Intrinsics & SIMD

Optimized C++

Ed Keenan

13 February 2019

13.0.6.4.5 Mayan Long Count

DePaul University

Goals

- **Intrinsics**
  - What are intrinsics?
  - How to use intrinsics?
  - Trends
- **SIMD**
  - Old style, too much assembly like alternative
  - Intrinsics SSE
- **Esoteric knowledge**
  - Increase your geek points
  - Impressed your friends
  - A Good seasoning…
    - Do not over use

5 out of 4 programmers are bad at math?

What are intrinsics?

- **Compiler intrinsics are psuedo-functions that expose CPU functionality**
  - Think of them as “*Compiler fragment macros*”
  - Intrinsics are the natural evolution from inline assembly
- **Common usage**
  - Implement vectorization and parallelization in languages which do not address such constructs

What are intrinsics?

- **Similar to inline functions**
  - Substitutes a sequence of automatically-generated instructions for the original function call.
- **Different than inline function**
  - Compiler has an intimate knowledge of the intrinsic function
  - Better integrate it and optimize it for the situation.

Properties of Intrinsics

- **Inline**
  - Code segment for that function is usually inserted inline,
  - Avoiding the overhead of a function call.
- **Written by compiler teams**
  - Highly efficient machine instructions
  - Access to specialized instructions
- **Very Fast**
  - Faster than the equivalent inline assembly
  - Optimizer has a built-in knowledge of how many intrinsics behave

Properties of Intrinsics

- **Assembly too limiting**
  - Some optimizations can be available that are not available when inline assembly is used.
  - Reorder intrinsics to mask off staging and processor delays
- **Black magic**
  - Also, the optimizer can expand the intrinsic differently
  - Align buffers differently
  - Adjustments depending on the context and arguments of the call

General Intrinsics

- **Most functions are contained in libraries**
  - Some functions are built in
    - ***intrinsic*** to the compiler
  - These are referred to as intrinsic functions or intrinsics.
  - Many functions look them up!
    - Math functions:
      - *acos acosf acosl asin asinf asinl*
      - *atan atanf atanl atan2 atan2f atan2l*
      - *ceil ceilf ceill cosh coshf coshl cos*
      - *sosf cosl exp expf expl floor floorf floorl*
      - *fmod fmodf fmodl log logf logl log10 log10f*
      - *log10l pow powf powl sin sinf sinl sinh sinhf*
      - *sinhl sqrt sqrtf sqrtl tan tanf tanl tanh tanhf tanhl*

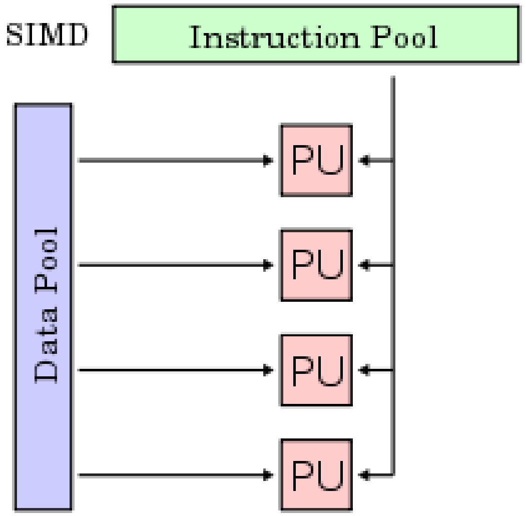
64-Bit compilers

- **Microsoft and Intel's C/C++ compilers as well as GCC implement intrinsics that map directly to the x86 SIMD instructions**
  - MMX
  - SSE
  - SSE2
  - SSE3
  - SSSE3
  - SSE4
- **64-bit windows**
  - Microsoft compiler (VC2005 as well as VC2008) inline assembly is not available when compiling for 64 bit Windows
  - Intrinsics are the only alternative
  - New intrinsics have been added that map to standard assembly instructions

SIMD history

- **SIMD**
  - Started early 1980's
  - DSP (Digital Signal Processors)
    - AT&T
      - DSP1 – 1980
    - Texas Instruments
      - TMS32010 - 1983
  - Wicked software processing
    - Convolution
    - FFT (Fast Fourier Transforms)
    - FIR filters and more

