# Custom Math Libraries

Sean Middleditch
DigiPen Game Engine Architecture Club

## Sean Middleditch

DigiPen CSRTIS Senior

GAM200/300/500 TA

Mr. Prez for Engine Club

DigiPen Games: Subsonic, Core

## What We're Talking 'Bout Today

Designing and implementing a math library

What features are needed

Testing and debugging

API design and usability

Miscellaneous Stuffs

## Uses for a Custom Math Library

Why bother?

## Consistency

External math libraries do not generally have the same conventions as the rest of an engine's framework

Writing a new math library can make the code more internally consistent and friendly to new users

## Speed

Many existing math libraries are not highly optimized

Many are designed for ease of implementation rather than speed

Optimized libraries are often tied to a particular platform and hard to port

## Features

Existing math libraries often are lacking important features

Featureful libraries exist, but tend to suffer from usability and performance problems more than their less-featureful brethren

Game engines can have some very particular needs

## Portability

Many excellent math libraries are designed for one platform

Many are designed for one graphics API

Many are designed for one CPU architecture

Many theoretically can be ported, but doing so is difficult without knowledge of internals

## NIH Syndrome

"Not Invented Here" Syndrome - we have an urge to write everything ourselves

Actually an excellent learning exercise, great for students and hobbyists

## Reasons to Ignore This Talk

Why you really shouldn't write your own math library

## Bugginess

Existing math libraries are more or less bug-free

Your library is likely going to be very buggy unless you take extreme care

## Speed

Your library is unlikely to match the speed of certain platform-specific math libraries

## Features

There are a surprising number of features a math library can need, and all the important ones exist in most popular math libraries

You will need to reimplement all the ones you need

## NIH Syndrome

"Not Invented Here Syndrome" is something to learn to avoid

Spending time on things you don't need to spend time on is a waste

Time could be spent making a better game rather than reinventing the wheel

## Do It Anyway

Students should take opportunities to expand knowledge

Pros outweigh the cons

And really, it's not *that* hard to write a great math library

## Features

Things a math library can use

## Vectors

Fundamental type for most games: vectors

Usually have 2-component, 3-component, and 4-component

Most commonly `float` as component type

Useful to have `int` as well, and occasionally useful to have `uint`, `double`, and `bool`

## Matrices

Very handy for transforms, cameras, projection, and other 3D operations

4x4 is most important for 3D (or 3x3 for 2D)

All square matrices up to 4x4 useful at times, rectangular also useful to have

Generally uses `float`, can help to have `double`

## Quaternions

Very handy in 3D for rotations

**Meme**: nobody understands how they work

**Reality**: simple properties and easy to use

Need `float`, *maybe* useful to have `double`

## Basic Operations

Vector operations

Dot product, cross product, magnitude, normalize, algebraic ops, etc.

Matrix operations

Transpose, determinant, inverse, product, etc.

## Graphics Functions

Generate projection matrices

Generate camera matrices

Generate rotation, scale, translation, transform matrices

Decompose transform matrix

## Geometric Library

Shapes like point, line, triangle, sphere, AABB, OABB, capsule, etc.

Trees like KD-Tree, BSP Tree, Octree, etc.

Distance and intersection tests for all shapes

Intersection manifold calculation (Minkowski Portal Refinement, etc.)

## Advanced Operations

There's a whole slew of advanced linear algebra functions and algorithms that can come in handy in specialized cases

This is one of the cases where using an existing math library looks attractive

Upside: your game is highly unlikely to need more than one or two, if that

## Minimum Features

What you need to even make it worth bothering

## The One Fundamental Math Type

You must have a 3-component vector (or 2-component for 2D apps)

- 4-component vector not mandatory
- Matrices are only for CPU-side transforms
- Quaternions are kinda mostly for animation
- Geometric library only needed for some algorithms

## Most Important Functions

Minimum set of projection matrix generation (perspective or orthographic)

- Doesn't need a matrix type
- Can generate a 16-element float array to pass into graphics API, no CPU transform required

Basic vector operations

## Reality Check

You're going to need a lot more than just that

Gives you a good idea of where to start, though

Matrix type and extra vector types good place to go next

## Utilize Time Well

A "complete" math library is a huge undertaking

You don't need everything

Write what you know you need up front

Write anything else that comes up when it comes up

## C++ API Design

Making the math library easy to use

## API Design

The goal of API design is to make a library:

1. Easy to use
2. Easy to discover
3. Difficult to use incorrectly
4. Difficult to use inefficiently

## Conflict Resolution

In cases where there are two ways to do something and you need both, make the common way the easier of the two!

