# Matrix Conventions In Computer Graphics

## Prefix versus Postfix conventions

There are two conventions used by mathematicians to compose functions (you might have learned this in your math courses): *prefix* and *postfix*. Note that these are conventions that are used to both think about and represent functions – the final result using either convention should give the same answer. In general, $\mathrm{OpenGL}$, $\mathrm{GLSL}$, and $\mathrm{GLM}$ can be thought of as using prefix convention while $\mathrm{DirectX}$ uses the postfix convention.

Prefix convention is more natural. When we want to apply a function $f$ on some domain $x$, we write $f(x)$. If we want to apply a function $g$ on the domain $f(x)$, we write $g(f(x))$ or $(gf)(x)$. This is how we treat the composition of mappings or transforms in class lectures and in $\mathrm{OpenGL}$ code - that is, first apply transformation matrix $f$ and then apply transformation matrix $g$. So, if we use a prefix convention, points and vectors must be manifested as column matrices to satisfy the rules of matrix multiplication. In class lectures, notes, assignments, and in $\mathrm{OpenGL}$ code, we follow the prefix convention.

Consider the following code fragment that performs some matrix computations:

```
// compute model-to-world-to-view-clip transformation matrix ...

// first, compute perspective transform matrix:
glm::mat4 proj_mtx = glm::perspective(glm::radians(45.f),
                                      (GLfloat)fb_width / fb_height,
                                      1.f, 200.f);
// compute view transform matrix:
glm::mat4 view_mtx = glm::lookAt(glm::vec3(0.f, 0.f, 10.f),
                                 glm::vec3(0.f, 0.f, 0.f),
                                 glm::vec3(0.f, 1.f, 0.f));

// compute model transform matrix:
// scale transform followed by rotation followed by translation
glm::mat4 trans_mtx = glm::translate(glm::mat4(1.f), glm::vec3(1.f));
glm::mat4 rot_mtx = glm::rotate(glm::mat4(1.f),
                                glm::radians(s_angular_displacement),
                                glm::vec3(0.f, 0.f, 1.f));
glm::mat4 scl_mtx = glm::scale(glm::mat4(1.f), glm::vec3(2.f));

// now, compute model-to-world-to-view-clip transformation matrix ...
glm::mat4 mvp_mtx = proj_mtx * view_mtx * trans_mtx * rot_mtx * scl_mtx;
```

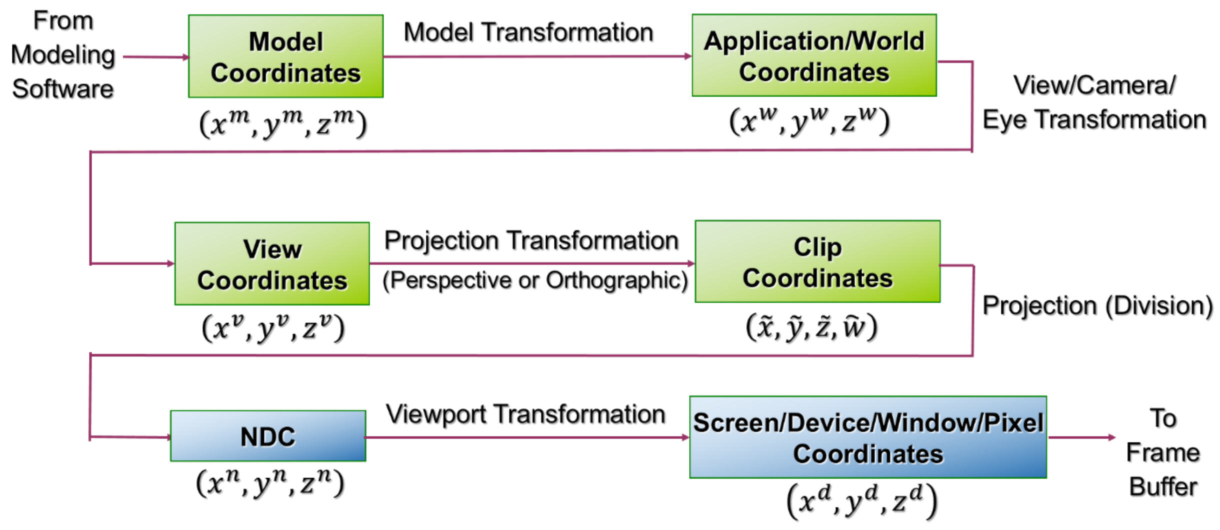
Five matrices are used in this code fragment: a perspective transformation, a view transformation, a translation transformation, a rotation transformation, and a scale transformation. The prefix convention is being used here to compute the final composite matrix. If $P^m$ represents a point in the model coordinate system and $\widetilde{P}$ represents a point in the clip coordinate system, matrices are being applied on point $P^m$ in the following prefix order:

$$\mathrm{M}_{perspective}\mathrm{M}_{view}\mathrm{M}_{translation}\mathrm{M}_{rotation}\mathrm{M}_{scale}P^m = \widetilde{P}$$