- **SIMD**
  - *Single Instruction, Multiple Data*

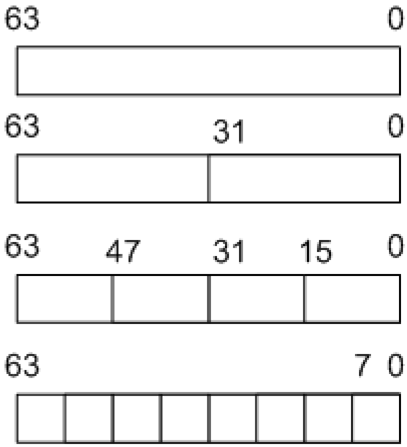


MMX - introduction

- **MMX is a single instruction, multiple data (SIMD) instruction set**
  - Designed by Intel
  - Introduced in 1997 in their Pentium line of microprocessors
  - MMX – “Matrix Math Extensions”
- **Competition: AMD vs Intel**
  - AMD enhanced Intel's MMX with the 3DNow!
    - Added floating point
  - Intel added floating-point math
  - Created the SSE extension two years later.

MMX - internals

- **MMX added eight new registers to the architecture**
  - MM0 through MM7
- **MMn registers holds a 64-bit integer.**
  - Concept of packed data types
    - single 64-bit integer (quadword)
    - two 32-bit integers (doubleword)
    - four 16-bit integers (word)
    - eight 8-bit integers (byte)



MMX instruction set

- Pack and unpack instructions:
  - PUNPCKLDQ, PUNPCKLWD, PUNPCKLBW,
    PUNPCKHDQ, PUNPCKHWD, PUNPCKHBW,
    PACKSSWB, PACKSSDW, PACKUSWB
- Shift instructions:
  - PSLLQ, PSLLD, PSLLW,
    PSRLQ, PSRLD, PSRLW,
    PSRAD, PSRAW
- Move instructions:
  - MOVQ and MOVD.
- Addition and subtraction operands:
  - PADDB, PADDW, PADDD,
    PADDUSB, PADDUSW,
    PADDSSB, PADDSSW,
  - PSUBB, PSUBW, PSUBD,
    PSUBUSB, PSUBUSW,
    PSUBSSB, PSUBSSW
- Multiplication instructions:
  - PMADDWD, PMULHW, PMULLW
- Comparison instructions:
  - PCMPEQB, PCMPEQW, PCMPEQD,
  - PCMPGTB, PCMPGTW, PCMPGTD
- Logical operation instructions:
  - PAND, PANDN, POR, PXOR
- EMMS is in a category of its own
  - instruction restores the floating point state

Sample MMX

```
void additiveblend2(unsigned char *dst, const unsigned char *src, int (
    __asm {
        mov     ecx, src
        mov     edx, dst
        mov     eax, quads
        pxor        mm7, mm7
top:
        movd        mm0, [ecx]      ;load four source bytes
        movd        mm1, [edx]      ;load four destination bytes
        punpcklbw   mm0, mm7        ;unpack source bytes to words
        punpcklbw   mm1, mm7        ;unpack destination bytes to words
        paddw       mm0, mm1        ;add words together
        packuswb    mm0, mm1        ;pack words with saturation
        movd        [edx], mm0      ;store blended result
        add     ecx, 4              ;advance source pointer
        add     edx, 4              ;advance destination pointer
        dec     eax                 ;loop back until done
        jne     top
        emms
    }
}
```

What do you think?

- **My thoughts**
  - Yuck
  - Evil
  - Crap…
- **Not assembly**
  - please no assembly!
- **MMX not that useful for our applications**
  - We use many floating point operations
  - SSE to the rescue

SSE

- **Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86**
  - Intel - 1999 in their Pentium III
  - Reply to AMD's 3DNow!
  - SSE contains 70 new instructions.
    - Scalar and packed floating point instructions.

SSE registers

- **SSE defines 8 new 128-bit registers**
  - (xmm0 ~ xmm7)
    - single-precision floating-point computations.
- **These registers are used for data computations only.**
  - Since each register has 128-bit long
    - can store total 4 of 32-bit floating-point numbers
      - (1-bit sign, 8-bit exponent, 23-bit mantissa).

SSE Instructions

Floating point instructions

- Memory-to-Register / Register-to-Memory / Register-to-Register data movement
  - Scalar – MOVSS
  - Packed – MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
- Arithmetic
  - Scalar – ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
  - Packed – **ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS**
- Compare
  - Scalar – CMPSS, COMISS, UCOMISS
  - Packed – CMPPS
- Data shuffle and unpacking
  - Packed – SHUFPS, UNPCKHPS, UNPCKLPS
- Data-type conversion
  - Scalar – CVTSI2SS, CVTSS2SI, CVTTSS2SI
  - Packed – CVTPI2PS, CVTPS2PI, CVTTPS2PI
- Bitwise logical operations
  - Packed – ANDPS, ORPS, XORPS, ANDNPS

