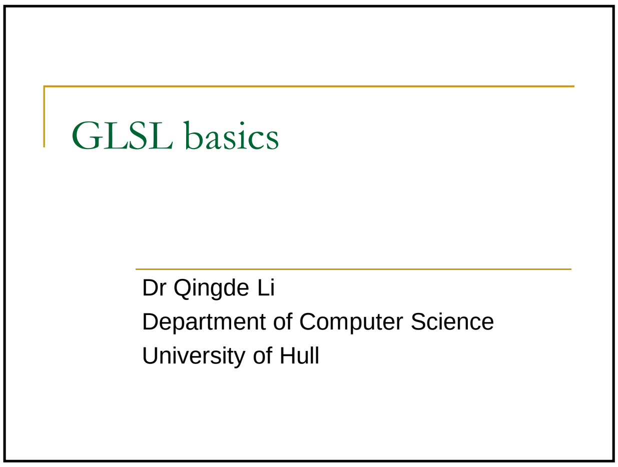
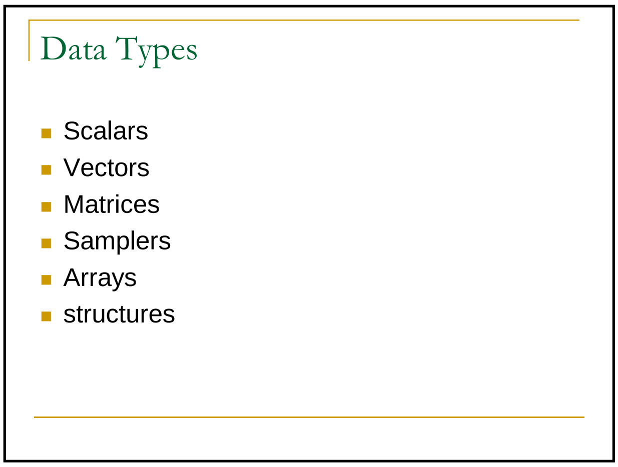


