cannot be used on the left-hand side of an assignment operation
 - cannot be referenced outside the shader that defines it

Qualifier *attribute*

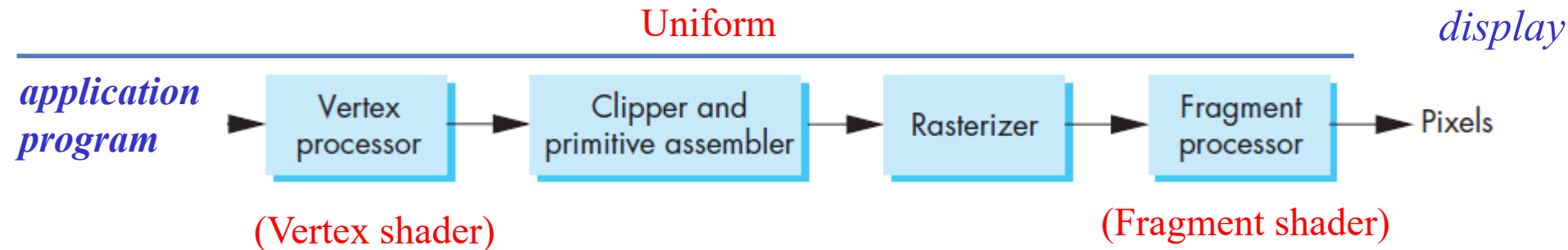
- The qualifier **attribute** is used to declare variables that are shared between a **vertex shader** and the application program; typically used for vertex coordinates passed to the vertex shader, e.g.,
- **attribute** vec4 vPosition;



- Attribute variables
 - specify per vertex data, e.g., object space position, the normal direction and the texture coordinates of a vertex.
 - must be initialized in the application program.
 - Have only float, vec, and mat data types, cannot be declared as structs

The Qualifier *uniform*

- The qualifier **uniform** is used to declare variables that are shared between a shader and the application program.



- Uniform variables:
 - can appear in the vertex shader and the fragment shader
 - must have a global scope.
 - do not change from one **shader invocation** to the next within a particular **rendering call** thus their value is **uniform** among all invocations
- If a **uniform** variable is used in both shaders, its declaration must be identical in both.

The Qualifier *uniform* (cont.)

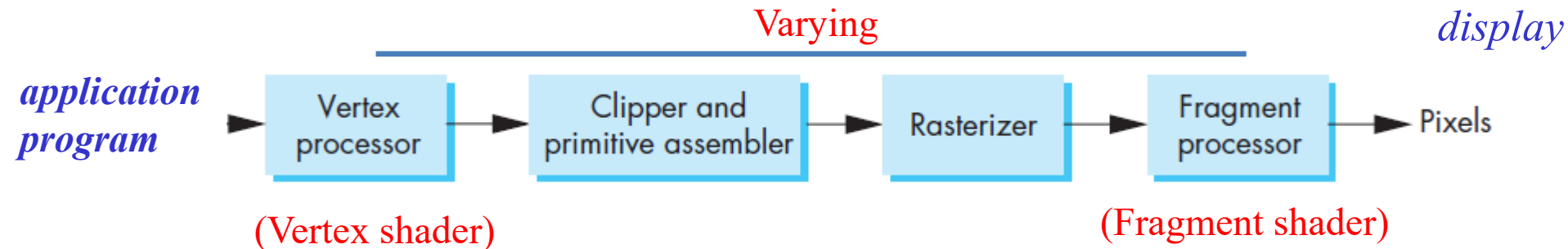
- **uniform** variables remain constant across a graphics primitive. They can be used to describe **global properties** that affect the scene to be rendered, e.g., projection matrix, light source position, etc. or **object** properties (e.g., colour, materials). e.g.,

uniform mat4 projection;
uniform float temperature;

- Variables declared as uniform are not changeable within the vertex shader or the fragment shader.
- However, their values can change in the application program. For each frame to be rendered, their new values are passed to the shader(s).

The Qualifier *varying*

- The qualifier **varying** is used to declare variables that are shared between the vertex shader and the fragment shader.