SSE Instructions

Integer instructions

- Arithmetic
  - PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW
- Data movement
  - PEXTRW, PINSRW
- Other
  - PMOVMSKB, PSHUFW

Other instructions

- MXCSR management
  - LDMXCSR, STMXCSR
- Cache and Memory management
  - MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE

Example: Cross Product

```
Vector4 SSE_CrossProduct(const Vector4 &Op_A, const Vector4 &Op_B)
{
    Vector4 Ret_Vector;
    __asm
    {
        MOV EAX Op_A                    // Load pointers into CPU regs
        MOV EBX, Op_B

        MOVUPS XMM0, [EAX]              // Move unaligned vectors to SSE regs
        MOVUPS XMM1, [EBX]
        MOVAPS XMM2, XMM0               // Make a copy of vector A
        MOVAPS XMM3, XMM1               // Make a copy of vector B

        SHUFPS XMM0, XMM0, 0xD8         // 11 01 10 00  Flip the middle elements of A
        SHUFPS XMM1, XMM1, 0xE1         // 11 10 00 01 Flip first two elements of B
        MULPS  XMM0, XMM1               // Multiply the modified register vectors

        SHUFPS XMM2, XMM2, 0xE1         // 11 10 00 01 Flip first two elements of the A copy
        SHUFPS XMM3, XMM3, 0xD8         // 11 01 10 00  Flip the middle elements of the B
copy
        MULPS XMM2, XMM3                // Multiply the modified register vectors

        SUBPS  XMM0, XMM2               // Subtract the two resulting register vectors

        MOVUPS [Ret_Vector], XMM0       // Save the return vector
    }
    return Ret_Vector;
}
```

// R.x = A.y * B.z - A.z * B.y

// R.y = A.z * B.x - A.x * B.z

// R.z = A.x * B.y - A.y * B.x

Instructions vs Intrinsics

Multiply a Vector by a Scalar and return the result

```
Vector4 SSE_Multiply(const Vector4 &Op_A, const float &Op_B)
{
        Vector4 Ret_Vector;

        // Create a 128 bit vector with four elements Op_B
        __m128 F = _mm_set1_ps(Op_B)

        __asm
                {
          // Load pointer into CPU reg
          MOV EAX, Op_A

          // Move the vector  to an SSE reg
           MOVUPS XMM0, [EAX]

          // Multiply vectors
           MULPS XMM0, F

          // Save the return vector
           MOVUPS [Ret_Vector], XMM0

          }

        return Ret_Vector;
}
```

```
Vect SSE_Multiply(const Vect &A, const float &scale)
{

        Vect B;
        Vect C;

        // Create a scale vector
        // [ scale | scale | scale | scale ]

        B.m = _mm_load1_ps( &scale );

        //  multiply vector by scale
        // [ s * A.x | s * A.y | s * A.z | s * A.z ]

        C.m = _mm_mul_ps(A.m, B.m);

        return C;
}
```

Which one did you like?

- **Intrinsics allow the code to look more C-like.**
  - Easier to read and understand
  - Faster to implement and experiment
  - No Assembly flash backs
- **Use the intrinsics**
  - Assembly instructions are being omitted for future extensions to the compilers
  - Compiler support will only be through intrinsics

Retro PS2 code alert~

```
#elif (MATH_IS_PS2)

	register Matrix out(MATRIX_NO_INIT);

	asm __volatile__ ("

			// First matrix
			lqc2			vf1, 0x00(%0)
			lqc2			vf2, 0x10(%0)
			lqc2			vf3, 0x20(%0)
			lqc2			vf4, 0x30(%0)

			// Second matrix
			lqc2			vf5, 0x00(%1)
			lqc2			vf6, 0x10(%1)
			lqc2			vf7, 0x20(%1)
			lqc2			vf8, 0x30(%1)

			// Add
			vadd			vf1, vf1, vf5
			vadd			vf2, vf2, vf6
			vadd			vf3, vf3, vf7
			vadd			vf4, vf4, vf8

			// Store
			sqc2			vf1, 0x00(%2)
			sqc2			vf2, 0x10(%2)
			sqc2			vf3, 0x20(%2)
			sqc2			vf4, 0x30(%2)

	":
	 : "r" (this),
	   "r" (&a),
	   "r" (&out)
	 : );

	return out;
```