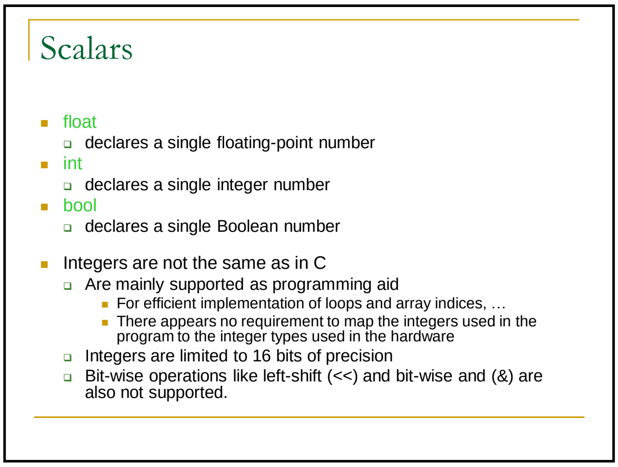
GLSL basics

Dr Qingde Li
Department of Computer Science
University of Hull



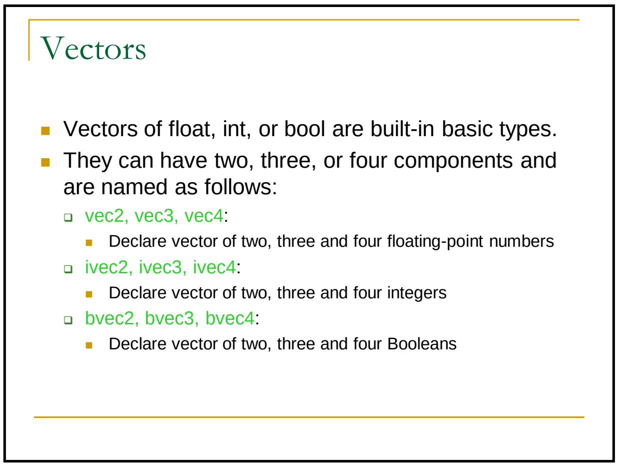
Data Types

- Scalars
- Vectors
- Matrices
- Samplers
- Arrays
- structures



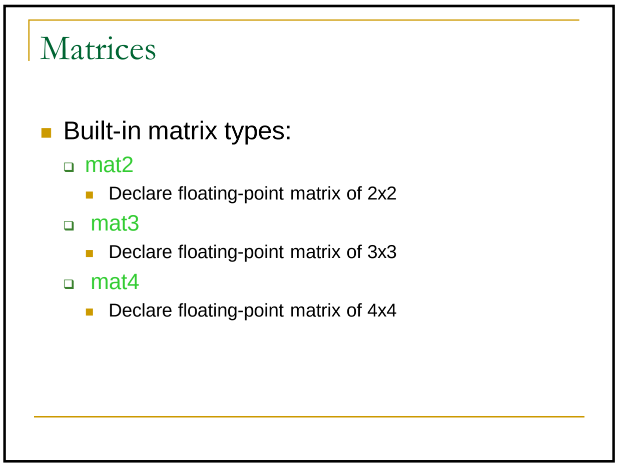
Scalars

- float
  - declares a single floating-point number
- int
  - declares a single integer number
- bool
  - declares a single Boolean number
- Integers are not the same as in C
  - Are mainly supported as programming aid
    - For efficient implementation of loops and array indices, …
    - There appears no requirement to map the integers used in the program to the integer types used in the hardware
  - Integers are limited to 16 bits of precision
  - Bit-wise operations like left-shift (<<) and bit-wise and (&) are also not supported.



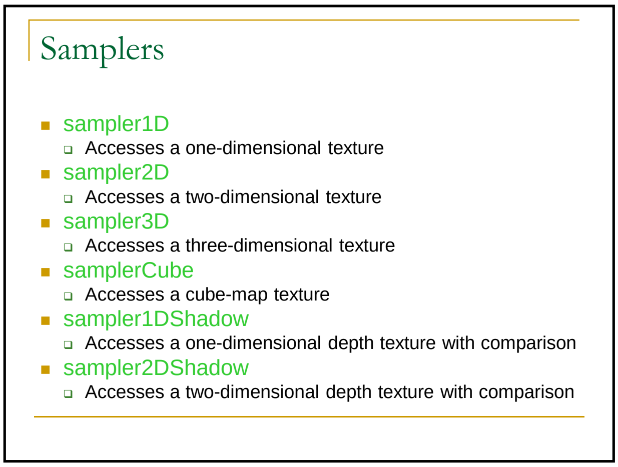
Vectors

- Vectors of float, int, or bool are built-in basic types.
- They can have two, three, or four components and are named as follows:
  - vec2, vec3, vec4:
    - Declare vector of two, three and four floating-point numbers
  - ivec2, ivec3, ivec4:
    - Declare vector of two, three and four integers
  - bvec2, bvec3, bvec4:
    - Declare vector of two, three and four Booleans



Matrices

- Built-in matrix types:
  - mat2
    - Declare floating-point matrix of 2x2
  - mat3
    - Declare floating-point matrix of 3x3
  - mat4
    - Declare floating-point matrix of 4x4



Samplers

- sampler1D
  - Accesses a one-dimensional texture
- sampler2D
  - Accesses a two-dimensional texture
- sampler3D
  - Accesses a three-dimensional texture
- samplerCube
  - Accesses a cube-map texture
- sampler1DShadow
  - Accesses a one-dimensional depth texture with comparison
- sampler2DShadow
  - Accesses a two-dimensional depth texture with comparison

Type Matching and Promotion

- The OpenGL Shading Language is strict with type matching
  - In general, types being assigned must match
  - argument types passed into functions must match formal parameter declarations
  - and types being operated on must match the requirements of the operator.
  - There are no automatic promotions from one type to another
  - This may occasionally make a shader have an extra explicit conversion. However, it also simplifies the language, preventing some forms of obfuscated code and some classes of defects.

