GLSL Programming/Applying Matrix Transformations

Applying the conventional vertex transformations (see Section “Vertex Transformations”) or any other transformations that are represented by matrices in shaders is usually accomplished by specifying the corresponding matrix in a uniform variable of the shader and then multiplying the matrix with a vector. There are, however, some differences in the details. Here, we discuss the transformation of points (i.e. 4D vectors with a 4th coordinates equal to 1), the transformation of directions (i.e. vectors in the strict sense: 3D vectors or 4D vectors with a 4th coordinate equal to 0), and the transformation of surface normal vectors (i.e. vectors that specify a direction that is orthogonal to a plane).

This section assumes some knowledge of the syntax of GLSL as described in Section “Vector and Matrix Operations”.

Transforming Points

For points, transformations are usually represented by 4×4 matrices since they might include a translation by a 3D vector $\mathbf{t}$ in the 4th column:

$$\mathrm{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with } \mathrm{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

(Projection matrices will also include additional values unequal to 0 in the last row.)

Three-dimensional points are represented by four-dimensional vectors with the 4th coordinate equal to 1:

$$P = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

In order to apply the transformation, the matrix M is multiplied with the vector P:

$$\mathrm{M}\,P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

The GLSL code to apply a 4×4 matrix to a point represented by a 4D vector is straightforward:

```
mat4 matrix;
vec4 point;
vec4 transformed_point = matrix * point;
```

Transforming Directions

Directions in three dimensions are represented either by a 3D vector or by a 4D vector with 0 as the fourth coordinate. (One can think of them as points at infinity; similar to a point at the horizon of which we cannot tell the position in space but only the direction in which to find it.)

In the case of a 3D vector, we can either transform it by multiplying it with a 3×3 matrix:

```
mat3 matrix;
vec3 direction;
vec3 transformed_direction = matrix * direction;
```

or with a 4×4 matrix, if we convert the 3D vector to a 4D vector with a 4th coordinate equal to 0:

```
mat4 matrix;
vec3 direction;
vec3 transformed_direction = vec3(matrix * vec4(direction, 0.0));
```

Alternatively, the 4×4 matrix can also be converted to a 3×3 matrix.

On the other hand, a 4D vector can be multiplied directly with a 4×4 matrix. It can also be converted to a 3D vector in order to multiply it with a 3×3 matrix:

```
mat3 matrix;
vec4 direction; // 4th component is 0
vec4 transformed_direction = vec4(matrix * vec3(direction), 0.0);
```

Transforming Normal Vectors

Similarly to directions, surface normal vectors (or “normal vectors” for short) are represented by 3D vectors or 4D vectors with 0 as the 4th component. However, they transform differently. (The mathematical reason is that they represent something that is called a covector, covariant vector, one-form, or linear functional.)

To understand the transformation of normal vectors, consider the main feature of a surface normal vector: it is orthogonal to a surface. Of course, this feature should still be true under transformations, i.e. the transformed normal vector should be orthogonal to the transformed surface. If the surface is being represented locally by a tangent vector, this feature requires that a transformed normal vector is orthogonal to a transformed direction vector if the original normal vector is orthogonal to the original direction vector.

Mathematically spoken, a normal vector $\mathbf{n}$ is orthogonal to a direction vector $\mathbf{v}$ if their dot product is 0. It turns out that if $\mathbf{v}$ is transformed by a 3×3 matrix $\mathbf{A}$, the normal vector has to be transformed by the **transposed inverse** of $\mathbf{A}$: $(\mathbf{A}^{-1})^T$. We can easily test this by checking the dot product of the transformed normal vector $(\mathbf{A}^{-1})^T\mathbf{n}$ and the transformed direction vector $\mathbf{A}\mathbf{v}$:

$$(\mathbf{A}^{-1})^T\mathbf{n} \cdot \mathbf{A}\mathbf{v} = \left((\mathbf{A}^{-1})^T\mathbf{n}\right)^T\mathbf{A}\mathbf{v} = \left(\mathbf{n}^T\left((\mathbf{A}^{-1})^T\right)^T\right)\mathbf{A}\mathbf{v} = (\mathbf{n}^T\mathbf{A}^{-1})\,\mathbf{A}\mathbf{v} = \mathbf{n}^T\mathbf{A}^{-1}\mathbf{A}\mathbf{v} = \mathbf{n}^T\mathbf{v} = \mathbf{n} \cdot \mathbf{v}$$

In the first step we have used $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b}$, then $(M\mathbf{a})^T = \mathbf{a}^T M^T$, then $(\mathbf{M}^T)^T = \mathbf{M}$, then $\mathbf{M}^{-1}\mathbf{M} = \mathbf{Id}$ (i.e. the identity matrix).