Alignment

- __declspec(align(#)) to control the alignment of user-defined data
  - Example 1:
    ```
    __declspec(align(32)) struct Str1{ int a, b, c, d, e; };
    ```
  - Example 2:
    ```
    #define CACHE_ALIGN __declspec(align(32))

    struct CACHE_ALIGN  S1
    {
        // cache align all instances of S1
        int a;
        int b;
        int c;
        int d;
    };
    ```
- All SSE instructions must be 16-Byte aligned
  - __declspec(align(16))
  - Example:
    ```
    __declspec(align(16)) float a[4];
    ```

__m128

- The **__m128 data** type, for use with the SSE and SSSE instructions intrinsics, is defined in ***xmmintrin.h***
  - __m128

- Example:

```
#include <xmmintrin.h>
int main()
{
    __m128 x;
}
```

```
typedef union __declspec(intrin_type)
    _CRT_ALIGN(16) __m128
{
     float                    m128_f32[4];
     unsigned __int64         m128_u64[2];
     __int8                   m128_i8[16];
     __int16                  m128_i16[8];
     __int32                  m128_i32[4];
     __int64                  m128_i64[2];
     unsigned __int8          m128_u8[16];
     unsigned __int16         m128_u16[8];
     unsigned __int32         m128_u32[4];
 } __m128;
```

Data Structure

```
Class Vect
{
	// anonymous union
	union
	{
		__m128	m;

		// anonymous struct
		struct
			{
			float x;
			float y;
			float z;
			float w;
			};
	};
};
```

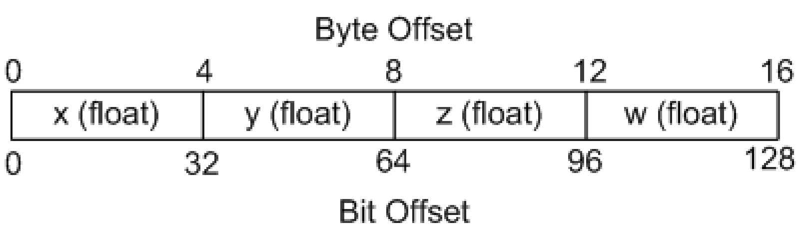
- **Unions**
  - Clever way to hide the alignment croft.
  - Unions must span the size of the largest component
  - Must be aligned to the most restrictive element.
- **Anonymous Unions allows access like:**

```
Vect A;

A.x = 5.0f;
A.y = 6.0f;

__m128 B = A.m;
```

Variables in memory

- It's all the same memory, just it can be seen from different ways.
  - X - element (0 bytes offset)
  - Y - element (4 bytes offset)
  - Z – element (8 bytes offset)
  - W - element (12 bytes offset)

Name	Value	Type
A	{m={...} x=1.00000	Vect_SIMD
m	{4, 3, 2, 1}	__m128
x	1.0000000	float
y	2.0000000	float
z	3.0000000	float
w	4.0000000	float



A.m	{4, 3, 2, 1}	__m128
m128_f32	0x0012fe50	float [4]
[0x0]	1.0000000	float
[0x1]	2.0000000	float
[0x2]	3.0000000	float
[0x3]	4.0000000	float

Memory 1

0x0012FE50	1.0000000
0x0012FE54	2.0000000
0x0012FE58	3.0000000
0x0012FE5C	4.0000000

Scale & Packed Data

- **SSE defines two types of operations**
  - scalar and packed.
- **Scalar**
  - Scalar operation only operates on the least-significant data element (bit 0~31)
  - SSE instructions have a suffix **-ss** for scalar operations (*Single Scalar*)
- **Packed**
  - Packed operation computes all four elements in parallel.
  - SSE instructions have a **-ps** for packed operations (*Parallel Scalar*).

Add PS (packed scalar)

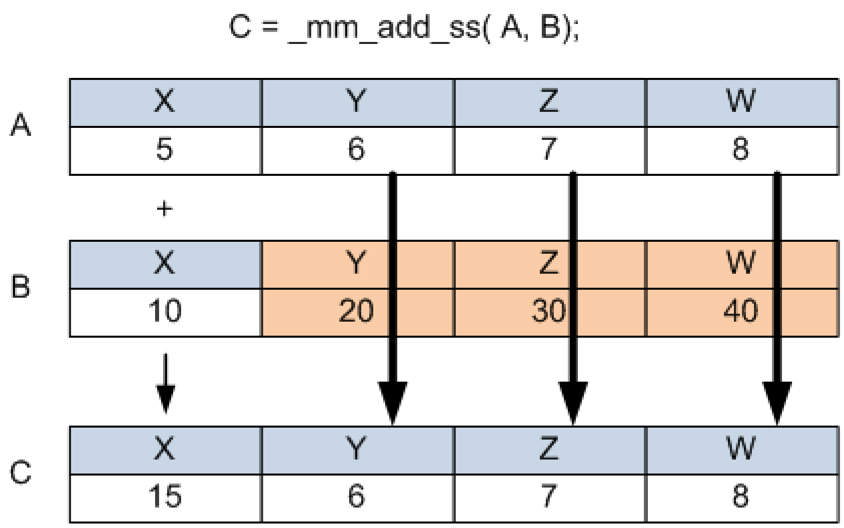
Intrinsic	Instruction	Operation	R0	R1	R2	R3
_mm_add_ps	ADDPS	Adds	Copy Code a0 [op] b0	Copy Code a1 [op] b1	Copy Code a2 [op] b2	Copy Code a3 [op] b3