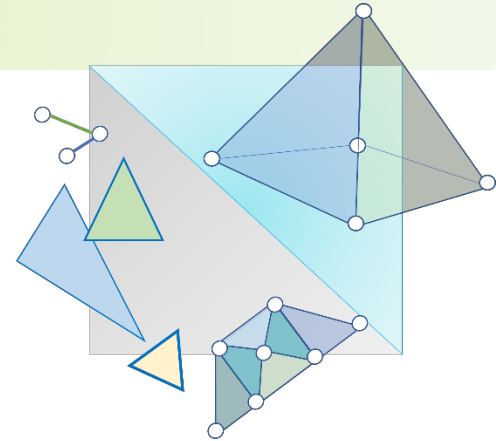
- varying** variables
 - are used to communicate results from one shader to another
 - must be declared identically in both shaders.

The Qualifier *varying* (cont.)

- The **varying** qualifier can only be used with floating point scalar, floating point vectors and (floating point) matrices as well as arrays containing these types.
- Example: the vertex shader can compute the color of the incoming vertex and then pass the value forward for interpolation. In both shaders:

varying vec4 colour;

Precision Qualifiers



The Qualifiers *highp*, *mediump*, *lowp*, and *precision*

- Supported for compatibility with OpenGL ES
 - Use is not recommended unless OpenGL ES compatibility is required
- The **highp**, **mediump**, and **lowp** qualifiers are used to specify the highest, medium, and lowest precision available for a variable. All these qualifiers can appear in the vertex and fragment shaders.
- All variables of a certain type can be declared to have a precision by using the **precision** qualifier

precision precision-qualifier type;

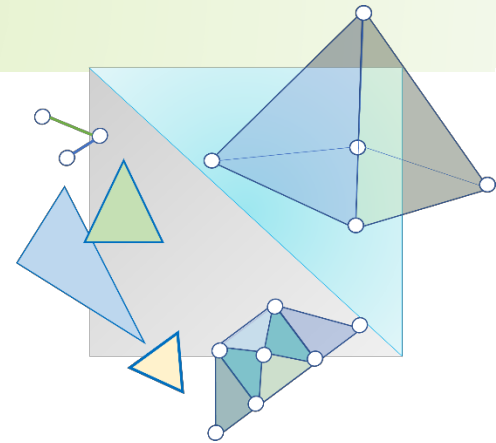
e.g., float



The Qualifiers *highp*, *mediump*, *lowp*, and *precision*

- The default precision is **highp**.
- 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



The Qualifiers *in*, *out* & *inout*

Function Parameter Qualifiers

- GLSL functions are declared and defined similarly to C/C++ functions. A function declaration in GLSL looks like this

```
void myFunc(in float inputVal, out int outputVal, inout float inAndOutVal)
```

Parameter qualifiers



- The values passed to functions are copied into parameters when the function is called, and outputs are copied out when the function returns (“value-return” calling convention)

The Qualifiers *in*, *out* & *inout*

```
void myFunc(in float inputVal, out int outputVal, inout float inAndOutVal)
{
    inputVal = 0.0;
    outputVal = int(inAndOutVal + inputVal);
    inAndOutVal = 3.0;
}
```

```
void main()
{
    float in1 = 10.5;
    int out1 = 5;
    float out2 = 10.0;
    myFunc(in1, out1, out2);
}
```

After myFunc is called

in1	10.5
out1	10
out2	3.0

The Qualifiers *in*, *out* & *inout*

- ❑ A parameter declared as **in** is intended to have a value when it is passed into a function but is not to be changed in the function.
 - Such parameters are intended to communicate only from the calling function to the called function.

- ❑ The **out** parameter is not assumed to have an initial value the first time it appears in the function. But it is assumed that a value will be assigned before the function returns.
 - Such parameters are intended to communicate only from the called function to the calling function.

- ❑ The **inout** parameter is intended to have a value when it is passed into a function and to have a value, possibly different, when the function returns.
 - Such parameters are intended to provide two-way communication between the called function and the calling function.

The Qualifiers *in*, *out* & *inout*

- The **in** and **out** qualifiers supersede the **attribute** and **varying** qualifiers in GLSL version 130+ 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**

An **out** variable is to get its value in the shader where it is defined and be passed from that shader to the next shader further along in the shader pipeline. It is a write-only variable in the shader where it is defined.

An **in** variable is to be received from a previous shader/application in the pipeline and used in the shader where it is defined. It is a read-only variable in the shader where it is defined.