If you don't need both, only add the common case

If you're unsure which is more common, pick the more efficient answer

## Pitfalls in Math Libraries

Optimization is harder than you think

- SIMD optimized libraries that don't work well with SIMD uses

Usability is harder than you think

- Math types as class members can be problematic

Need lots and lots of testing!

## Naming

Appellation consternation

## Naming

Naming things is one of the harder parts of API design

Should the 4-component vector be called `float4`, `vec4`, `f32v4`, or something else?

Largely a matter of personal opinion, but I'll try to sway you towards my own opinion

## Consistency in Naming

D3D prefers `float4` for example

We use types like `int32` for a 32-bit integer

Imagine we want a vector of integers, e.g. `int4`

Imagine we want a vector of `int8`'s... `int84`? `int8_4`? `int8x4`? Now it looks like a matrix!

## Unambiguity & Common Cases

I prefer the GLSL style `vec4` for a 4-component float vector

Three-component int vector would be `ivec3`

Float vectors are the common case, so no prefix for them, and short "clear enough" prefixes for other types

Could use bigger prefix, e.g. `int8vec4`

## Nouns vs Verbs vs Conciseness

Methods should be named as verbs

`transform.Matrix()` **vs** `transform.RecalculateMatrix()`

Sometimes a pain for math ops though

`v1.Dot(v2)` **vs** `v1.ComputeDotProduct(v2)`

## Methods vs Free Functions

Function free or method hard?

## C++ Wisdom

The great minds of C++ recommend using free functions as much as possible

> *"If an operation can be written using an object's public interface, make it a free function"* - paraphrased wisdom

The dot product of a vector for instance only needs to use the vectors public fields, so make it a function

## Practical Guidance

Code completion is awesome

Class inspectors with documentation are awesome

Free functions hinder both greatly; you can see easily inspect all the methods of a vector, but that won't include free functions that operate on vectors

## A Mystery!

Why does this code compile?

```
namespace foo {
  class bar { };

  void dostuff(bar) {}
}

foo::bar quz;
dostuff(quz);
```

## ADL

"Argument Dependent Lookup" is an important feature of C++

If a matching function definition cannot be found, the namespaces of the types passed in are implicitly searched as well

`bar` is in namespace `foo` so C++ will find `foo::dostuff` during name resolution

## ADL for Operators

ADL was originally invented to solve the operator overload problem

You wouldn't want to be forced to include a namespace just to get custom operator support for a type in that namespace

Also works for "named operators" e.g. free functions

## Named Operator Example

Absolute value is an example of a math operator that is expressed as a named function in most programming languages

```
template <typename V>
V ClosestToZero(V a, V b) {
  return std::min(
    std::abs(a),
    std::abs(b));
}
```

## Namespace Problem

Using `std::abs` means that `ClosestToZero` can't work with a custom `GameMath::abs`

```
namespace GameMath {
  vec4 abs(const vec4& v) {
    return vec4(std::abs(v[0]),
      std::abs(v[1]), std::abs(v[2]),
      std::abs(v[3]));
  }
}

// won't compile:
std::abs(vec4());
```

## Rewrite to support ADL

Note that we include the `std` namespace, since builtin types like `float` aren't in std by default, but that's where the `float` version of `abs` is

```
template <typename V>
V ClosestToZero(V a, V b) {
  using std::abs;
  using std::min;
  return min(abs(a), abs(b));
}
```