C = _mm_add_ps(A, B);

	X	Y	Z	W
A	5	6	7	8
	+	+	+	+
B	10	20	30	40
	↓	↓	↓	↓
C	15	26	37	48

Add SS (single scalar)

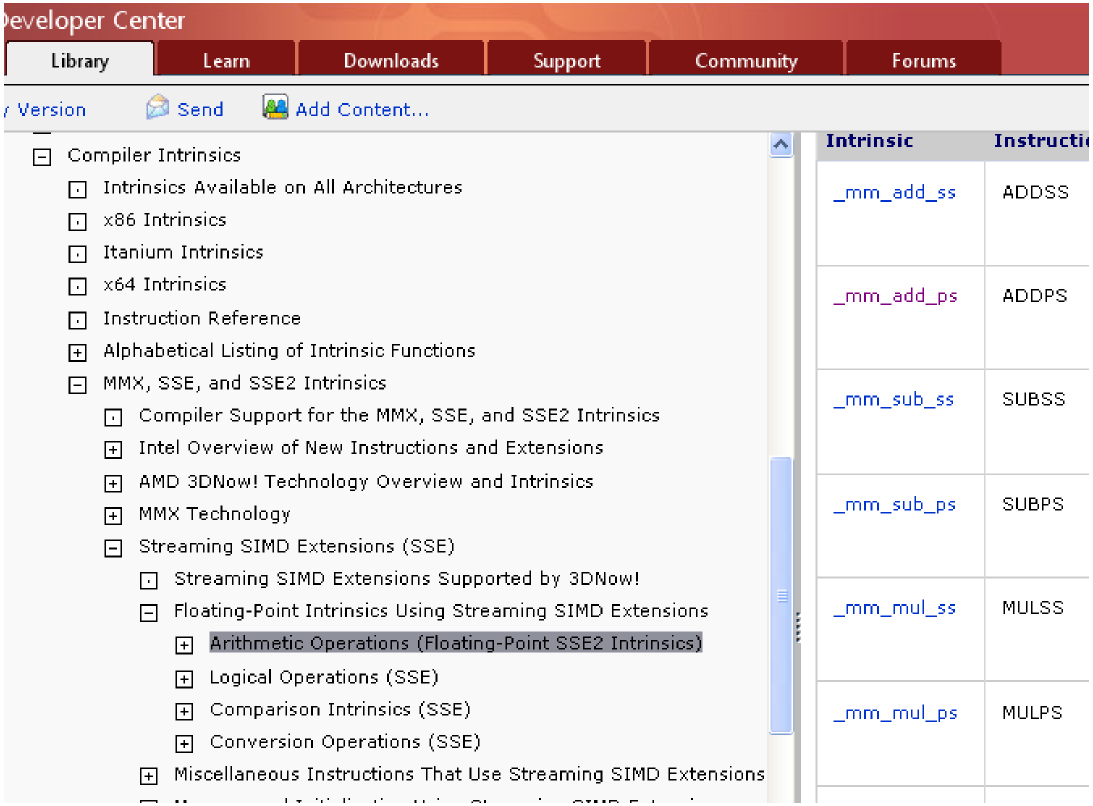
Intrinsic	Instruction	Operation	R0	R1	R2	R3
_mm_add_ss	ADDSS	Adds	Copy Code a0 [op] b0	Copy Code a1	Copy Code a2	Copy Code a3



Reference instructions

- Microsoft Reference manual
  - live in this link:

http://msdn.microsoft.com/en-us/library/708ya3be%28v=vs.100%29.aspx



SSE2

- **SSE2 - Streaming SIMD Extensions 2**
  - IA-32 SIMD instruction sets.
  - SSE2 Intel - Pentium 4 in 2001.
- **SSE2 adds 70 instructions**
  - Allows you to now use double precision.
  - Since doubles are 64-bits, __m128 only holds 2 doubles
  - So it behaves like a double precision MMX
  - 2 doubles (SIMD) in the multiple data.
- **-dp in the instructions**
  - Not that useful for game development
  - We are single precision ☺

