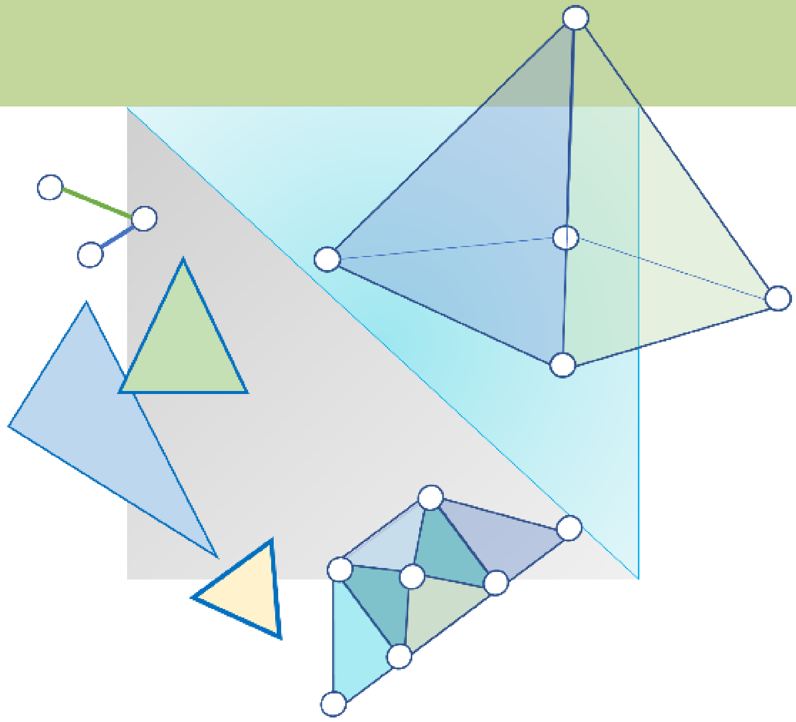
CITS3003 Graphics & Animation

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

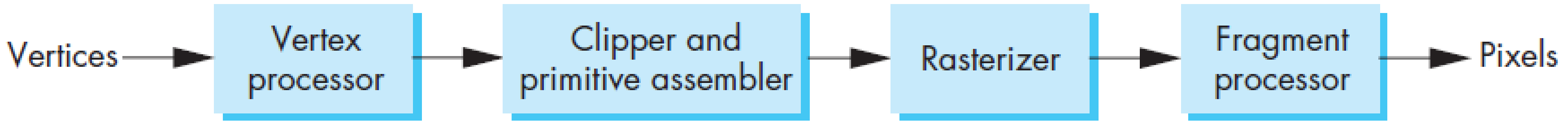
GLSL – A Quick Review

- 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)

The Rendering Pipeline and the Shaders

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

application program (Vertex shader) (Fragment shader) *display*



- Transformations
- Vertex color
- Projection

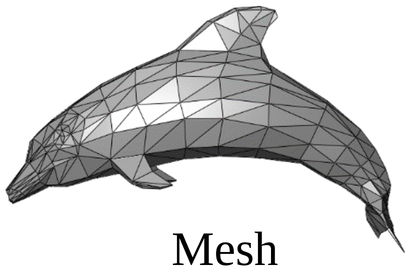
- Polygons
- Clip to view vol.