Since points are represented as column matrices, the following matrix multiplication holds:

$$\mathrm{M}_{perspective}\mathrm{M}_{view}\mathrm{M}_{translation}\mathrm{M}_{rotation}\mathrm{M}_{scale}\begin{bmatrix}x^m\\y^m\\z^m\\1\end{bmatrix}=\begin{bmatrix}\tilde{x}\\\tilde{y}\\\tilde{z}\\\tilde{w}\end{bmatrix}$$

First, model transformation consisting of scale, rotation, and translation (in that order), followed by the view and perspective transformations are being applied. This is borne out by Figure $1$ which illustrates the passage of a vertex through the graphics pipeline:



The projection (division) and viewport transformation seen in Figure $1$ are not included in the code fragment.

The thing to remember about prefix convention is that you write down the matrices in the order in which they'll be applied from right-to-left and compute the composite matrix from left-to-right.

On the other hand, if you're using the postfix convention, then points and vectors are manifested as row matrices. The five matrices would then be applied on points using postfix convention:

$$[x^m \;\; y^m \;\; z^m \;\; 1]\mathrm{M}^{\mathrm{T}}_{scale}\mathrm{M}^{\mathrm{T}}_{rotation}\mathrm{M}^{\mathrm{T}}_{translation}\mathrm{M}^{\mathrm{T}}_{view}\mathrm{M}^{\mathrm{T}}_{perspective}=[\tilde{x} \;\; \tilde{y} \;\; \tilde{z} \;\; \tilde{w}]$$

We use the notation $\mathrm{M}^{\mathrm{T}}_{scale}$ to denote that this [postfix convention] matrix is the transpose of the scale matrix $\mathrm{M}_{scale}$ used in the prefix convention. The postfix convention is used by $\mathrm{DirectX}$. Note that the above picture can also represent the $\mathrm{DirectX}$ graphics pipeline – the representation of the matrices is what makes $\mathrm{DirectX}$ matrices different from $\mathrm{OpenGL}$, $\mathrm{GLSL}$, and $\mathrm{GLM}$ matrices.

## Row-Major or Column-Major layout of matrices

Graphics $\mathrm{APIs}$ represent $4 \times 4$ matrices as an array of $16$ elements. These $16$ values can be interpreted using either row-major or column-major layout. In row-major layout, all elements of the first row come before all elements of the second row, and so on, and, within each row, elements are ordered by column. In contrast, in column-major layout, all elements of the first column come before all elements of the second column, and so on, and, within each column, elements are ordered by row. Consider the following $\mathrm{C}/\mathrm{C}{+}{+}$ array of $16$ elements:

```
float mtx[16] = {
  m0,  m1,  m2,  m3,
  m4,  m5,  m6,  m7,
  m8,  m9,  m10, m11,
  m12, m13, m14, m15
};
```

In a graphics $\text{API}$ using *row-major* layout, the first set of four values `m0, m1, m2, m3` in array `mtx` represent the $1^{\text{st}}$ row of the corresponding matrix, the second set of four values `m4, m5, m6, m7` represent the $2^{\text{nd}}$ row of the corresponding matrix, and so on. That is, the graphics $\text{API}$ using row-major layout would interpret the $16$ values in array `mtx` as the following matrix:

$$\begin{bmatrix} m0 & m1 & m2 & m3 \\ m4 & m5 & m6 & m7 \\ m8 & m9 & m10 & m11 \\ m12 & m13 & m14 & m15 \end{bmatrix}$$

In a graphics $\text{API}$ using *column-major* layout, the first set of four values `m0, m1, m2, m3` in array `mtx` represent the $1^{\text{st}}$ column of the corresponding matrix, the second set of four values `m4, m5, m6, m7` represent the $2^{\text{nd}}$ column of the corresponding matrix, and so on. That is, the graphics $\text{API}$ using column-major layout would interpret the $16$ values in array `mtx` as the following matrix:

$$\begin{bmatrix} m0 & m4 & m8 & m12 \\ m1 & m5 & m9 & m13 \\ m2 & m6 & m10 & m14 \\ m3 & m7 & m11 & m15 \end{bmatrix}$$