SSE3 (minor extension)

- **SSE3 - Intel code name Prescott New Instructions (PNI)**
  - 3rd the SSE instruction set for the IA-32
  - Intel introduced SSE3 in early 2004 for Pentium 4 CPU.
- **SSE3 contains 13 new instructions**
  - Additional functions deal with integer land extensions.
  - Not that useful ☹

SSSE3

- **Supplemental Streaming SIMD Extension 3 (SSSE3)**
  - Intel's name for the SSE instruction set's fourth iteration
  - Didn’t want to rev the number to SSE4
  - My guess, it didn’t get done in time for SSE3 release….
- **SSSE3 contains 16 new discrete instructions over SSE3.**
  - More packed integer data instructions
  - Not useful for Games, but a cool reference

SSE4

- **SSE4 another extension**
  - It was announced on September 27, 2006
  - Became available at the Spring 2007
  - 54 new instructions
    - 47 new instructions v4.1
    - 7 new instruction v4.2
- **Some crazier extensions**
  - 4.1 Version
    - HDTV codecs
    - Added, dot product
      - _mm_dp_ps()
  - 4.2 Version
    - CRC32 function

SSE5 / Future….

- **SSE5 here's where it gets weird**
  - SSE5 – AMD supported part of the SSE5 spec then changed some stuff for their processors.
    - So it's a spec / but not official
  - Intel came out with their AVX-512 instruction set
    - Added 512 registers
    - MAC instructions and backward compatibility with the SSE instruction set
    - Too much crap to walk through …
      - if you care – dig into this stuff more.

This Class

- **Optimized C++**
  - Restrict use to SSE4.1 and under for this class
    - Nothing higher allowed!
- **__cpuid, __cpuidex**
  - Can query to see what version your processor supports
- ***WARNING…***
    - Many people got a 0 on Particles
    - They used a version of SIMD that wasn't supported on the TESTING PC.

Coding with SSE

- **To code using SSE is easy and a puzzle.**
  - Get the data correctly formatted.
  - Use the data ***Union*** trick I came up with.
    - Removes the alignment issues
  - Off to the races piecing the puzzle together.
- **Keep the reference manual open**
  - Single step everything
  - Look at the data
  - Use unique data to start with
    - A(1,2,3,4) and B(10,20,30,40).
    - Easy to determine any aliasing or instruction confusion

Add PS (packed scalar)

Intrinsic	Instruction	Operation	R0	R1	R2	R3
_mm_add_ps	ADDPS	Adds	Copy Code a0 [op] b0	Copy Code a1 [op] b1	Copy Code a2 [op] b2	Copy Code a3 [op] b3

C = _mm_add_ps(A, B);

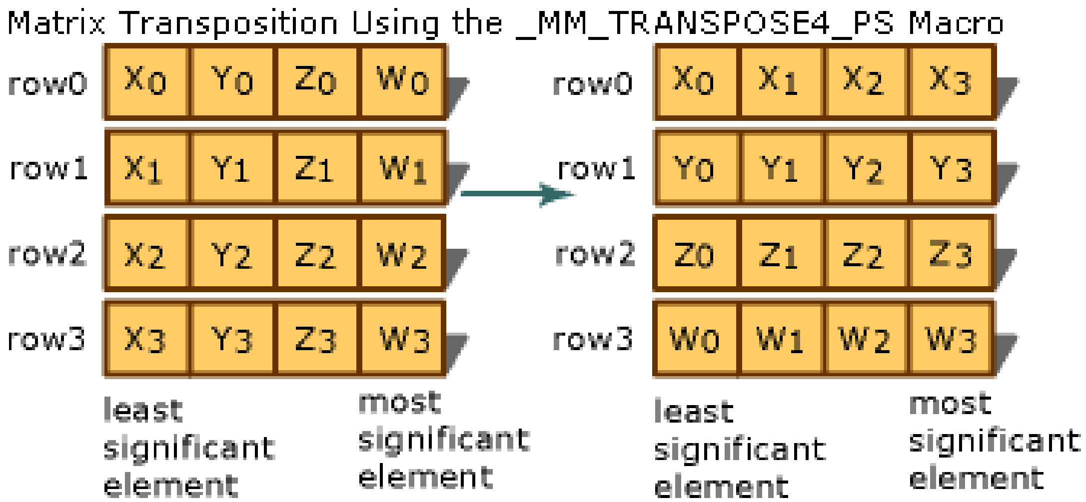
	X	Y	Z	W
A	5	6	7	8
	+	+	+	+
B	10	20	30	40
	↓	↓	↓	↓
C	15	26	37	48

