GLSL Programming/Vector and Matrix Operations

The syntax of GLSL is very similar to C (and therefore to C++ and Java); however, there are built-in data types and functions for floating-point vectors and matrices, which are specific to GLSL. These are discussed here. A full description of GLSL can be found in the literature in the "Further Reading" section.

Data Types

In GLSL, the types `vec2`, `vec3`, and `vec4` represent 2D, 3D, and 4D floating-point vectors. (There are also types for integer and boolean vectors, which are not discussed here.) Vector variables are defined as you would expect if C, C++ or Java had these types:

```
vec2 a2DVector;
vec3 three_dimensional_vector;
vec4 vector4;
```

The data types for floating-point 2×2, 3×3, and 4×4 matrices are: `mat2`, `mat3`, and `mat4`:

```
mat2 m2x2;
mat3 linear_mapping;
mat4 trafo;
```

Precision Qualifiers

When declaring a floating-point variable (including vector and matrix variables), you can suggest a precision by using one of the precision qualifiers `lowp`, `mediump`, or `highp`, for example:

```
lowp vec4 color; // for colors, lowp is usually fine
mediump vec4 position; // for positions and texture coordinates, mediump is usually ok
highp vec4 astronomical_position; // for some positions and time, highp is necessary
//(precision of time measurement decreases over time
//and things get jumpy if they rely on absolute time since start of the application)
```

The idea is that `lowp` variables require less computation time and less storage (and therefore also less bandwidth) than `mediump` variables, and `mediump` variables require less time than `highp` variables. Thus, for best performance you should use the lowest precision that still gives satisfactory results. (There are only a few exceptions to this, for example accessing individual elements of a `lowp` vector can be slower than for a `mediump` vector because it requires additional decoding.)

Note that it is up to the compiler and driver to decide which precision is actually used; the precision qualifiers are only suggestions by the programmer. Also note that the range of `lowp` variables can be very limited (e.g. from -2.0 to 2.0), thus, `lowp` is most useful for colors but usually not for positions.

To define default precisions for all floating-point variables, you should use the `precision` command, for example:

```
precision mediump float;
```

This will use `mediump` if no explicit precision qualifier is specified for a floating-point variable. Without the command, the default precision for float is `highp` in an OpenGL or OpenGL ES vertex shader, `highp` in an OpenGL fragment shader, and undefined in an OpenGL ES fragment shader. (In Unity, however, the `precision` command is not available and the default precision is `highp` in a vertex shader and `mediump` in a fragment shader.)

Constructors

Vectors can be initialized and converted by constructors of the same name as the data type:

```
vec2 a = vec2(1.0, 2.0);
vec3 b = vec3(-1.0, 0.0, 0.0);
vec4 c = vec4(0.0, 0.0, 0.0, 1.0);
```

Note that some GLSL compilers will complain if integers are used to initialize floating-point vectors; thus, it is good practice to always include the decimal point.

One can also use one floating-point number in the constructor to set all components to the same value:

```
vec4 a = vec4(0.0); // = vec4(0.0, 0.0, 0.0, 0.0)
```

Casting a higher-dimensional vector to a lower-dimensional vector is also achieved with these constructors:

```
vec4 a = vec4(-1.0, 2.5, 4.0, 1.0);
vec3 b = vec3(a); // = vec3(-1.0, 2.5, 4.0)
vec2 c = vec2(b); // = vec2(-1.0, 2.5)
```

Casting a lower-dimensional vector to a higher-dimensional vector is achieved by supplying these constructors with the correct number of components:

```
vec2 a = vec2(0.1, 0.2);
vec3 b = vec3(0.0, a); // = vec3(0.0, 0.1, 0.2)
vec4 c = vec4(b, 1.0); // = vec4(0.0, 0.1, 0.2, 1.0)
```

Similarly, matrices can be initialized and constructed. Note that the values specified in a matrix constructor are consumed to fill the first column, then the second column, etc.:

```
mat3 m = mat3(
   1.1, 2.1, 3.1, // first column (not row!)
   1.2, 2.2, 3.2, // second column
   1.3, 2.3, 3.3  // third column
);
mat3 id = mat3(1.0); // puts 1.0 on the diagonal
                     // all other components are 0.0
vec3 column0 = vec3(0.0, 1.0, 0.0);
vec3 column1 = vec3(1.0, 0.0, 0.0);
vec3 column2 = vec3(0.0, 0.0, 1.0);
mat3 n = mat3(column0, column1, column2); // sets columns of matrix n
```