- Fragment forming
- Attribute interpolation

- Texture mapping
- Hidden-surface removal

- 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

Data Types in GLSL

- **Scalar (non-vector) types**:
  - bool
  - int
  - uint
  - float:
  - double
- **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
  - ivecn: a vector of signed integers
  - uvecn: a vector of unsigned integers
  - vecn: a vector of single-precision floating-point numbers
  - dvecn: 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 mat*nxn*
- 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);
```

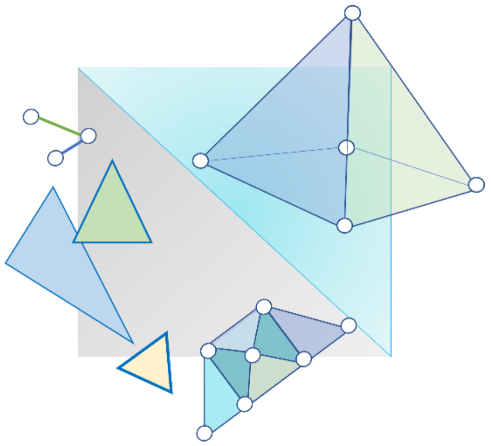
GLSL Type Qualifiers

- A type 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 (these are **parameter qualifiers**)

Precision qualifiers in GLSL are supported for compatibility with OpenGL ES

Reminder:

- We expect data exchange between the application program and shaders
- We expect data to move along in the pipeline
  - vertex attributes are interpolated by the rasterizer into fragment attributes



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.

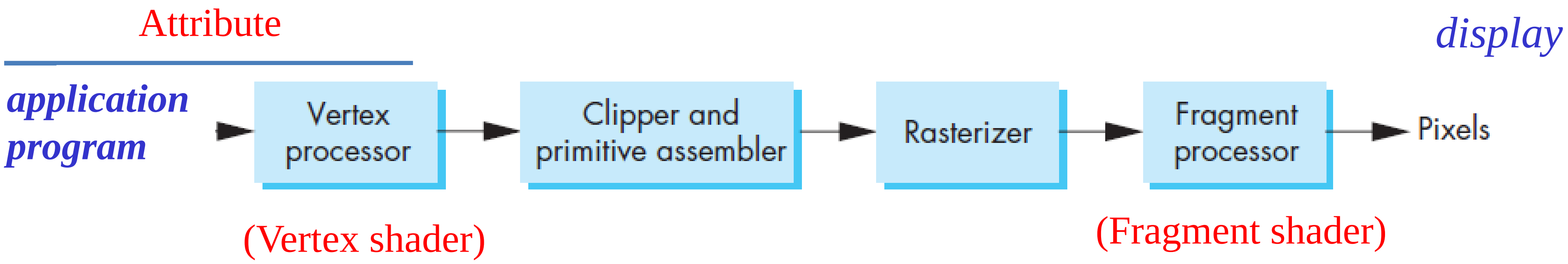
```
// a vector assigned a value
        const vec4 point = vec4(1.0, 2.0, 3.0, 1.0);
// illegal statement because the variable must be assigned a value
        const float time;
```

- The qualifier **const** is used for variables that are compile-time constants or for function parameters that are read-only.

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.,

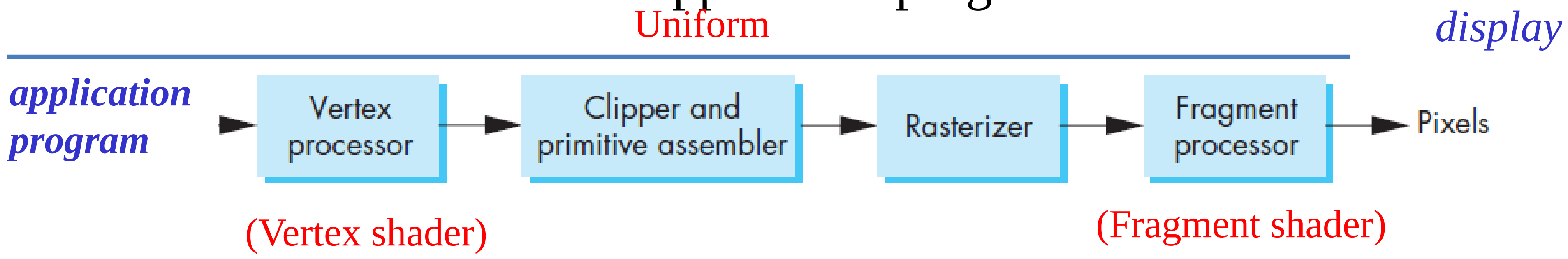
attribute vec4 vPosition;



- Vertex attributes specify per vertex data, e.g., object space position, the normal direction and the texture coordinates of a vertex.
- Variables declared using this qualifier must be initialized in the application program.

The Qualifier *uniform*

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



- Variables declared using this qualifier can appear in the vertex shader and the fragment shader and they must have a global scope.
- Uniforms are so named because they do not change from one **shader invocation** to the next within a particular **rendering call** thus their value is uniform among all invocations
- Since both shaders share the same name space, if a uniform variable is used in both shaders, its declaration must be identical in both.

The Qualifier *uniform* (cont.)

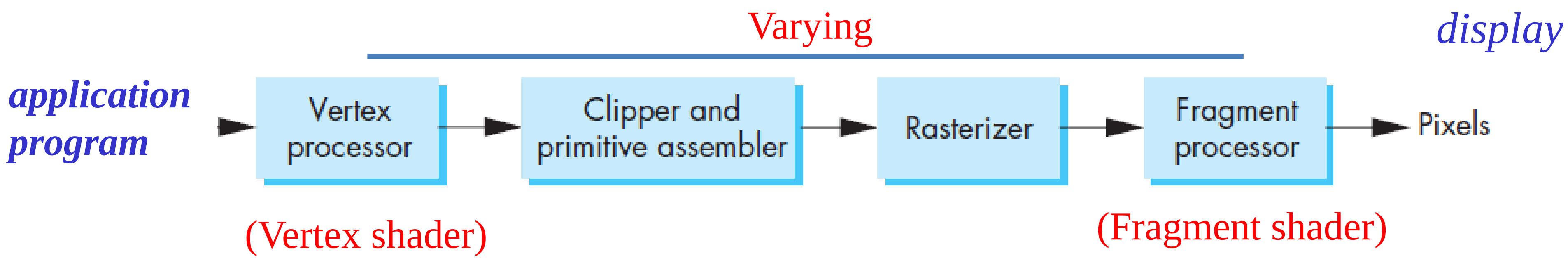
- uniform variables are used to describe **global properties** that affect the scene to be rendered, e.g., projection matrix, light source position, etc. They can be used to describe **object properties** (e.g., colour, materials). E.g.,