Initialize and construct

- Similar to c. For example

```
float a, b = 3.0, c;
const int Size = 4; // initializer is required
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
vec4 v0;
v0 = vec4(1.0, 2.0, 3.0, 4.0);
```

- For matrices, the components are written in **column major** order. For example,

```
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

Will give a matrix $\begin{pmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{pmatrix}$

Initialize and construct

- Matrices can be constructed in different ways:
  - To initialize the diagonal of a matrix with all other elements set to zero:

```
mat2(float)
mat3(float)
mat4(float)
```

  - To initialize a matrix by specifying vectors , the components are assigned to the matrix elements in column-major order.

```
mat3(vec3, vec3, vec3); // one column per argument
mat4(vec4, vec4, vec4, vec4); // one column per argument
```

Qualifiers and Interface to a Shader

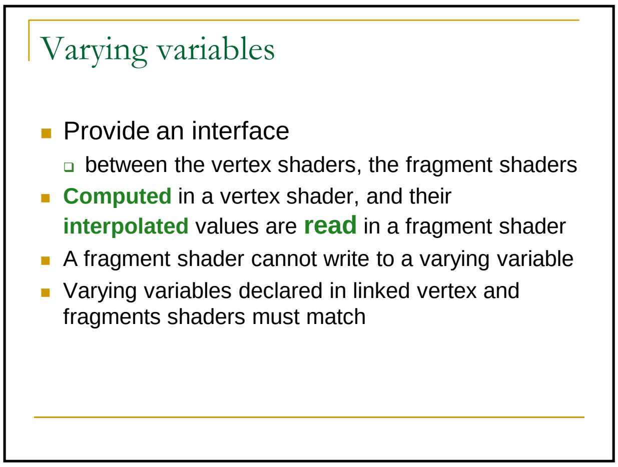
- attribute
  - linkage between a **vertex shader** and **OpenGL** for per-vertex data
- uniform
  - value does not change across the primitive being processed
  - form the **linkage** between a **shader**, **OpenGL**, and the **application**
- varying
  - linkage between a **vertex shader** and a **fragment shader** for interpolated data
- const
  - a compile-time constant, or a function parameter that is read-only

Attribute Qualifiers

- Used to declare variables that are passed **to a vertex shader from OpenGL** on a per-vertex basis.
  - Eg. Normal, Tengent, Binormal
- Can only be declare in vertex shaders
- Read only
  - Attributes cannot be modified
- Used only with floating-point scalars, vectors, and matrices.
  - Attributes declared as integers or Booleans are not allowed, nor are attributes declared as structures or arrays.
- They convey vertex attributes to the vertex shader and are expected to change on every vertex shader run

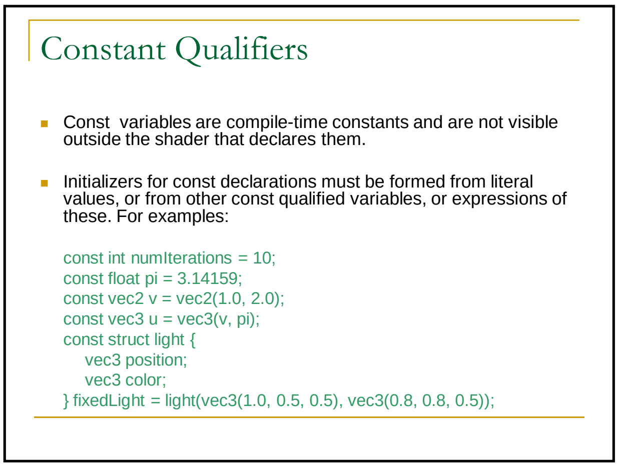
Uniform Qualifiers

- Used to declare global variables whose values are the same across the entire primitive being processed
- are **read-only** and are **initialized externally** either at link time or through the API
- All data types are supported
- Uniform variables of the same name in a vertex and fragment program will be the same



Varying variables

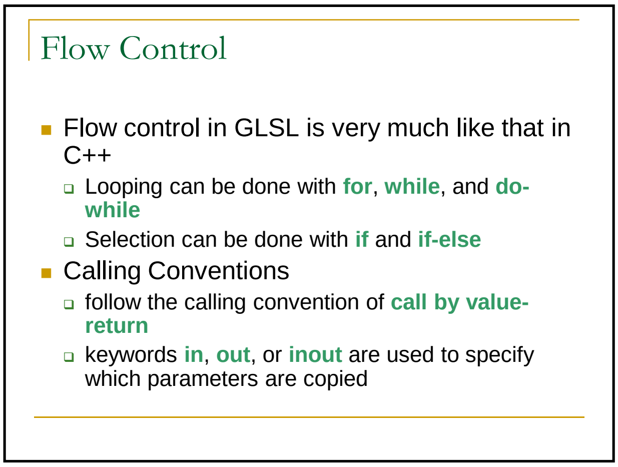
- Provide an interface
  - between the vertex shaders, the fragment shaders
- **Computed** in a vertex shader, and their **interpolated** values are **read** in a fragment shader
- A fragment shader cannot write to a varying variable
- Varying variables declared in linked vertex and fragments shaders must match



Constant Qualifiers

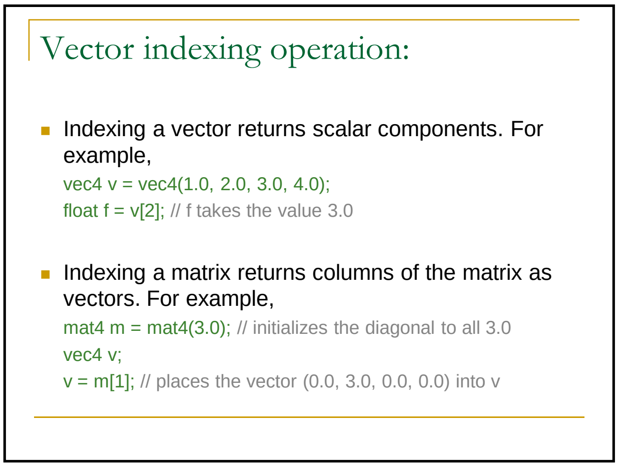
- Const variables are compile-time constants and are not visible outside the shader that declares them.
- Initializers for const declarations must be formed from literal values, or from other const qualified variables, or expressions of these. For examples:

```
const int numIterations = 10;
const float pi = 3.14159;
const vec2 v = vec2(1.0, 2.0);
const vec3 u = vec3(v, pi);
const struct light {
    vec3 position;
    vec3 color;
} fixedLight = light(vec3(1.0, 0.5, 0.5), vec3(0.8, 0.8, 0.5));
```



Flow Control

- Flow control in GLSL is very much like that in C++
  - Looping can be done with **for**, **while**, and **do-while**
  - Selection can be done with **if** and **if-else**
- Calling Conventions
  - follow the calling convention of **call by value-return**
  - keywords **in**, **out**, or **inout** are used to specify which parameters are copied



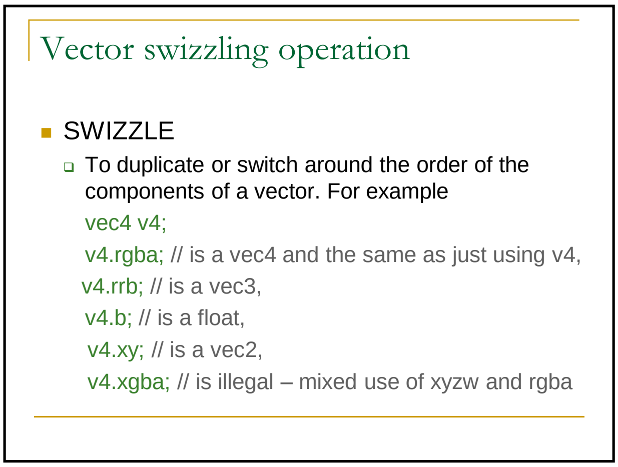
Vector indexing operation:

- Indexing a vector returns scalar components. For example,

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
float f = v[2]; // f takes the value 3.0
```