## Use for ADL & Free Functions

ADL lets you write generic versions of functions like `clamp`, `lerp`, and so on that work with floats, vectors, and even matrices

Even if you prefer methods for code completion, you might consider *also* having free functions for at least those kinds of operations

## Static Methods

Static methods are a (small) step between free functions and methods

Slightly better for discoverably than a free function in a namespace... barely

## Final Verdict

I've written math libraries using free functions, and ones using methods, and ones using static methods, and ones using combinations

In the end, it's almost entirely a matter of taste and style, so go whichever way you like the best

If you've tried one approach before, try another for kicks; maybe you'll like it more, maybe not

## Vector Classes

Developing vector classes

## Approaches

Several approaches to building vector classes

Many different classes for each type

Templates

Hybrid approach

## Templated Vectors

And matrices too

## Template Vectors

Vectors can be defined as a template of both the component type (float, int, etc.) and size (2, 3, 4, etc.)

```
template <typename TYPE, int SIZE>
struct Vector {
  typedef TYPE Type;
  static const int Size = SIZE;

  Type a[Size];

  Type Dot(const Vector& rhs) const;
};
```

## Template Vectors Continued

Notice that you can write most (but not all) vector and matrix operations fully genericized from base type and size

```
template <typename TYPE, int SIZE>
TYPE Vector<TYPE, SIZE>::Dot(const Vector&
rhs) const {
  Type rs = 0;
  for (int i = 0; i < Size; ++i)
    rs += a[i] * rhs.a[i];
  return rs;
}
```

## Template Matrix

```
template <typename TYPE, int MAJOR, int
MINOR>
struct Matrix {
  typedef TYPE Type;
  static const int Major = MAJOR;
  static const int Minor = MINOR;

  Type a[Major][Minor];

  Matrix operator*(const Matrix& rs) const;
};
```

## Problems with Template Types #1

Some operations are not generic

- Cross product is only really defined for 3 components (well, some other sizes too, but certainly not all sizes)
- Matrix inversion only works on square matrices

## Problems with Template Types #2

Templates restrict API design

- Very difficult, ugly, and imperfect tricks required to get friendly constructors
- Requires some fishy policy classes to get component accessors (`.x`, `.y`, etc.)

## Problems with Template Types #3

Templates restrict optimizations

- SSE types require a lot of hand-tweaking of operations
- Cannot easily be written generically, since they only work with 4-component floats (so no 3-component vectors, 5-component vectors, etc.)

## Problems with Template Types #4

Templates bloat compile times

No bullet points on this one, pretty simple

## Template Specializations

All of the major problems with template types can be "fixed" by specializing the templates

Still has compile time problems

Not super useful in games (you mostly use a handful of specific types), except maybe for non-square matrices (2x3, etc.)

## Verdict

Don't use templates for vector types

Consider templates for non-square matrices if and only if you really need them and need a full complement of operators on them

## Specific Types

Less generic, more usable

## Duplicate Types

Write each type as a standalone class

e.g., a specific `vec4`, a specific `vec3`, a specific `mat33`, etc.

This will require rewriting (or cut-n-pasting) lots of code

## Cut-n-Paste... Really?

Yes, really

We are (rightfully) trained to never do this, but it's appropriate (and not terribly inappropriate) in this case

Each type is slightly different, and C++ unfortunately doesn't quite offer all the features needed to avoid the duplication

## Start with the Biggest Types

Implement `vec4` and `mat44` first

Implementing the rest is a matter of copying and stripping down the code

Notable exception is the cross product operator to be written

## SIMD Intrinsics

Making the most of the modern CPU

## SIMD

**"Single Instruction, Multiple Data"**

On desktop Intel compatible (ia32/amd64) CPUs, this is **SSE** (Streaming SIMD Extensions)