## Matrices in OpenGL

In the $1980$s, $\text{Iris GL}$ [the predecessor of $\text{OpenGL}$] used *postfix* convention to represent matrix transformations with points and vectors represented as row matrices and transformation matrices represented using the *row-major* layout [probably because the graphics library was implemented in C]. This means the $16$ values in the following array:

```
GLfloat trans_mtx[16] = {
  1.f, 0.f, 0.f, 0.f,
  0.f, 1.f, 0.f, 0.f,
  0.f, 0.f, 1.f, 0.f,
  tx, ty, tz, 1.f
};
```

are interpreted by $\text{Iris GL}$ as the matrix:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ tx & ty & tz & 1.0 \end{bmatrix}$$

and a vertex $P(x, y, z)$ is represented as row matrix:

$$[x \quad y \quad z \quad 1.0]$$

The transformation of $P$ by the matrix represented by array `trans_mtx` can be described as:

$$[x \quad y \quad z \quad 1.0]\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 1.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 1.0 & 0.0\\ tx & ty & tz & 1.0\end{bmatrix} = [x+tx \quad y+ty \quad z+tz \quad 1.0]$$

$\mathrm{OpenGL}\ 1.0$ [released in the early $1990$s] was designed to use *prefix* convention and therefore represented points and vectors as column matrices. To enable $\mathrm{OpenGL}$ to be compatible with $\mathrm{Iris\ GL}$ data and programs, $\mathrm{OpenGL}$ designers were left with no choice but to use the column-major layout for interpreting the $16$ values in an array. This means that the $1^{\mathrm{st}}$ four numbers in `trans_mtx` are stored in the $1^{\mathrm{st}}$ column of a matrix, the $2^{\mathrm{nd}}$ set of four numbers are stored in the $2^{\mathrm{nd}}$ column, and so on. Hence array `trans_mtx` represents the matrix:

$$\begin{bmatrix}1.0 & 0.0 & 0.0 & tx\\ 0.0 & 1.0 & 0.0 & ty\\ 0.0 & 0.0 & 1.0 & tz\\ 0.0 & 0.0 & 0.0 & 1.0\end{bmatrix}$$

and a vertex $P(x, y, z)$ is represented as column matrix:

$$\begin{bmatrix}x\\ y\\ z\\ 1.0\end{bmatrix}$$

and the transformation of $P$ by matrix defined by array `trans_mtx` [in column-major format] is represented as:

$$\begin{bmatrix}1.0 & 0.0 & 0.0 & tx\\ 0.0 & 1.0 & 0.0 & ty\\ 0.0 & 0.0 & 1.0 & tz\\ 0.0 & 0.0 & 0.0 & 1.0\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1.0\end{bmatrix} = \begin{bmatrix}x+tx\\ y+ty\\ z+tz\\ 1.0\end{bmatrix}$$

Commands such as `glMultMatrix`, `glLoadMatrix`, `glOrtho`, and `glFrustum` are now deprecated from the core profile but are available for use in the compatibility profile. The `glMultMatrix` and `glLoadMatrix` commands take an array of $16$ values (more precisely, a pointer to the first element of an array of values of type `GLfloat`) and interpret these $16$ values using column-major layout.

Core profile OpenGL [or modern OpenGL] is not concerned with matrices except for the `glUniformMatrix*` commands. These commands allow the OpenGL application to transfer matrices to shaders *in either* column-major or row-major format.

## Vectors and Matrices in GLSL

A *basic type* is a type defined by a keyword in $\mathrm{GLSL}$. Basic types consist of many of the fundamental types present in $\mathrm{C}$ and $\mathrm{C++}$. In addition, many other basic types such as vectors and matrices have been added to expose the internal $\mathrm{GPU}$ architecture to common graphics operations. Before describing matrix basic types, the more fundamental basic data types are enumerated.

## `void`

Just as in $\mathrm{C}$ and $\mathrm{C++}$, $\mathrm{GLSL}$ uses the `void` type to indicate that functions will not return a value.