Passing Values

- Variable declared **out** in vertex shader can be declared as **in** in fragment shader and used

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
```

```
out vec3 color_out;
```

```
void main(void) {
```

```
    gl_Position = vPosition;
```

```
    color_out = red;
```

```
}
```

Vertex
shader



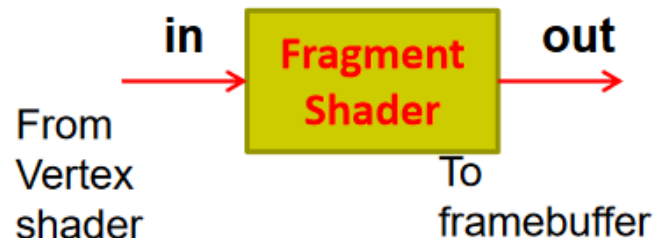
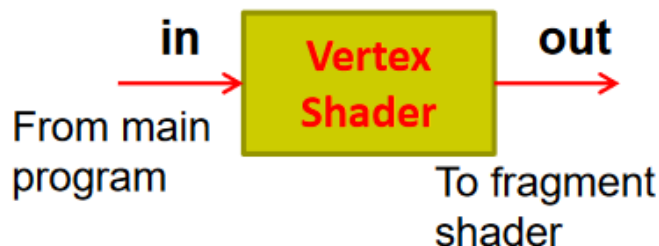
```
in vec3 color_out;
```

```
void main(void) {
```

```
    // can use color_out here.
```

```
}
```

Fragment
shader



slido



Choose the correct option(s)

① Start presenting to display the poll results on this slide.

Qualifiers

Qualifier	Variable specification/behaviour
const	Compile-time constants or function parameters that are read-only.
attribute	Shared between a vertex shader and application program for per-vertex data.
Uniform	Shared between application and shaders, does not change across the primitive.
varying	Shared between the vertex shader and the fragment shader for interpolated data.
in	Function parameter that is read only, or read-only. An in variable is to be received from a previous shader/application in the pipeline and used in the shader where it is defined. It is a read-only variable in the shader where it is defined.
out	Function parameter that is write-only. An out variable is to get its value in the shader where it is defined and be passed from that shader to the next shader further along in the pipeline. It is a write-only variable in the shader where it is defined.
inout	Function parameter that is intended to have two-way communication between the called function and the calling function.

Built-in variables in Shaders

- **gl_Position**

- Its value must be defined in the vertex shader

```
in vec4 vPosition;
```

```
void main()
```

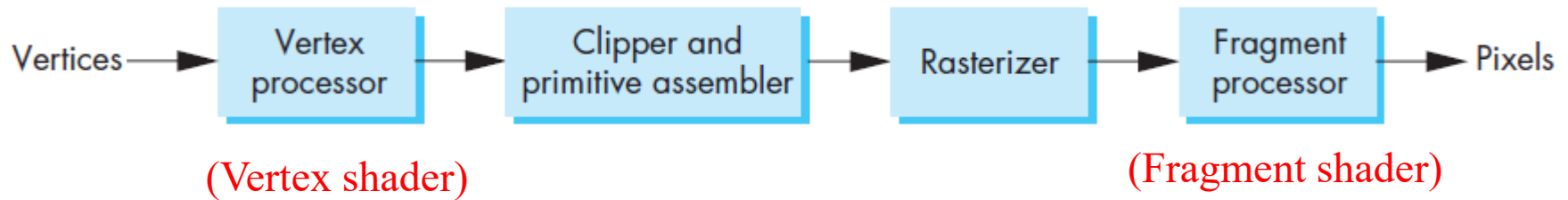
```
{
```

```
    gl_Position = vPosition;
```

```
}
```

- The input vertex's location is given by the four-dimensional vector vPosition whose specification includes the keyword ***in*** to signify that its value is input to the shader when the shader is initiated.
- **gl_Position** is a special state variable, which is the position that will be passed to the rasterizer and must be output by every vertex shader. Because gl_Position is known to OpenGL, we need not declare it in the shader.

Built-in Variables in Shaders



- **gl_FragColor**

- Now deprecated
- Its value must be defined in the fragment shader

```
void main()
```

```
{
```

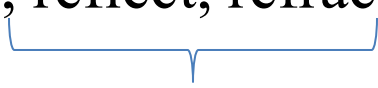
```
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
```

```
}
```

(R, G, B, Opacity)

- 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.

Functions and Operators

- Standard C functions
 - **Trigonometric:** cos, sin, tan, etc.,
 - **Arithmetic:** min, max, log, abs, etc.,
 - Normalize, reflect, length
- Examples
 - float **length**(TYPE x)
 - float **distance**(TYPE x1, x2)
 - TYPE **normalize**(TYPE x)
 - Other examples are dot, cross,  reflect, refract
 - Reflection/refraction direction for an incident vector
- If you are performing an operation in GLSL that is somewhat graphics specific, check the documentation if there is an inbuilt function for it

Functions and Operators

- **Operators**

- Binary operators `*`, `/`, `+`, `-`, `=`, `*=`, `/=`, `+=`, `-=` used between vectors of the same type, and work component-wise

