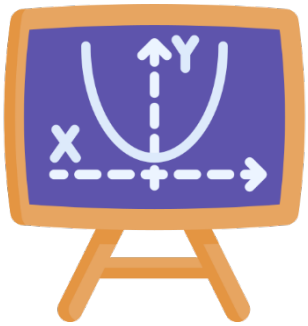
DirectX Math

A high-performance C++ math library for graphics



Math Libraries

A math library is a set of code functions for performing common mathematical operations. Most programming languages come with a basic math library for simple algebraic and trigonometric functions like abs(), sin(), cos(), etc. While we'll certainly make use of these, we'll also need more complex mathematical constructs and functions specific to 3D graphics.

Unfortunately, most languages do not have built-in support for constructs like vectors, matrices and quaternions. We could build these ourselves – a vector, for example, is just a set of 2, 3 or 4 numbers – but we'd also need to implement all of the corresponding mathematical operations. Building a custom math library is entirely possible but is also time consuming, error prone and not strictly necessary, as DirectX comes with its own high-performance math library: DirectX Math.

> *Note that DirectX Math is a library for use in our C++ code. Shading languages, used for writing shaders, contain their own sets of common math functions (which execute on the GPU).*

Choices

For this course, you're required to use DirectX Math. However, at the end of the day, the output of a math library is just a set of numbers: one float value, a vector of float values, a matrix, a quaternion, etc. How we get our final values isn't actually important for rendering, as long as the results are correct.

What this means is that you can technically use any math library with any Graphics API. The OpenGL Mathematics library, GLM, can be used with Direct3D. DirectX Math can be used with OpenGL. You can write your own for use with either API. (But again, for this course, you'll be using DirectX Math.)

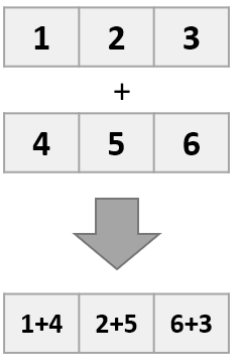
The Hidden Costs of Our Math

We perform a lot of math for games and 3D graphics even before we get the GPU involved. Entities in a 3D scene need vectors for positions, rotations and scales (or quaternions for rotations). Player input often alters this data. Physics and A.I. systems will also use and alter this data as our game simulation runs. These transformations are then combined into 4x4 matrices to simplify later rendering steps. 3D cameras need their own sets of 4x4 matrices. The list goes on.

The true complexity of this math is not always apparent, as many of these operations are actually comprised of several distinct steps. Even if our code only appears to have a single operator, like when we add the two 3D vectors below, multiple steps are actually required to compute the result.

```
// Looks like one operation, but requires 3 separate additions
finalPosition = currentPosition + offset;
```

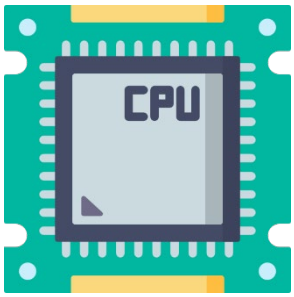
In the example above, adding two 3-component vectors together requires summing each of their respective components as separate actions. In other words, adding the vectors (1,2,3) and (4,5,6) requires 3 discrete steps: add the x's (1+4), then the y's (2+5), and finally the z's (3+6). Here's what the computer actually does:




```
// Actual work to add two 3D vectors
finalPosition.x = currentPosition.x + offset.x;
finalPosition.y = currentPosition.y + offset.y;
finalPosition.z = currentPosition.z + offset.z;
```

All of this math is happening in C++ (the CPU-side of our engines), which means our code is performing each mathematical operation in sequence; there is no inherent parallelism. However, graphics math like the example above often requires performing the same operation on several pieces of data: adding vectors is really several adds, scaling a vector is really several multiplies, etc. In other words, we have a single instruction that we need to apply to multiple pieces of data.

SIMD for CPUs?



Recall the concept of SIMD – Single Instruction, Multiple Data – from our GPU discussions: a single processor instruction, like *add* or *multiply*, can be run by many GPU cores at once. Each core has access to different sets of data (pixels, vertices, etc.), allowing the GPU to perform lots of work at once, as long as it's all the *same* work. This is the basis for modern GPU architecture and rendering pipelines.

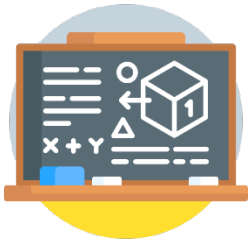
However, CPU instructions are not inherently SIMD-capable, so distributing one program's workload over multiple cores is a much more complex task. While we can technically create and launch separate threads from a single program, there is a lot of operating system overhead in doing so, and probably isn't worth doing over and over for each mathematical operation.