## Scalars

As the name implies, *scalar* types are used to represent single values. The following table describes the basic scalar types.

| **Type** | **Meaning** |
|---|---|
| `bool` | as in $\mathrm{C++}$ with literals `true` and `false` provided |
| `int` | $32-$bit signed integer in two's complement form |
| `uint` | $32-$bit unsigned integer |
| `float` | $32-$bit single-precision floating-point matching $\mathrm{IEEE754}$ specification |
| `double` | $64-$bit double-precision floating-point matching $\mathrm{IEEE\ 754}$ specification |

## Vectors

Vectors are built-in types and reflect the vector-processing capabilities of $\mathrm{GPU}$ hardware. Vectors with elements of type `double`, `float`, `int`, `uint`, and `bool` having two, three, or four components are available.

| **Type** | **Meaning** |
|---|---|
| `vec2` | two-component single-precision floating-point vector |
| `vec3` | three-component single-precision floating-point vector |
| `vec4` | four-component single-precision floating-point vector |
| `dvec2` | two-component double-precision floating-point vector |
| `dvec3` | three-component double-precision floating-point vector |
| `dvec4` | four-component double-precision floating-point vector |

`ivec2`, `ivec3`, and `ivec4` types specify two, three, and four component signed integer vectors, respectively. Types `uvec2`, `uvec3`, and `uvec4` represent two, three, and four component unsigned integer vectors, respectively. Finally, `bvec2`, `bvec3`, and `bvec4` types represent two, three, and four component Boolean vectors, respectively.

It is important for you to realize that vector calculations requiring multiple $\mathrm{CPU}$ operations are implemented as single atomic operations on $\mathrm{GPU}$s. Suppose you want to calculate the dot product of two three-component vectors on the $\mathrm{CPU}$. Typically, your code would declare a $\mathrm{C/C++}$ structure for a vector of `float`s:

```
struct Vec3 {
  float x, y, z;
};
```

The dot product of two vectors would require a sum of products that would be coded like this:

```
Vec3 u { 1.f, 2.f, 3.f };
Vec3 v { 1.1f, 2.2f, 3.3f };
float sop = u.x*v.x + u.y*v.y + u.z*v.z;
```

Calculating the dot product on the $\text{CPU}$ requires $3$ multiplications, $2$ additions, and $1$ assignment. In contrast, $\text{GLSL}$ can use the basic vector types in conjunction with $\text{GPU}$s that are optimized to perform vector operations, to execute the dot product operation as an atomic operation:

```
Vec3 u { 1.f, 2.f, 3.f };
Vec3 v { 1.1f, 2.2f, 3.3f };
float sop = dot(u, v);
```

In addition to dot products, computing cross products is commonplace in computer graphics:

```
// CPU code ...
Vec3 u { 1.f, 2.f, 3.f };
Vec3 v { 1.1f, 2.2f, 3.3f };
Vec3 w;
w.x = u.y*v.z - u.z*v.y;
w.y = u.z*v.x - u.x*v.z;
w.z = u.x*v.y - u.y*v.x;

// GLSL code ...
vec3 a = vec3(1.f, 2.f, 3.f);
vec3 b = vec3(1.f, 2.2f, 3.3f);
vec3 c = cross(a, b);
```

While the scalar operations on the $\text{CPU}$ require $6$ multiplications, $3$ subtractions, and $1$ assignment, $\text{GLSL}$ can reduce these scalar computations to a single, atomic $\text{GPU}$ operation.

## Matrices

$\text{GLSL}$ has built-in types for containing single- and double-precision floating-point numbers in square and non-square matrices up to $4 \times 4$ in size: $2 \times 2$, $2 \times 3$, $2 \times 4$, $3 \times 2$, $3 \times 3$, $3 \times 4$, $4 \times 2$, $4 \times 3$, and $4 \times 4$. In general, matrix types in $\text{GLSL}$ are defined to use *column-major format*. Matrix types beginning with `mat` have single-precision components while matrices beginning with `dmat` have double-precision components. The first number in the type is the number of columns, the second is the number of rows. If there is only one number, the matrix is square. The following table illustrates the full range of matrix data types:

| Type | Meaning |
|---|---|
| `mat2x2` | single-precision floating-point matrix with $2$ columns and $3$ rows |
| `mat2x3` | single-precision floating-point matrix with $2$ columns and $3$ rows |
| `mat2x4` | single-precision floating-point matrix with $2$ columns and $4$ rows |
| `mat3x2` | single-precision floating-point matrix with $3$ columns and $2$ rows |
| `mat3x3` | single-precision floating-point matrix with $3$ columns and $3$ rows |
| `mat3x4` | single-precision floating-point matrix with $3$ columns and $4$ rows |
| `mat4x2` | single-precision floating-point matrix with $3$ columns and $2$ rows |
| `mat4x3` | single-precision floating-point matrix with $4$ columns and $3$ rows |
| `mat4x4` | single-precision floating-point matrix with $4$ columns and $4$ rows |
| `mat2` | same as `mat2x2` |
| `mat3` | same as `mat3x3` |
| `mat4` | same as `mat4x4` |

The `dmat` types are similar to the `mat` types with the exception of having double-precision components.