- Indexing a matrix returns columns of the matrix as vectors. For example,

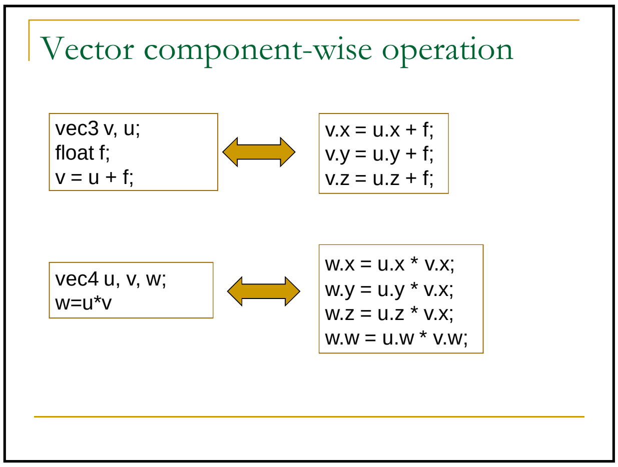
```
mat4 m = mat4(3.0); // initializes the diagonal to all 3.0
vec4 v;
v = m[1]; // places the vector (0.0, 3.0, 0.0, 0.0) into v
```



Vector swizzling operation

- SWIZZLE
  - To duplicate or switch around the order of the components of a vector. For example

```
vec4 v4;
v4.rgba; // is a vec4 and the same as just using v4,
v4.rrb; // is a vec3,
v4.b; // is a float,
v4.xy; // is a vec2,
v4.xgba; // is illegal – mixed use of xyzw and rgba
```



Vector component-wise operation

```
vec3 v, u;
float f;
v = u + f;
```

↔

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
```

```
vec4 u, v, w;
w=u*v
```

↔

```
w.x = u.x * v.x;
w.y = u.y * v.x;
w.z = u.z * v.x;
w.w = u.w * v.w;
```

Matrix indexing operation:

```
mat4 m;

m[0]=vec4(1.0, 2.0, 0.0, 0.0); //sets the first column

m[1]=vec4(0.0); //sets the second column to all 0.0

M[3][2]=1.0; //sets third element of the fourth column to 1.0;
```

Matrix operation

```
mat4 m, m1;
vec4 v;