  ```
  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 store data calculated in the vertex shader and to pass down to the fragment shader. Again, because of the sharing of name space of the two shaders, varying variables 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*

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

```
void myFunct(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 myFunct(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;
  myFunct(in1, out1, out2);
}
```

Value not initialized
by the calling code

After myFunct is called

in1	10.5
out1	10
out2	3.0

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

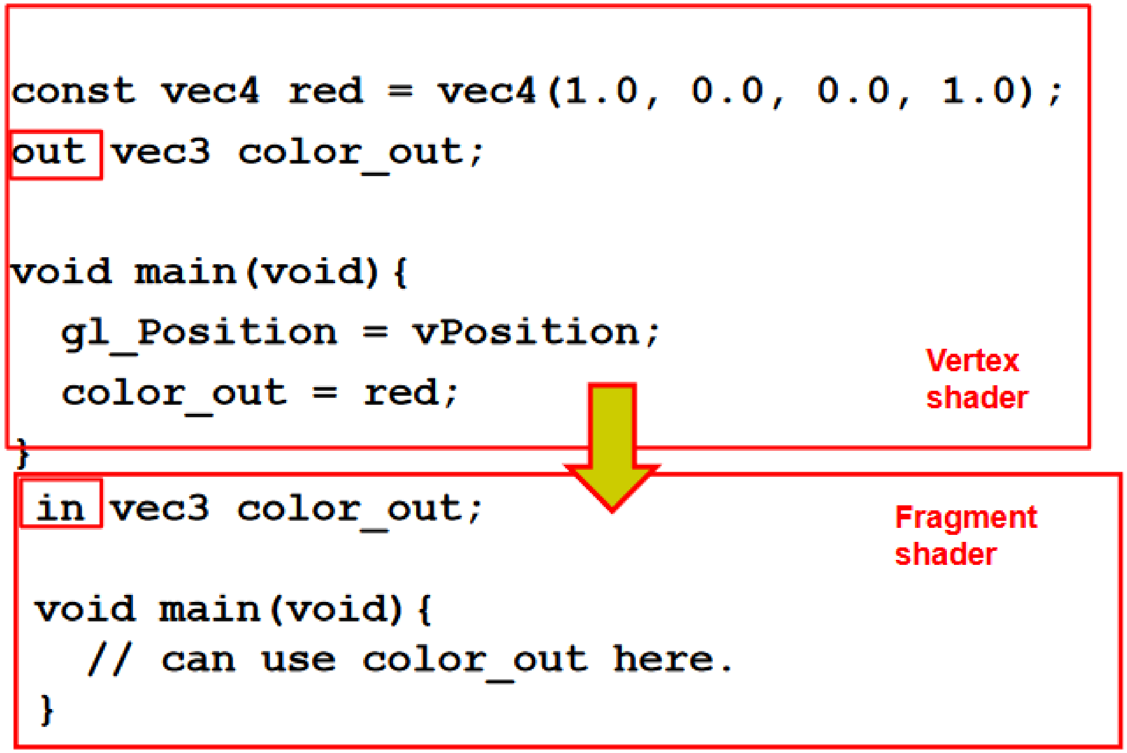
- A parameter declared as **in** means that the value given to that parameter will be *copied into* the parameter when the function is called. The function may then modify that parameter, but those changes will not affect the calling code.
- A parameter declared as **out** will not have its value initialized by the caller. The function will modify the parameter, and after the function's execution is complete, the value of the parameter will be copied out into the variable that the user specified when calling the function.
- The **inout** declaration combines both. The parameter’s value will be initialized by the value supplied by the calling code, and its final value will be output.