Code Experiment:

```
Vect_SIMD A(1,2,3,4);
Vect_SIMD B(10,20,30,40);
Vect_SIMD C;

C.m = _mm_add_ps( A.m, B.m );
```

Transpose macro

_MM_TRANSPOSE4_PS(row0, row1, row2, row3)



- Convert row vectors into column vectors

Dot Product: _mm_dp_ps()

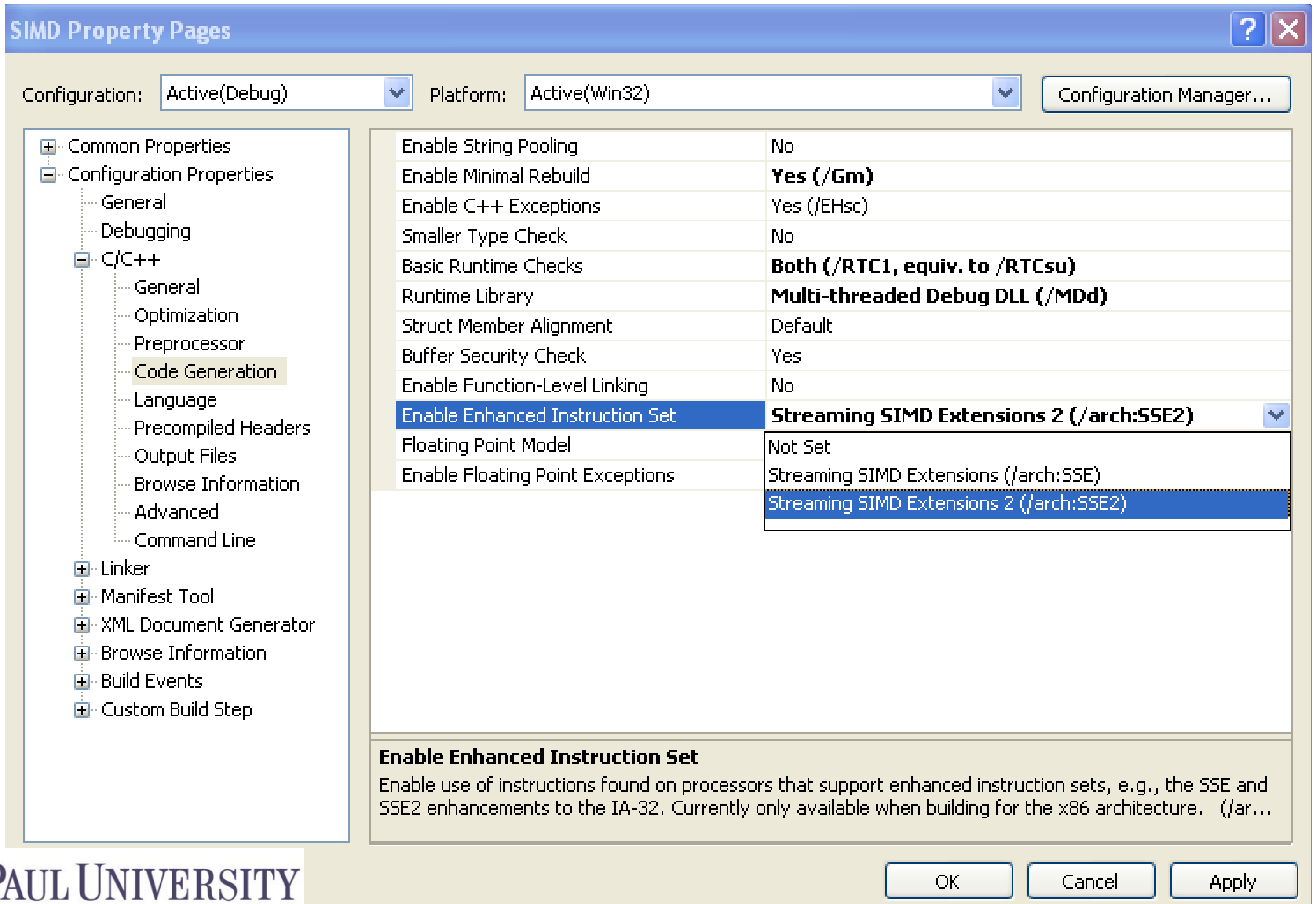
- Way cool…
  - SSE4.1 now has dot products…
    - About time, DSP's had them for 20 years.
  - _mm_dp_ps(A,B,Mask)
    - Mask sets who gets multiplied and written

```
tmp0 := (mask4 == 1) ? (a0 * b0) : +0.0
tmp1 := (mask5 == 1) ? (a1 * b1) : +0.0
tmp2 := (mask6 == 1) ? (a2 * b2) : +0.0
tmp3 := (mask7 == 1) ? (a3 * b3) : +0.0

tmp4 := tmp0 + tmp1 + tmp2 + tmp3

r0 := (mask0 == 1) ? tmp4 : +0.0
r1 := (mask1 == 1) ? tmp4 : +0.0
r2 := (mask2 == 1) ? tmp4 : +0.0
r3 := (mask3 == 1) ? tmp4 : +0.0
```