```
vec3 a = vec3(1.0, 2.0, 3.0);  
vec3 b = vec3(0.1, 0.2, 0.3);  
vec3 c = a + b; // = vec3(1.1, 2.2, 3.3)  
vec3 d = a * b; // = vec3(0.1, 0.4, 0.9)
```

Products act component-wise when

- One operand is a scalar and one is either a vector or matrix, or
- Both are vectors.

But `*` does not work component-wise for matrix multiplication

$$AB = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

Performs
correct linear
algebra
operations

```
mat2 a = mat2(1., 2., 3., 4.);  
mat2 b = mat2(10., 20., 30., 40.);  
mat2 c = a * b; //  
= mat2(1. * 10. + 3. * 20., 2. * 10. + 4. * 20.,  
1. * 30. + 3. * 40., 2. * 30. + 4. * 40.)
```

$$\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix} * \begin{bmatrix} 10.0 & 30.0 \\ 20.0 & 40.0 \end{bmatrix} = \begin{bmatrix} 70.0 & 150.0 \\ 100.0 & 220.0 \end{bmatrix}$$

Operators and Functions

- For component matrix multiplication **matrixCompMult** is provided
- The ***** operator can also be used for matrix-vector product

$$\mathbf{M}\mathbf{v} = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{1,1}v_1 + m_{1,2}v_2 \\ m_{2,1}v_1 + m_{2,2}v_2 \end{bmatrix}$$

```
vec2 v = vec2(10., 20.);  
mat2 m = mat2(1., 2., 3., 4.);  
vec2 w = m * v; // = vec2(1. * 10. + 3. * 20., 2. * 10. + 4. * 20.)
```


Selection and Swizzling

- Can refer to array elements by their indices using `[]` or by **selection operator** `(.)` with
 - `x, y, z, w` % 3D coordinates and perspective scale
 - `r, g, b, a` % Color values and opacity
 - `s, t, p, q` % texture coordinates (later)
 - `vec4 m;`
 - `m[2], m.b, m.z, and m.p` are the same
- } name sets
- GLSL supports some standard name sets for vector components

<code>v4.rgba</code>	Is a <code>vec4</code> and is the same as just using <code>v4</code> .
<code>v4.rgb</code>	Is a <code>vec3</code> made from the first three components of <code>v4</code> .
<code>v4.b</code>	Is a float whose value is the third component of <code>v4</code> ; also <code>v4.z</code> or <code>v4.p</code> .
<code>v4.xz</code>	Is a <code>vec2</code> made from the first and third components of <code>v4</code> ; also <code>v4.rb</code> or <code>v4.sp</code> .
<code>v4.xgba</code>	Is illegal because the component names do not come from the same set.

Selection and Swizzling

- Can refer to array elements by their indices using `[]` or by **selection operator** `(.)` with
 - `x, y, z, w` % 3D coordinates and perspective scale
 - `r, g, b, a` % Color values and opacity
 - `s, t, p, q` % texture coordinates (later)
 - `vec4 m;`
 - `m[2], m.b, m.z,` and `m.p` are the same
- **Swizzling** operator lets us manipulate components easily, e.g.,
`vec4 a; // (0.0, 0.0, 0.0, 0.0)`
`a.yz = vec4(1.0, 2.0); // (0.0, 1.0, 2.0, 0.0)`
`vec4 newColour = a.bgra; // swap red and blue // (2.0, 1.0, 0.0, 0.0)`

name sets

GLSL supports some standard name sets for vector components

This lets you rearrange or reorganize the components of a vector.

You can also duplicate some of the components of a vector

Selection and Swizzling with Matrices

- Swizzling **does not work** with matrices. You can instead access a matrix's fields with array syntax:

```
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
```

- However, the result of the first array accessor is a vector, so you can swizzle that:

```
mat3 theMatrix;  
theMatrix[1].yzx = vec3(3.0, 1.0, 2.0);
```

References

“Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL” by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec2. 2.8.2-2.8.5 The Vertex Shader ...The InitShader Function
- Sec 3.12.2 Uniform Variables

Graphics Shaders (second edition) Bailey and Cunningham

- GLSL Shader Language
- Vertex Shader
- Fragment Shader