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

- The in and out qualifiers supersede the attribute and varying qualifiers in GLSL version 4.20 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

Passing Values

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

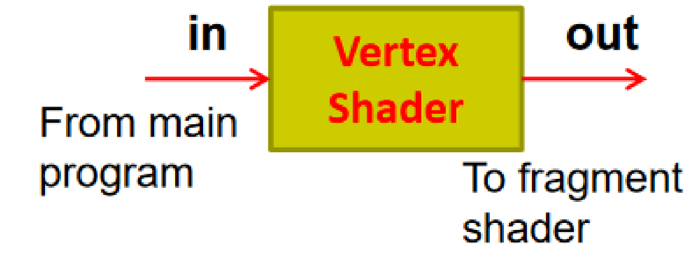


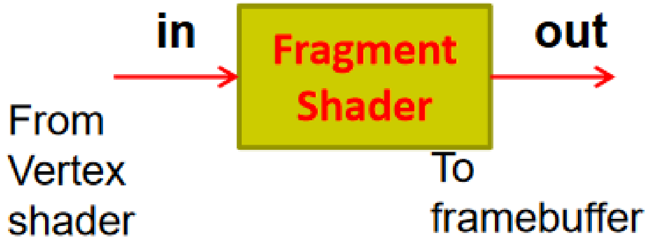
```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

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

```
 in vec3 color_out;

 void main(void){
   // can use color_out here.
 }
```





Slide Credits: Prof Emmanuel Agu, CS4731, lecture 6

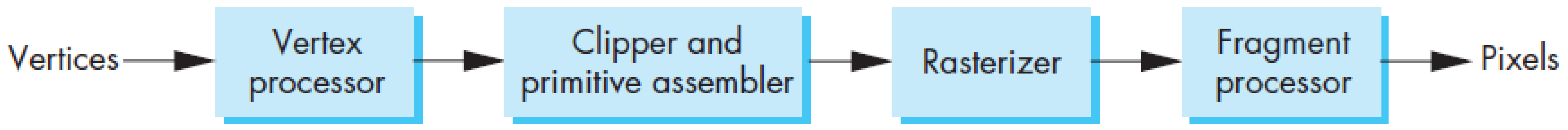
Built-in variables in Shaders

- **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



(Vertex shader) (Fragment shader)

- **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, $\underbrace{\text{reflect, refract}}_{\text{Reflection/refraction direction for an incident vector}}$

- 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, 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)
```

But * does not work for matrix multiplication like that

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

```
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.)
```

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

$\left.\right\}$ Three masks

- **Swizzling** operator lets us manipulate components easily, e.g.,

  vec4 a; // (0.0, 0.0, 0.0, 0.0)
  a.yz = vec2(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)

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

A good reference on OpenGL shaders:
http://antongerdelan.net/opengl/shaders.html