If a larger matrix is constructed from a smaller matrix, the additional rows and columns are set to the values they would have in an identity matrix:

```
mat2 m2x2 = mat2(
   1.1, 2.1,
   1.2, 2.2
);
mat3 m3x3 = mat3(m2x2); // = mat3(
   // 1.1, 2.1, 0.0,
   // 1.2, 2.2, 0.0,
   // 0.0, 0.0, 1.0)
mat2 mm2x2 = mat2(m3x3); // = m2x2
```

If a smaller matrix is constructed from a larger matrix, the top, left submatrix of the larger matrix is chosen, e.g. in the last line of the previous example.

Components

Components of vectors are accessed by array indexing with the `[]`-operator (indexing starts with 0) or with the `.`-operator and the element names `x, y, z, w` or `r, g, b, a` or `s, t, p, q`:

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
float a = v[3]; // = 4.4
float b = v.w; // = 4.4
float c = v.a; // = 4.4
float d = v.q; // = 4.4
```

It is also possible to construct new vectors by extending the `.`-notation ("swizzling"):

```
vec4 v = vec4(1.1, 2.2, 3.3, 4.4);
vec3 a = v.xyz; // = vec3(1.1, 2.2, 3.3)
vec3 b = v.bgr; // = vec3(3.3, 2.2, 1.1)
vec2 c = v.tt; // = vec2(2.2, 2.2)
```

Matrices are considered to consist of column vectors, which are accessed by array indexing with the `[]`-operator. Elements of the resulting (column) vector can be accessed as discussed above:

```
mat3 m = mat3(
   1.1, 2.1, 3.1, // first column
   1.2, 2.2, 3.2, // second column
   1.3, 2.3, 3.3  // third column
);
vec3 column3 = m[2]; // = vec3(1.3, 2.3, 3.3)
float m20 = m[2][0]; // = 1.3
float m21 = m[2].y; // = 2.3
```

Operators

If the binary operators `*, /, +, -, =, *=, /=, +=, -=` are used between vectors of the same type, they just 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)
```

Note in particular that `a * b` represents a component-wise product of two vectors, which is not often seen in linear algebra.

For matrices, these operators also work component-wise, **except** for the `*`-operator, which represents a matrix-matrix product, e.g.:

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

And in GLSL:

```
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.)
```

For a component-wise matrix product, the built-in function `matrixCompMult` is provided.

The `*`-operator can also be used to multiply a floating-point value (i.e. a scalar) to all components of a vector or matrix (from left or right):

```
vec3 a = vec3(1.0, 2.0, 3.0);
mat3 m = mat3(1.0);
float s = 10.0;
vec3 b = s * a; // vec3(10.0, 20.0, 30.0)
vec3 c = a * s; // vec3(10.0, 20.0, 30.0)
mat3 m2 = s * m; // = mat3(10.0)
mat3 m3 = m * s; // = mat3(10.0)
```

Furthermore, the `*`-operator can be used for matrix-vector products of the corresponding dimension, e.g.:

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

And in GLSL:

```
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.)
```

Note that the vector has to be multiplied to the matrix from the right.

If a vector is multiplied to a matrix **from the left**, the result corresponds to multiplying a row vector from the left to the matrix. This corresponds to multiplying a column vector to the **transposed** matrix from the right:

$$\mathbf{v}^T\mathbf{M} = \left(\mathbf{M}^T\mathbf{v}\right)^T$$

In components:

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

$$= \left(\begin{bmatrix} m_{1,1} & m_{2,1} \\ m_{1,2} & m_{2,2} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \right)^T = \left(\mathrm{M}^T\mathbf{v}\right)^T$$

Thus, multiplying a vector from the left to a matrix corresponds to multiplying it from the right to the transposed matrix:

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

Since there is no built-in function to compute a transposed matrix, this technique is extremely useful: whenever a vector should be multiplied with a transposed matrix, one can just multiply it from the left to the original matrix. Several applications of this technique are described in Section “Applying Matrix Transformations”.

Built-In Vector and Matrix Functions

Component-Wise Functions

As mentioned, the function

```
TYPE matrixCompMult(TYPE a, TYPE b) // component-wise matrix product
```

computes a component-wise product for the matrix types `mat2`, `mat3` and `mat4`, which are denoted as `TYPE`.