On mobile ARM CPUs, this is NEON

On console PPC CPUs, this is AltiVec

## Doing Multiple Things at Once

SIMD instructions allow multiple pieces of data to be processed at once

```
; v1 = v1 * v2 (component-wise)
; four non-SIMD instructions
MUL v1.x, v1.x, v2.x
MUL v1.y, v1.y, v2.y
MUL v1.z, v1.z, v2.z
MUL v1.w, v1.w, v2.w

; v1 = v1 * v2
; one SIMD instruction
MUL v1, v1, v2
```

## Intrinsics

Using SIMD from C++ code is done with CPU intrinsics

These are special built-in functions in the compiler that expose certain CPU instructions

There are different intrinsics for SSE, NEON, and AltiVec, so portability can be a problem

## SSE Intrinsics Headers

`pmmintrin.h` - SSE2

`emmintrin.h` - SSE3

There's around 10 of these for each new version (and the older pre-SSE2 versions)

Stick with SSE2 or SSE3 for now; SSE4+ is new enough that many CPUs in use still don't support it

## Reference Pages

There are tons of ops, hard to remember them

Intel Instrinsics Guide (Windows)

Microsoft MMX, SSE, SSE2 Reference

SSEPlus Reference (Not recommending the library itself, just its concise reference page)

## Using SSE Intrinsics

SSE registers are 128 bits

The primarily support four 32-bit floats

Some instructions available for 64-bit doubles and 32-bit ints (and even 8- and 16-bit ints)

## Notes on SSE Intrinsic Names

Intrinsics start with `__mm_`, which is reserved for compiler use

Most intrinsics you use will end with `_ps` (for "packed single-precision" e.g. 4 floats)

Non-vector math will likely also use SSE ops internally, namely the `_ss` variety (scalar single-precision)

## The Dot Product

Dot product operations are naturally antithetical to SIMD processing in AoS approaches

SSE3 adds `HADDPS`
SSE4 adds `DPPS`

Can't rely on users having those (yet) though

Just have to suck it up and deal with it

## Advanced SSE Optimizations

There's plenty more information on optimization SSE at sites like Optimizing for SSE

Instruction pairing and data dependencies are a big one

Compiler can do a decent job of this for you... sometimes

## SSE Alignment

Heading down the right path when using SSE

## Alignment

Memory storing SSE values must be aligned to 16-byte (128-bit) boundaries to efficiently load into SSE registers (`__mm_loada_ps`)

**On most platforms,** `malloc` **and** `new` **only guarantee 8-byte (64-bit) alignment!**

This is probably the "hardest" part of using SSE

## Alignment Solution #1

The first solution to the alignment problem is to make sure all SIMD vectors are aligned with custom memory allocators

Annotate vector/matrix types: `__declspec (align(16))`

Write custom `new`, `delete`, `malloc`, `free`

Pain in the butt for student/hobby games

## Alignment Solution #2

Write two versions for all maths types:

- "Storage" versions to put in classes
- "Local" versions for writing math kernels

Storage versions use unaligned loads, but all heavy work done using local versions

Difficult to enforce proper use of types, lots of extra work, error prone

## Alignment Solution #3

Use unaligned loads (`__mm_loadu_ps`)

Least efficient (but still way more efficient than not using SIMD at all!)

Easiest by a very wide margin

## Recommendation

For a student game, I recommend solution #3

For more advanced projects, solution #1 is a good path (custom memory managers aren't all that hard) but there are other occasional issues to deal with (do you really want your vertex buffers to be forced to 16-byte alignment and hence 16-byte multiples?)

Solution #2 is "best" but rarely worth the effort

## AoS vs SoA

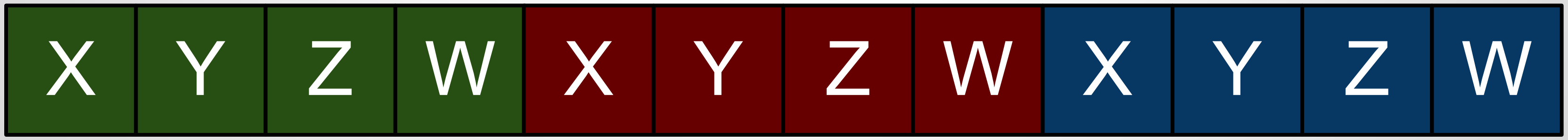
Acronym of Silliness vs Surplus of Acronyms?