v * m; // This is a linear-algebraic row-vector
       // by matrix multiplication
m * v; // This is a linear-algebraic matrix by
       // column-vector multiplication
m * m1; // This is a linear-algebraic matrix
        // by matrix multiplication
```

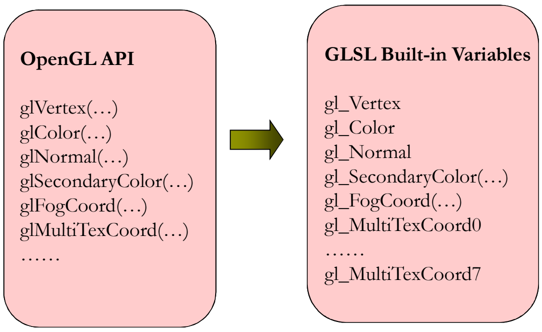
GLSL Variables

- GLSL built-in variables
- User defined variables

GLSL Built-in Variables

- Unlike ‘C’, there are many GLSL **built-in** variables to **communicate** data between shaders and OpenGL fixed functionality
  - Some OpenGL fixed operations are still required in programmable graphics pipeline, such as operations occurred
    - between vertex processor and fragment processors
    - and after the fragment processor

Mapping between OpenGL API and GLSL



OpenGL API

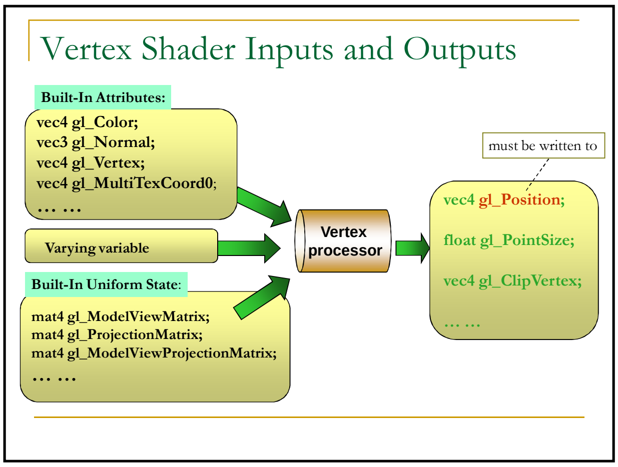
glVertex(…)
glColor(…)
glNormal(…)
glSecondaryColor(…)
glFogCoord(…)
glMultiTexCoord(…)
……

GLSL Built-in Variables

gl_Vertex
gl_Color
gl_Normal
gl_SecondaryColor(…)
gl_FogCoord(…)
gl_MultiTexCoord0
……
gl_MultiTexCoord7

Frequently used GLSL Built-in States

- As an aid to accessing OpenGL processing state
  - gl_ModelViewMatrix;
  - gl_ProjectionMatrix;
  - gl_ModelViewProjectionMatrix;
  - gl_NormalMatrix;
  - gl_ModelViewMatrixInverse;
  - … …



Vertex Shader Inputs and Outputs

Built-In Attributes:

vec4 gl_Color;
vec3 gl_Normal;
vec4 gl_Vertex;
vec4 gl_MultiTexCoord0;
… …

Varying variable

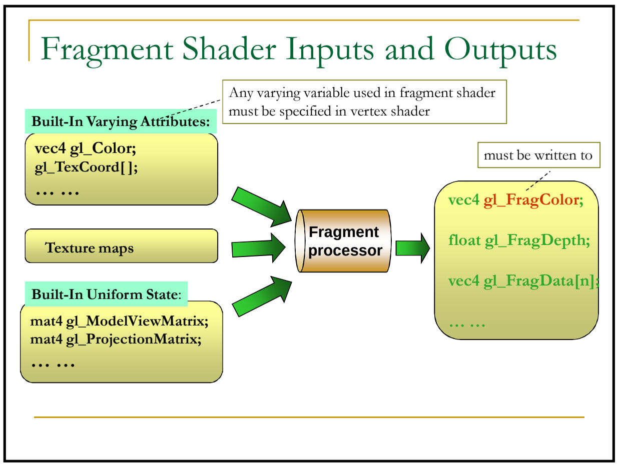
Built-In Uniform State:

mat4 gl_ModelViewMatrix;
mat4 gl_ProjectionMatrix;
mat4 gl_ModelViewProjectionMatrix;
… …

Vertex processor

must be written to

vec4 gl_Position;
float gl_PointSize;
vec4 gl_ClipVertex;
… …



Fragment Shader Inputs and Outputs

Any varying variable used in fragment shader must be specified in vertex shader

Built-In Varying Attributes:

vec4 gl_Color;
gl_TexCoord[ ];
… …

Texture maps

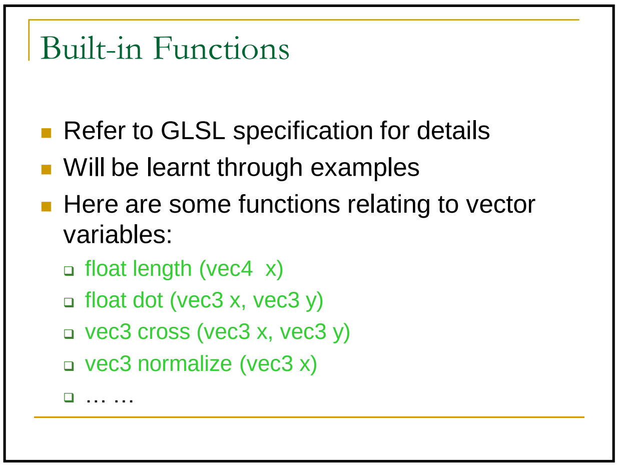
Built-In Uniform State:

mat4 gl_ModelViewMatrix;
mat4 gl_ProjectionMatrix;
… …

Fragment processor

must be written to

vec4 gl_FragColor;
float gl_FragDepth;
vec4 gl_FragData[n];
… …



Built-in Functions

- Refer to GLSL specification for details
- Will be learnt through examples
- Here are some functions relating to vector variables:
  - float length (vec4 x)
  - float dot (vec3 x, vec3 y)
  - vec3 cross (vec3 x, vec3 y)
  - vec3 normalize (vec3 x)
  - … …