The calculation shows that the dot product of the transformed vectors is in fact the same as the dot product of the original vectors; thus, the transformed vectors are orthogonal if and only if the original vectors are orthogonal. Just the way it should be.

Thus, in order to transform normal vectors in GLSL, the **transposed inverse matrix** is often specified as a uniform variable (together with the original matrix for the transformation of directions and points) and applied as any other transformation:

```
mat3 matrix_inverse_transpose;
vec3 normal;
vec3 transformed_normal = matrix_inverse_transpose * normal;
```

In the case of a 4×4 matrix, the normal vector can be cast to a 4D vector by appending 0:

```
mat4 matrix_inverse_transpose;
vec3 normal;
vec3 transformed_normal = vec3(matrix_inverse_transpose * vec4(normal, 0.0));
```

Alternatively, the matrix can be cast to a 3×3 matrix.

If the inverse matrix is known, the normal vector can be multiplied from the left to apply the transposed inverse matrix. In general, multiplying a transposed matrix with a vector can be easily expressed by putting the vector to the left of the matrix. The reason is that a vector-matrix product makes only sense for row vectors (i.e. transposed column vectors) and corresponds to a matrix-vector product of the transposed matrix with the corresponding column vector:

$$\mathbf{p}^T\mathbf{A} = \left(\mathbf{A}^T\mathbf{p}\right)^T$$

Since GLSL makes no distinction between column and row vectors, the result is just a vector.

Thus, in order to multiply a normal vector with the transposed inverse matrix, we can multiply it from the left to the inverse matrix:

```
mat3 matrix_inverse;
vec3 normal;
vec3 transformed_normal = normal * matrix_inverse;
```

In the case of multiplying a 4×4 matrix to a 4D normal vector (from the left or the right), it should be made sure that the 4th component of the resulting vector is 0. In fact, in several cases it is necessary to discard the computed 4th component (for example by casting the result to a 3D vector):

```
mat4 matrix_inverse;
vec4 normal;
vec4 transformed_normal = vec4(vec3(normal * matrix_inverse), 0.0);
```

Note that any normalization of the normal vector to unit length is not preserved by this transformation. Thus, normal vectors are often normalized to unit length after the transformation (e.g. with the built-in GLSL function `normalize`).

Transforming Normal Vectors with an Orthogonal Matrix

A special case arises when the transformation matrix $\mathbf{A}$ is orthogonal. In this case, the inverse of $\mathbf{A}$ is the transposed matrix; thus, the transposed of the inverse of $\mathbf{A}$ is the twice transposed matrix, which is the original matrix, i.e. for a orthogonal matrix $\mathbf{A}$:

$$\left(\mathbf{A}^{-1}\right)^T = \left(\mathbf{A}^T\right)^T = \mathbf{A}$$

Thus, in the case of **orthogonal** matrices, normal vectors are transformed with the same matrix as directions and points:

```
mat3 matrix; // orthogonal matrix
vec3 normal;
vec3 transformed_normal = matrix * normal;
```

Transforming Points with the Inverse Matrix

Sometimes it is necessary to apply the inverse transformation. In most cases, the best solution is to define another uniform variable for the inverse matrix and set the inverse matrix in the main application. The shader can then apply the inverse matrix like any other matrix. This is by far more efficient than computing the inverse in the shader.

There is, however, a special case: If the matrix $\mathbf{M}$ is of the form presented above (i.e. the 4th row is (0,0,0,1)):

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with an orthogonal 3×3 matrix $\mathbf{A}$ (i.e. the row (or column) vectors of $\mathbf{A}$ are normalized and orthogonal to each other; for example, this is usually the case for the view transformation, see Section “Vertex Transformations”), then the inverse matrix is given by (because $\mathbf{A}^{-1} = \mathbf{A}^T$ for an orthogonal matrix $\mathbf{A}$):

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T & -\mathbf{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

For the multiplication with a point P that is represented by the 4D vector $(p_x, p_y, p_z, 1)$ with the 3D vector $\mathbf{p} = (p_x, p_y, p_z)$ we get:

$$\mathbf{M}^{-1}P = \begin{bmatrix} \mathbf{A}^T & -\mathbf{A}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T\mathbf{p} - \mathbf{A}^T\mathbf{t} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}^T(\mathbf{p} - \mathbf{t}) \\ 1 \end{bmatrix}$$