## AoS

**AoS** (Array of Structures)

A structure contains the x, y, z, w values of a vector

You have an array of those

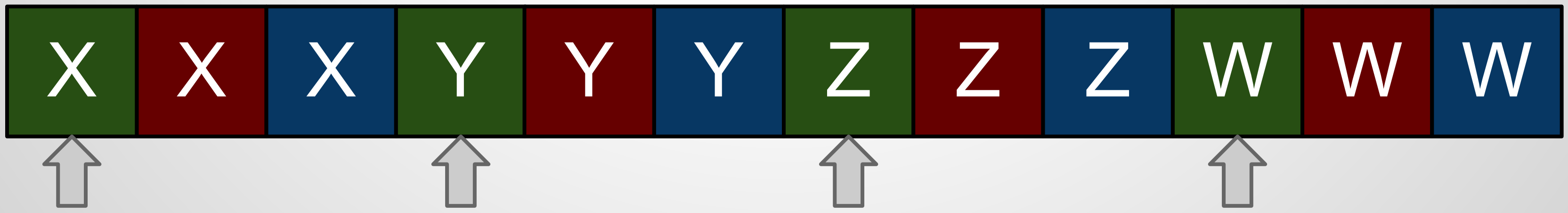


## SoA

**SoA** (Structure of Arrays)

Use an array for each component (x, y, z, w), or one array split into four regions

Structure contains those four arrays/regions



## Speed

SoA is the fastest approach

Allows the best use of data parallelism

Fast dot product!

Process each component of four vectors simultaneously

## Ease of Use

AoS is the far more practical approach for most game needs

*Much* easier to have a `Transform` object with a position than to have a structure with four arrays for each component of all positions

**The GPU basically requires that you use AoS**

## Some Benchmarks

Benchmarks for AoS vs SoA

AoS & SoA Explorations (Part 1)

## The Trouble With Return Values

What to return and when

## Problem

What potential major inefficiency exists here?

```
vec4 a;
vec4 b;
vec4 c;

vec4 d = dot(a, b) * c;
```

## The Dot Product Problem

The dot product returns a scalar

SSE math (including dot products) produces and consumes vectors

Converting a vector op to a scalar, returning that, and then converting it back into a vector to scale another vector is wasteful

## Example of Difference (SSE4.1)

```
; ideal code
DPPS xmm0, xmm1, 0xFF
MULPS xmm2, xmm0

; treating dot product as a scalar result
DPSS xmm0, xmm1, 0xFF
MOVSS $tmp, xmm0
  ; function call boundary
MOVSS xmm3, $tmp
SHUFPS xmm3, xmm3, 0x0
MULPS xmm2, xmm3
```

## Solution

Have a version of dot product that returns a vector (each component contains the dot product)

This version will likely be used more often than the version that returns a scalar, so consider making it the shorter/easier version to use

## Naming Examples

Maybe `DotScalar` vs `DotVector`

Could use `dots` or `sdot` vs `dotv` or `vdot`

One version could just be `Dot` and the other gets the suffix/prefix

Find a balance of clarity, ease of use, discoverability, and your personal taste

## The Dual Nature of Vec4

Vec4 serves many purposes

## Vec4 vs Vec3

Vec4 is often used in place of a vec3

Not just for affine transforms; vec4 can be SIMD accelerated, vec3 cannot

For optimized code, vec4 will be used for all operations, and vec2 and vec3 mostly only used for storage in objects or VBOs

## Vec4 Pretending To Be a Vec3

It can be useful to have a cross product operation for vec4 that simply ignores the w component

It can also be useful to have matrix multiplication operations that implicitly override the fourth component to be 0 or 1

Same goes for magnitude, normalization, etc.