## Matrix initialization

Initialization of matrix types is done with constructors. You would initialize the diagonal of a matrix with all other elements set to zero like this:

```
mat2 m0(1.0); // initialize 2 x 2 matrix as identity matrix
mat3 m1(2.2); // initialize diagonal with 2.2 with other elements set to 0
mat4 m2(1.0); // initialize 4 x 4 matrix as identity matrix
```

To initialize a matrix by specifying vectors or scalars, the components are assigned to the matrix elements in *column-major* order:

```
vec2 v02 = vec2(2.0, 3.0), v12 = vec2(4.0, 5.0);
mat2 m0(v02, v12); // initialize 2 x 2 matrix as [v02 v12]

vec3 v03 = vec3(2.0, 3.0, 4.0), v13 = vec3(5.0, 6.0, 7.0),
     v23 = vec3(8.0, 9.0, 10.0);
mat3 m1(v03, v13, v23); // initialize 3 x 3 matrix as [v03 v13 v23]

vec4 v04 = vec4(2.0, 3.0, 4.0, 5.0); v14 = vec4(6.0, 7.0, 8.0, 9.0);
vec4 v24 = vec4(10.0, 11.0, 12.0, 13.0); v34 = vec4(14.0, 15.0, 16.0, 17.0);
mat4 m2(v04, v14, v24, v34); // initialize 4 x 4 matrix as [v04 v14 v24 v34]
```

```

// initialize m3 with following column vectors
// (2.0, 3.0, 4.0, 5.0), (6.0, 7.0, 8.0, 9.0),
// (10.0, 11.0, 12.0, 13.0), (14.0, 15.0, 16.0, 17.0)
mat4 m3(2.0,  3.0,  4.0,  5.0,
        6.0,  7.0,  8.0,  9.0,
        10.0, 11.0, 12.0, 13.0,
        14.0, 15.0, 16.0, 17.0);
```

It is important for you to understand that initializing $\mathrm{GLSL}$ matrices is very different from $\mathrm{C/C{+}{+}}$. If you define and initialize a $2\mathrm{D}$ matrix of $16$ elements in $\mathrm{C/C{+}{+}}$, the elements are specified in row-major order. To create a $\mathrm{C/C{+}{+}}$ $2\mathrm{D}$ array similar to $\mathrm{GLSL}$ matrix `m3` [defined in the previous code fragment], you would define the matrix as:

```
// C/C++ two-dimensional array m4 defined as:
GLfloat m4[4][4] = {
  { 2.f, 6.f, 10.f, 14.f },
  { 3.f, 7.f, 11.f, 15.f },
  { 4.f, 8.f, 12.f, 16.f },
  { 5.f, 9.f, 13.f, 17.f }
};
// is equivalent to GLSL matrix object m4 initialized as:
// mat4 m3(2.0,  3.0,  4.0,  5.0,
//         6.0,  7.0,  8.0,  9.0,
//         10.0, 11.0, 12.0, 13.0,
//         14.0, 15.0, 16.0, 17.0);
```

## Matrix length

The column-major layout of $\mathrm{GLSL}$'s matrix type is further stressed by the `length()` method. When applied to objects of matrix type, the `length()` method returns the number of columns of the matrix:

```
mat23 m1;
int const L1 = m1.length(); // initializes L1 to 2
mat32 m2;
int const L2 = m2.length(); // initializes L2 to 3
```

## Matrix components

Just as with $\mathrm{C/C{+}{+}}$, the components of a matrix can be accessed using the subscript `[]` operator:

```
mat32 m;  // uninitialized 3 x 2 matrix with 3 columns and 2 rows
m[2][0] = 0.0; // sets upper-right element of matrix
m[2][1] = 1.5; // sets bottom-right element of matrix
```

However, there are important differences between $\mathrm{C/C{+}{+}}$ and $\mathrm{GLSL}$ syntax. Applying a single subscript to a $\mathrm{GLSL}$ matrix treats the matrix as an array of column vectors, and selects a single column, whose type is a vector of the same size as the row size of the matrix:

```
mat32 m;  // uninitialized 3 x 2 matrix with 3 columns and 2 rows
m[0] = vec2(2.0); // sets all 3 elements of first column to 2.0
m[1] = vec2(1.1, 2.2); // sets 3 elements of 2nd column to 1.1, 2.2, 1.0
```