Luckily for us, in the late 90's CPU manufacturers like Intel and AMD realized there was a need for these kinds of math speed-ups. Since then, modern CPUs have included a set of instructions called SSE, or *Streaming SIMD Extensions*, that allow a CPU perform operations on an entire 2, 3 or 4-component vector of data at once. This is essentially small-scale SIMD specifically for graphics-like math on the CPU which can give us an almost *4x speed-up* of the bulk of our C++ math!

However, this feature requires us to write very low-level code, specific to each type of CPU, to ensure the compiler actually uses these special instructions. In addition, it requires our data to be placed in special SIMD-capable registers on the CPU or the instructions don't produce results. As you can imagine, this is more complex than the standard math operator and assignment syntax we're used to. This also means our vector and matrix math code would get very complex very quickly.

So, what's the alternative? We find a math library that provides a relatively simple set of functions and does the heavy lifting behind the scenes; in other words, a SIMD-capable math library like **DirectX Math**.

DirectX Math Overview



The DirectX Math library provides SIMD-friendly data types and functions for graphics math. Because of the aforementioned challenges in utilizing the SIMD capabilities of modern CPUs, the usage pattern of this library will be a little different than what you're used to. That's the trade-off: we lose a little simplicity in our code but we gain performance.

Some official links for your reference: Overview, Getting Started, GitHub

General Usage

DirectX Math comes with the Windows SDK, so the headers should already be available to your project. Include `<DirectXMath.h>` to get started. All structures and functions in the library are in the *DirectX* namespace, meaning your header files should prepend data types with `DirectX::` while your code files can declare `using namespace DirectX;` to simplify variable usage.

> *Though we won't explicitly need it for class, a second header file, DirectXCollision.h, contains structures and functions for basic bounding shapes and their intersections.*

Data Types

The data types in DirectX Math can be broken down into two categories: data types safe for **general storage** and data types for actively **performing SIMD math**.

General Storage Types

These types are simply C++ structs with no added functionality or operator overloads. Their only purpose is to store data while we're not performing math. These are what we'll use for class members. Below are the most common (though our matrices will almost exclusively be 4x4).

Vector Storage Types	*Matrix* Storage Types
XMFLOAT2	XMFLOAT3X3
XMFLOAT3	XMFLOAT3X4
XMFLOAT4	XMFLOAT4X3
	XMFLOAT4X4

SIMD Math Types

There are exactly two of these: **XMVECTOR** and **XMMATRIX**. Each can be thought of as directly representing the special SIMD registers on our CPUs. The library can only perform operations on data within one of these two data types, and any function that produces a result will return one of these two types.

XMVECTOR has room for 4 floats, so it is effectively a 4-component vector, though we can still use it for 2 or 3-component vectors. XMMATRIX has room for 16 floats, so it can represent any matrix up to 4x4. (XMMATRIX is essentially four XMVECTORs internally.)

Because they require specific memory alignment, it is advised to only use these as *local variables*; they should not be class or struct members! The compiler will correctly align local variables, but not class members. If they're not aligned properly, the math quite literally will not work correctly.

DirectX Math General Workflow

The overall workflow of using DirectX Math boils down to the following:

- Keep data in **storage types** (like XMFLOAT3) until you need to perform math
- Ready to do some math?
  - **Load** the data into a **SIMD math type**
  - **Perform** any and all mathematical operations on that data
  - **Store** the data back into a **storage type** when you're done

Using Storage Types

This is the easiest part of using DirectX Math, as the storage types are basic C++ structs. However, do note that there are no operators for the types themselves!

Declaring in Header Files

Remember to use DirectX:: in your header files:

```
class Example
{
private:
	DirectX::XMFLOAT3 translation;
	DirectX::XMFLOAT3 rotation;
	DirectX::XMFLOAT3 scale;
};
```

Declarations, Initializations and Assignments

Local variables within a function:

```
XMFLOAT2 uv(0.0f, 1.0f);
XMFLOAT3 position(1.0f, 0.0f, 0.25f);
XMFLOAT4 color(1.0f, 0.0f, 0.0f, 1.0f);
```

Assigning (overwriting) entire types:

```
scale = XMFLOAT3(1.0f, 1.0f, 2.0f);
uv = XMFLOAT2(1, 1);
```

Using Component Variables

Each storage type has float variables you can directly access and change. Each vector type has a subset of x, y, z, and w members, while the matrix types use members named _11, _12, _34, etc. where the first digit corresponds to a row and the second digit corresponds to a column in the matrix.

```
// Notice we are using individual components (x, y, z, etc.) here!
position.x = 99.0f;
position.y += 2.0f;
position.z = position.x + position.y;

worldMatrix._11 = 1.0f;
worldMatrix._12 = 0.0f;
```

No Operators on Storage Types!

While you can directly access and change storage type members, there are **no overloaded operators** for the types themselves. This is on purpose, as we aren't meant to use these types for SIMD math! This means the following code ***does not work***:

```
// None of these are valid!
position *= 2.0f;
position -= XMFLOAT3(1, 1, 1);
position = position + offset;
```

Using SIMD Math Types

The storage types are easy enough to use but do not provide functionality beyond storing sets of numbers. If we want to perform any complex operations and/or get those speed-ups to our simple vector and matrix math, we need to use data types that map directly to the proper CPU registers: XMVECTOR or XMMATRIX.