## Quaternions

Not that scary

## What Quaternions Are

Quaternions are an extension of complex numbers

They can be thought of as a combination of a 3-component vector and a scalar

## What Quaternions Are Good For

Quaternions can efficiently represent rotation in 3D space

The vector component represents the axis, scaled by the sine of half the angle

The scalar component reprsents the cosine of half the angle

## Normalized Quaternions

All the easy efficient math for spatial rotations with quaternions requires them to be normalized

Note that this is not a normalized axis vector, but a normalized quaternion

Normalizing a quaternion is the same as normalizing a vec4

## Quaternions vs Vec4

It can seem tempting to just use a vec4 for quaternions

Multiplying a quaternion by a vec4 should perform the rotation logic, not a component-wise vector multiplication though

There are many other different operations; make them separate types

## Not Scary

Plenty of articles online on what math to use

Actually explaining and mastering the math is a whole course (which DigiPen has, naturally)

Just using them to handle 3D rotations is dead easy, though

# Graphics Math

Using math with graphics APIs

## OpenGL vs Direct3D

There's actually almost no difference

D3D math is typically left-handed, row-major, with pre-multiplication

GL is typically right-handed, column-major, with post-multiplication

Neither API actually cares what you use, though

## Staying Compatible

Row-major matrices should use pre-multiplication

Column-major matrices should use post-multiplication

Ensures that the resulting matrices have an identical in-memory layout, and hence are compatible

## Handedness

The APIs don't actually care about coordinate system handedness

They use whichever camera setup you define

You can use any crazy coordinate system you want, it really doesn't matter

Just remember to set triangle winding!

## Projection Problems

Keep in mind that GL and D3D use different origin coordinates for screen and texture coordinates

Also, OpenGL maps depth buffers to [-1,+1] by default (can be changed),

Projection matrix functions may have to differ for GL and D3D if you don't reconfigure defaults

## Handy Dandy Articles

Matrices, Handedness, Pre and Post Multiplication, Row vs Column Major, and Notations

All on seanmiddleditch.com

## Geometric Math

Handling geometry client-side

## Geometric Math Uses

Rendering on the GPU, but culling on the CPU

Build spatial data structures on the CPU

Physics generally on the CPU

Ray testing and picking on the CPU

## Important Operations

Distance between two geometric objects

Closest point between two geometric objects

Intersection point (or shape) between two geometric objects

## The Orange Book

Good covering of geometric and collision math is part of the book Real-Time Collision Detection

realtimecollisiondetection.net

## Heads Up for DigiPen Students

You need a complete geometric library for DigiPen's CS350

It's a lot of work for a two week project at the start of the semester

Get it done ahead of time if you can!

## Testing

Making sure it all works

## Importance of Testing

If the math library doesn't work, developers have trouble writing their own code

"Physics doesn't work" - is it because the physics code is wrong, or the math library is wrong?

Write tests for all math operations

## Test-Driven Development

Write tests before you implement features

Saves you a lot of time by finding bugs early on in your code

No sense wondering why your camera matrix generator doesn't work if you could have easily found out your vector normalize doesn't work

## Writing Good Tests

Writing tests is an art form unto itself

Need to thoroughly test general functionality

Need to think of all the **boundary conditions** (conditions just at the edges of valid and invalid ranges)

Find values that actually test the math

## Writing Bad Tests

In a math library, bad tests are ones that only stress super simple cases

For example, axis-aligned unit vectors

> Test those *too*, but don't let them be your only test values!

Bad tests have lots of math that "cancels out"

## Generating Test Data

Plug values into an existing well-tested math library (or two) to generate expected results

Feel free to share test data with other folks; it's useful and friendly to do

Naturally, mistrust any data shared with you; be sure your tests are broken, not that the data is broken

## Right Kind of Programmer

Not all programmers are great test engineers

Even skilled programmers are often just bad at building good tests

Look at the talking man to the left of these slides for an example of such a person

Find a person who's good at writing tests, and treat them like royalty for their talents

## Summary

Take away from this talk

## Key Points

- Keep it Simple
- Design APIs for ease of correctness and usability
- SIMD makes math go zoom
- Use Test Driven Development practices

## Questions & Answers