The following functions work component-wise for variables of type `float`, `vec2`, `vec3` and `vec4`, which are denoted as `TYPE`:

```
TYPE min(TYPE a, TYPE b) // returns a if a < b, b otherwise
TYPE min(TYPE a, float b) // returns a if a < b, b otherwise
TYPE max(TYPE a, TYPE b) // returns a if a > b, b otherwise
TYPE max(TYPE a, float b) // returns a if a > b, b otherwise
TYPE clamp(TYPE a, TYPE minVal, TYPE maxVal)
   // = min(max(a, minVal), maxVal)
TYPE clamp(TYPE a, float minVal, float maxVal)
   // = min(max(a, minVal), maxVal)
TYPE mix(TYPE a, TYPE b, TYPE wb) // = a * (TYPE(1.0) - wb) + b * wb
TYPE mix(TYPE a, TYPE b, float wb) // = a * TYPE(1.0 - wb) + b * TYPE(wb)
```

There are more built-in functions, which also work component-wise but are less useful for vectors, e.g., `abs`, `sign`, `floor`, `ceil`, `fract`, `mod`, `step`, `smoothstep`, `sqrt`, `inversesqrt`, `pow`, `exp`, `exp2`, `log`, `log2`, `radians` (converts degrees to radians), `degrees` (converts radians to degrees), `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (with one argument and with two arguments for signed numerator and signed denominator), `lessThan`, `lessThanEqual`, `greaterThan`, `greaterThanEqual`, `equal`, `notEqual`, and `not`.

Geometric Functions

The following functions are particular useful for vector operations. `TYPE` is any of: `float`, `vec2`, `vec3` and `vec4` (only one of them per line).

```
vec3 cross(vec3 a, vec3 b) // = vec3(a[1] * b[2] - a[2] * b[1],
   // a[2] * b[0] - a[0] * b[2],
   // a[0] * b[1] - a[1] * b[0])
float dot(TYPE a, TYPE b) // = a[0] * b[0] + a[1] * b[1] + ...
float length(TYPE a) // = sqrt(dot(a, a))
float distance(TYPE a, TYPE b) // = length(a - b)
TYPE normalize(TYPE a) // = a / length(a)
TYPE faceforward(TYPE n, TYPE i, TYPE nRef)
   // returns n if dot(nRef, i) < 0, -n otherwise
TYPE reflect(TYPE i, TYPE n) // = i - 2. * dot(n, i) * n
   // this computes the reflection of vector 'i'
   // at a plane of normalized(!) normal vector 'n'
```

Functions for Physics

The function

```
TYPE refract(TYPE i, TYPE n, float r)
```

computes the direction of a refracted ray if `i` specifies the normalized(!) direction of the incoming ray and `n` specifies the normalized(!) normal vector of the interface of two optical media (e.g. air and water). The vector `n` should point to the side from where `i` is coming, i.e. the dot product of `n` and `i` should be negative. The floating-point number `r` is the ratio of the refractive index of the medium from where the ray comes to the refractive index of the medium on the other side of the surface. Thus, if a ray comes from air (refractive index about 1.0) and hits the surface of water (refractive index 1.33), then the ratio `r` is 1.0 / 1.33 = 0.75. The computation of the function is:

```
float d = 1.0 - r * r * (1.0 - dot(n, i) * dot(n, i));
if (d < 0.0) return TYPE(0.0); // total internal reflection
return r * i - (r * dot(n, i) + sqrt(d)) * n;
```

As the code shows, the function returns a vector of length 0 in the case of total internal reflection (see the entry in Wikipedia), i.e. if the ray does not pass the interface between the two materials.

Further Reading

All details of GLSL for OpenGL are specified in the “OpenGL Shading Language 4.10.6 Specification” available at the Khronos OpenGL web site (http://www.khronos.org/opengl/).

More accessible descriptions of the OpenGL shading language for OpenGL are given in recent editions of many books about OpenGL, e.g. in Chapter 15 of the “OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1” (7th ed.) by Dave Shreiner published 2009 by Addison-Wesley.

All details of GLSL for OpenGL ES 2.0 are specified in the “OpenGL ES Shading Language 1.0.17 Specification” available at the “Khronos OpenGL ES API Registry” (http://www.khronos.org/registry/gles/).

A more accessible description of the OpenGL ES shading language is given in Chapter 5 and Appendix B of the book “OpenGL ES 2.0 Programming Guide” by Aaftab Munshi, Dan Ginsburg and Dave Shreiner published by Addison-Wesley (see its web site (http://www.opengles-book.com/)).