Note that the vector $\mathbf{t}$ is just the 4th column of the matrix $\mathbf{M}$, which can be conveniently accessed in GLSL:

```
mat4 matrix;
vec4 last_column = matrix[3]; // indices start with 0 in GLSL
```

As mentioned above, multiplying a transposed matrix with a vector can be easily expressed by putting the vector to the left of the matrix because a vector-matrix product makes only sense for row vectors (i.e. transposed column vectors) and corresponds to a matrix-vector product of the transposed matrix with the corresponding column vector:

$$\mathbf{p}^T\mathbf{A} = \left(\mathbf{A}^T\mathbf{p}\right)^T$$

Using these features of GLSL, the term $\boldsymbol{A}^T(\mathbf{p} - \mathbf{t})$ is easily and efficiently implemented as follows (note that the 4th component of the result has to be set to 1 separately):

```
mat4 matrix; // upper, left 3x3 matrix is orthogonal;
   // 4th row is (0,0,0,1)
vec4 point; // 4th component is 1
vec4 point_transformed_with_inverse = vec4(vec3((point - matrix[3]) * matrix), 1.0);
```

Transforming Directions with the Inverse Matrix

As in the case of points, the best way to transform a direction with the inverse matrix is usually to compute the inverse matrix in the main application and communicate it to a shader via another uniform variable.

The exception is an orthogonal 3×3 matrix $\mathbf{A}$ (i.e. all rows (or columns) are normalized and orthogonal to each other) or a 4×4 matrix $\mathbf{M}$ of the form

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{A}$ is an orthogonal 3×3 matrix. In these cases, the inverse matrix $\mathbf{A}^{-1}$ is equal to the transposed matrix $\mathbf{A}^T$.

As discussed above, the best way in GLSL to multiply a vector with the transposed matrix is to multiply it from the left to the original matrix because this is interpreted as a product of a row vector with the original matrix, which corresponds to the product of the transposed matrix with the column vector:

$$\mathbf{p}^T\mathbf{A} = \left(\mathbf{A}^T\mathbf{p}\right)^T$$

Thus, the transformation with the transposed matrix (i.e. the inverse in case of a orthogonal matrix) is written as:

```
mat4 matrix; // upper, left 3x3 matrix is orthogonal
vec4 direction; // 4th component is 0
vec4 direction_transformed_with_inverse = vec4(vec3(direction * matrix), 0.0);
```

Note that the 4th component of the result has to be set to 0 separately since the 4th component of `direction * matrix` is not meaningful for the transformation of directions. (It is, however, meaningful for the transformation of plane equations, which are not discussed here.)

The versions for 3x3 matrices and 3D vectors only require different cast operations between 3D and 4D vectors.

Transforming Normal Vectors with the Inverse Transformation

Suppose the inverse matrix $\mathbf{M}^{-1}$ is available, but the transformation corresponding to $\mathbf{M}$ is required. Moreover, we want to apply this transformation to a normal vector. In this case, we can just apply the transpose of the inverse by multiplying the normal vector from the left to the inverse matrix (as discussed above):

```
mat4 matrix_inverse;
vec3 normal;
vec3 transformed_normal = vec3(vec4(normal, 0.0) * matrix_inverse);
```

(or by casting the matrix).

Built-In Matrix Transformations

Some frameworks (in particular the OpenGL compatibility profile but neither the OpenGL core profile nor OpenGL ES 2.x) provide several built-in uniforms to access certain vertex transformations in GLSL shaders. They should not be declared, but here are the declarations to specify their types:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix; // transpose of the inverse of the
   // upper left 3x3 matrix of gl_ModelViewMatrix
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
uniform mat4 gl_TextureMatrixTranspose[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixInverseTranspose;
uniform mat4 gl_ProjectionMatrixInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverseTranspose;
uniform mat4 gl_TextureMatrixInverseTranspose[gl_MaxTextureCoords];
```

Further Reading

The transformation of normal vectors is described in Section 2.12.2 of the “OpenGL 4.1 Compatibility Profile Specification” available at the Khronos OpenGL web site (http://www.khronos.org/opengl/).

A more accessible description of the transformation of normal vectors is given in Appendix E of the free HTML version of the “OpenGL Programming Guide” available online (http://www.glprogramming.com/red/appendixe.html).

The built-in uniforms of matrix transformations are described in Section 7.4.1 of the “OpenGL Shading Language 4.10.6 Specification” available at the Khronos OpenGL web site (http://www.khronos.org/opengl/).

< GLSL Programming