Conffetti Math Conventions

Marijn Tamis, Volkan Ilbeyli

February 22, 2018

Abstract

This document will describe the math conventions used in Confetti codebase - The Forge.

1 Matrices and Vectors

1.1 Matrix multiplication

All matrices are defined and used as column major matrices. This means that vectors (specifically those representing positions and directions) are column vectors. Transforming vector v with transformation matrix M is written as follows:

$$v' = Mv$$

$$\begin{bmatrix} m_{11}x + m_{12}y + m_{13}z \\ m_{21}x + m_{22}y + m_{23}z \\ m_{31}x + m_{32}y + m_{33}z \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

It follows from this convention and the associative property of matrices that the sequential transformations can be combined in a single matrix as follows:

$$(M_2 M_1)\, v = M_2\, (M_1 v)$$

Where v is first transformed by M_1 and then by M_2.

1.2 Matrix memory layout

The elements in the matrices are laid out in memory in column major order. Note that this is an implementation detail which is unrelated to the notational convention.[1] Figure 1 shows how a 4×4 matrix is laid out in continuous memory.

Also note that the constructor of `mat4`[2] takes 4 `vec4`'s which are the columns of the matrix. This makes it look like the matrix is written transposed in the code.

1.3 Vector Types

The math library used in The Forge codebase provides two types of vectors: `vec2/3/4` and `float2/3/4`.

- `vec2/3/4` is used for optimized math calculations and may contain padding. A `vec3` has a `w` component for alignment and optimization reasons, hence `sizeof(vec3)` will return 16 Bytes.
- `float2/3/4` guarantees struct sizes and is used for storing data. A `float3` will contain 3 floating-point elements and will be 12 Bytes in size.

[1] Although it is easier to utilize vector instructions when the conventions match.
[2] `mat4` is a `typedef` for the Sony Matrix4 class

Matrix

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

Memory Layout

	0x0	0x4	0x8	0xC
0x00	m_{11}	m_{21}	m_{31}	m_{41}
0x10	m_{12}	m_{22}	m_{32}	m_{42}
0x20	m_{13}	m_{23}	m_{33}	m_{43}
0x30	m_{14}	m_{24}	m_{34}	m_{44}

Figure 1: The memory layout of `mat4`

2 Coordinate System

2.1 World Space

The coordinate system for world space coordinates is left handed when the `mat4::perspective()` and `mat4::orthographic()` are used. Functions like `mat4::rotationX()` will generate a matrix that rotates vectors in clockwise direction when the rotation axis points towards the observer (as described by the left hand rule). The up vector is $\hat{y} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^\top$. $\hat{z} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^\top$ points away from the camera and $\hat{x} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^\top$ points towards the right.

2.2 Clip Space

The clip space is also left handed with $\hat{y}$ up, $\hat{x}$ right, and $\hat{z}$ away from the camera. The visible volume is defined by the following ranges:

$$\begin{aligned} -w < x < w \\ -w < y < w \\ 0 \leq z < w \end{aligned}$$

After clipping the perspective divide is done to arrive at normalized device coordinates. We currently use the Direct3D clip space when outputting vertex coordinates in the shader. Notice that after projection, the DirectX projection model maps the Z component into [0, 1] range, unlike the OpenGL projection model which maps the Z component into [-1, 1] range.

2.3 Screen space

Screen space coordinates (as passed to `BaseApp::onMouseMove()`) are left handed, $\hat{x}$ points towards the right and $\hat{y}$ points down. The origin (0, 0) is the upper left corner, (WindowWidth, WindowHeight) is the bottom right corner.