Declaring in Header Files

It is advised that you *do not* do this, as class members are not necessarily aligned properly.

Declarations, Initializations and Assignments

You can and should use these as local variables, though you'll need to use special functions to given them initial values. There are no parameterized constructors to start with.

> *Remember: we're not just storing data anywhere in memory; we need these variables to house their data in very specific places on a CPU, so special functions are necessary!*

```
XMVECTOR position; // No values yet
XMVECTOR scale = XMVectorSet(1.0f, 1.0f, 1.0f, 0.0f); // Set all components manually
XMMATRIX identity = XMMatrixIdentity(); // Built-in function for a basic matrix
```

Copying (Loading) From Storage Types

If you want to perform math on data in a storage type, call the corresponding *XMLoad…()* function to copy it to a math type first. Then we can operate on the math type and eventually copy (store) it back.

```
XMVECTOR uvVec = XMLoadFloat2(&uv);
XMVECTOR posVec = XMLoadFloat3(&position);
XMVECTOR colorVec = XMLoadFloat4(&color);
XMMATRIX worldMat = XMLoadFloat4x4(&world);
```

Calculations with Math Types

This is where the fun begins! The math types have common overloaded mathematical operators, such as +, -, * and /. In addition, function versions of each operator exist since the compiler has an easier time optimizing the functions than the operators (meaning the functions are slightly more efficient).

```
// Overloaded operator examples
posVec *= 2.0f;
posVec += offsetVec;
colorVec *= XMVectorSet(0.5f, 0.5f, 1.0f, 0.0f);

// Functions duplicating the above code (more efficient than operators)
posVec = XMVectorScale(posVec, 2.0f);
posVec = XMVectorAdd(posVec, offsetVec);
colorVec = XMVectorMultiply(colorVec, XMVectorSet(0.5f, 0.5f, 1.0f, 0.0f));
```

Built-In Functions

The library provides *hundreds* of functions for all kinds of vector, matrix and quaternion graphics math, as well as comparisons, data conversion and so forth. Far more than I can list here. These functions all consume and/or produce math types, so now that we have those, we can get to work.

```
// Functions for advanced vector3 operations
// Note: Equivalent functions exist for 2 and 4-component vectors as well
XMVECTOR dotVec = XMVector3Dot(forwardVec, rightVec);
XMVECTOR crossVec = XMVector3Cross(forwardVec, rightVec);
XMVECTOR angleVec = XMVector3AngleBetweenVectors(forwardVec, rightVec);

// Matrix creation and manipulation
XMMATRIX transMat = XMMatrixTranslation(2.0f, 5.0f, -1.0f);
XMMATRIX scaleMat = XMMatrixScalingFromVector(scaleVec);
XMMATRIX combinedMatOperator = scaleMat * transMat;
XMMATRIX combinedMatFunction = XMMatrixMultiply(scaleMat, transMat);

// Quaternion creation and manipulation
XMVECTOR quatId = XMQuaternionIdentity();
XMVECTOR quatRotation = XMQuaternionRotationRollPitchYaw(0, 5.0f, 0);
XMVECTOR quatCombined = XMQuaternionSlerp(quatId, quatRotation, 0.5f);
```

Copying (Storing) Our Results

Once we've gotten our finalized data in a math data type, we can copy it back to a storage type using the corresponding *XMStore…()* function. Ideally you should do this when you have the final result of your calculation(s); don't Load/Store/Load/Store over and over between each step.

```
// Provide the address of where the data should be stored (copied)
XMStoreFloat3(&position, posVec);
XMStoreFloat4x4(&world, worldMat);
```

This is fine for entire vectors and matrices, but what if our result is just a single float value, like in the case of a dot product? There are functions for this, too!

```
// From an above example
XMVECTOR dotVec = XMVector3Dot(forwardVec, rightVec);

// Store the vector's first component
float dotProduct;
XMStoreFloat(&dotProduct, dotVec);


// Alternatively, you could use a function to retrieve a specific component
float dotProduct = XMVectorGetX(dotVec);
```

Wait, why is the result of a function like the dot product an entire XMVECTOR type? Shouldn't it just be one number? It's because the SSE instructions on CPUs must output their data to the special SIMD registers, even if the actual result is just a single number. Since that result – the dot product in this case – might be used in the next SIMD calculation anyway, the library keeps it in the XMVECTOR type until we explicitly store it ourselves.

Conclusion

This seems like a lot of extra work! This is an industry-grade math library! It certainly is more complicated than a standard math library, but what we lose in simplicity we gain in performance.

It takes a little practice to get the hang of it, but once you understand the workflow, it'll become second nature. And, since it does most of the heavy lifting for us, our overall work load is lower than having to implement operations like matrix multiplication or vector cross product by hand.