In contrast to $\mathrm{GLSL}$ syntax, applying a single subscript to a $\mathrm{C/C{+}{+}}$ $2\mathrm{D}$ array gives you an array of row elements:

```
GLfloat m4[4][4] = {
  { 2.f, 6.f, 10.f, 14.f }, // row 1
  { 3.f, 7.f, 11.f, 15.f }, // row 2
  { 4.f, 8.f, 12.f, 16.f }, // row 3
  { 5.f, 9.f, 13.f, 17.f }  // row 4
};

GLfloat *row_ptr = m4[2]; // pointer to 1st element of row 3
row_ptr[3] = 21.f; // update value of last element of row 3
```

## Matrix multiplication

Whether prefix or postfix convention is used is dependent on whether the matrix is multiplied by a vector or whether the vector is multiplied by a matrix. Postfix convention is used when a matrix is multiplied by a vector and prefix convention when a vector is multiplied by a matrix. To avoid confusion, you need to consistently stick to one of the two conventions. If you are not sure which convention to pick, the prefix convention is recommended since it matches the convention used in class notes.

```
// postfix convention with u and v thought of as row matrices
vec3 u, v; // assume v is initialized
mat3 m;    // assume m is initialized
u = v * m;
// u = v * m is equivalent to
// u.x = dot(v, m[0])
// u.y = dot(v, m[1])
// u.z = ot(v, m[2])

// prefix convention with u and v thought of as column matrices
u = m * v;
// u = m * v is equivalent to
// u.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z
// u.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z
// u.z = m[0].z * v.x + m[1].y * v.y + m[2].z * v.z
```

## Matrices in GLM

As indicated here, $\mathrm{GLM}$ is based on $\mathrm{GLSL}$ and therefore uses *column-major layout*. This means that the following initialization of a matrix using $\mathrm{GLM}$ type `glm::mat4`:

```
glm::vec3 displacement(10.f, 20.f, 30.f);
glm::mat4 trans_mtx(glm::vec4(1.f, 0.f, 0.f, 0.f),
                    glm::vec4(0.f, 1.f, 0.f, 0.f),
                    glm::vec4(0.f, 0.f, 1.f, 0.f),
                    glm::vec4(displacement, 1.f));
```

represents a translation matrix in prefix convention:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 10.0 \\ 0.0 & 1.0 & 0.0 & 20.0 \\ 0.0 & 0.0 & 1.0 & 30.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

The orthographic projection transformation matrix for the $\mathrm{OpenGL}$ graphics pipe has the form:

$$\mathrm{M}_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $l$ and $r$ specify the coordinates for left and right vertical clipping planes; $b$ and $t$ specify coordinates for bottom and top horizontal clipping planes; $n$ and $f$ specify the distances to near and far depth clipping planes. You can use method `glm::ortho()` to define a matrix for an orthographic parallel viewing volume:

```
glm::mat4 ortho_mtx = glm::ortho(l, r, b, t, n, f);
```

Or, you could directly specify the orthographic projection matrix:

```
// don't forget that GLM uses column-major layout ...
glm::mat4
ortho_mtx(glm::vec4(2.f/(r-l),    0.f,          0.f,          0.f),
          glm::vec4(0.f,          2.f/(t-b),    0.f,          0.f),
          glm::vec4(0.f,          0.f,          -2.f/(f-n),   0.f),
          glm::vec4(-(r+l)/(r-l), -(t+b)/(t-b), -(f+n)/(f-n), 1.f));
```

## Final words

Matrices must be represented as $2\mathrm{D}$ arrays in a computer program. While the $\mathrm{C/C{+}{+}}$ programming languages represent a $2\mathrm{D}$ array using row-major order, there are other programming languages - $\mathrm{FORTRAN}$ comes to mind - that use column-major order. Since there are prefix and postfix conventions on paper and row- and column-major orders of representing matrices in memory, there are four possible combinations to represent mappings in a computer program. This document has explained the particular conventions used by $\mathrm{OpenGL}$, $\mathrm{GLSL}$, and $\mathrm{GLM}$. As users of these $\mathrm{API}$s, you should not be concerned with the programming language in which these $\mathrm{API}$s are authored.

Instead, you should always use the interfaces provided by these $\mathrm{API}$s to initialize, update, and use these matrices.