2.4 Examples

2.4.1 Model View Projection Matrix

The model view projection matrix M_{PVM} is defined as follows:

$$M_{\text{PVM}} = M_{\text{Projection}} M_{\text{View}} M_{\text{Model}}$$

```
// Example Application Code
const float aspectInverse = (float)gWindowHeight / (float)gWindowWidth;
const float fovh = gPi / 2.0f;

mat4 projMat = mat4::perspective(fovh, aspectInverse, 0.1f, 1000.0f);
mat4 viewMat = pCameraController->getViewMatrix();

gUniformData.mProjectView = projMat * viewMat;
```

```
// Example Shader Code
cbuffer uniformBlock : register(b0)
{
   matrix projView;
   matrix model[MAX_INSTANCES];
}

VSOutput VSMain(VSInput input, uint InstanceID : SV_InstanceID)
{
   VSOutput result;

   // Transform model-space vertices w/ Model(World)-View-Projection matrix
   matrix mPVM = mul(projView, model[InstanceID]);
   result.Position = mul(mPVM, input.Position);
}
```

2.4.2 View Matrix

The view matrix for the default example camera is constructed as follows:

$$M_{\text{View}} = M^{-1}_{\text{CamRotation}} M^{-1}_{\text{CamTranslation}}$$

Example Code:

```
mat4 Rotation = mat4::rotateXY(-wx,-wy);
mat4 Translation = mat4::translation(-camPos);
mat4 View = Rotation * Translation;
```

Note that we did not invert any matrix. The inverse of `rotateYX(wy,wx)` is `rotateXY(-wx,-wy)`, and the inverse of `mat4::translation(camPos)` is `mat4::translation(-camPos)`.

3 Using & Extending the Math Library

3.1 Project Structure

The Forge uses Modified Sony Math library, an open source math library from GitHub. This is a cleaned up version of the Sony Math Library, removing some legacy interfaces such as PS3. The library is located at `TheForge\Common_3\ThirdParty\OpenSource\ModifiedSonyMath\` folder and contains the math function declarations and definitions. The application code is not directly using this path. Instead, apps use the `OS\Math\MathTypes.h` which includes the open source library and handles the typedefs.

The structure of the library `TheForge\Common_3\ThirdParty\OpenSource\ModifiedSonyMath\` is as follows:

- `vectormath.hpp` - This is the main header file that chooses the **Scalar** (`scalar\vectormath.hpp`) or the **SSE** (`sse\vectormath.hpp`) implementation of the library based on the hardware support.
- `scalar|sse\vectormath.hpp` - Contains declarations for `Vector3`, `Vector4`, `Matrix3`, `Matrix4`, `Transform` and `Quaternion` structs. Definitions are in `vector.hpp`, `matrix.hpp` and `quaternion.hpp` files in both of the `scalar\` or `sse\` folders.
- `vec2d.hpp` - The original Sony Math library doesn't have support for `Vector2` and `Matrix2`. The author of ModifiedSonyMath added support for `Vector2` and we have added support for `Matrix2` in `vec2.hpp`.
- `common.hpp` - This is where the common helper/utility math functions live such as intersection helpers, noise generators, half-precision floating point struct, and various utility functions like lerp, clamp, round, etc.

3.2 Adding New Math Functionality

In order to have a smoother experience updating the open source math library in the future, we have marked the header files where we make additions to the open source library. For example, search for `#ConfettiMathExtensions`[3] tag in the code base to see how extensions are handled.

To keep the codebase maintainable and consistent, one should check if the required math functionality exists in the current math library (`TheForge\Common_3\ThirdParty\OpenSource\ModifiedSonyMath\`) before creating a new function or file. If the task at hand cannot be achieved with the existing functions and introducing new functions is absolutely necessary, then the code should be added to a proper place in the `ModifiedSonyMath` directory.

Please do not add any utility/helper functions/classes in the `Common_3\OS\Math` folder.

Note that the developer should also implement the SSE[4] version of the function that is being added[5].

[3] Make sure to select Entire Solution (Including External Items) as the 'Look In' search option.
[4] Intel's Guide for SSE Instructions: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE
[5] See the implementation of `mat4::perspective()` or take a look at `common.hpp` as an example of how to extend the library.