You can get help

- **Starting with Visual Studio 8 will self generate SSE instructions by simply flipping a switch.**
  - Cheap and Cheezy
  - Outstanding…
- Thank you Herb Sutter

Visual Studio 10 - SSE



AVX – arm race continues

- Advanced Vector Extensions (AVX or AVX-256) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008
- AVX-256 expands SIMD sse to 256-bit support
  - 1st in Sandy Bridge processor
    - Q1 2011 -Intel
  - 1st in Bulldozer processor
    - Q3 2011 – AMD
- AVX2 expands most integer commands to 256 bits and introduces FMA.
  - AVX-512 expands AVX to 512-bit support

AVX – details

- AVX extends even further, from 128 bits to 256 bits
  - ***<u>VEX</u>*** prefix
    - ***<u>V</u>***ector ***<u>EX</u>***tension
- Processors with AVX support, the legacy SSE instructions
  - Extended using the VEX prefix to operate on the lower 128 bits
- AVX-512 register scheme as extension from the AVX
  - ***<u>EVEX</u>*** prefix
    - ***<u>E</u>***xtension ***<u>V</u>***ector ***<u>EX</u>***tension
- AVX introduces a three-operand SIMD instruction format
  - Destination register is distinct from the two source operands.
  - An [SSE] instruction using the conventional two-operand form $a = a + b$ can now use a non-destructive three-operand form $c = a + b$, preserving both source operands.
- Alignment requirement of SIMD memory operands is relaxed.

AVX – Layout

- AVX-512 (ZMM0-ZMM31)
- AVX (YMM0-YMM15)
- SSE (XMM0-XMM15)

511 … 256	255 … 128	127 … 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6

511 … 256	255 … 128	127 … 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Thank You!

- Questions?

Vector & Matrix: Quick & Dirty

Optimized C++

Ed Keenan

DePaul University

Overview

- Making a fast library
  - Lessons from a battled scared programmer
- Math Library
  - What do you use Matrices
- 1x4 Vectors
  - Standard operators
  - Dot / Cross
- 4x4 Matrices
  - Standard operators
- 3D transformations
  - Tying this all together
- Why is this good?

Making Your Library FAST

- Quick summary of how to make a fast library
  - If you took Game Physics or Engine
    - You should have an idea of my thoughts.
    - Here they are again.
- This knowledge is a reaction to bad libraries
  - Horrible ones I used
  - Horrible mistakes I made
  - Experience gained over the last 20 yrs…

General rules

- Direct access to variables
  - Do not use an index or any loop inside your code
- Const correctness
  - This helps the compiler optimize your code.
- Add specialized instructions
  - Rx*Ry*Rz if it happens a lot, make a custom function
- Add Intrinsics to everything
  - Keep your data 16 Byte aligned

General rules

- Adding implicit prevention
  - Last assignment stuff (private is your friend)
- Adding proxy for complicated functionality
  - You can distribute a library now
    - Optimize it later with proxies
- Testbed
  - Metrics your timing and assumptions
- Use only FLOATS
  - We went to the moon on 16-bit
  - 32-bits is enough for any conventional math system
    - Doubles SUCK! (slow)

Matrix

- Matrix
  - m x n array of numbers
  - m – rows
  - n – columns

3x2 matrix

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

3x3 matrix

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

2x4 matrix

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}$$

Matrix - add

- Matrix
  - A + B = B + A

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_1+b_1 & a_2+b_2 \\ a_3+b_3 & a_4+b_3 \end{bmatrix}$$

Same process for 3x3 or 3x4 or 4x4

Like positions are added together

Matrix - Subtract

- Matrix
  - A - B = -B + A

$$\mathbf{A} - \mathbf{B} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} - \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

$$\mathbf{A} - \mathbf{B} = \begin{bmatrix} a_1 - b_1 & a_2 - b_2 \\ a_3 - b_3 & a_4 - b_3 \end{bmatrix}$$

Same process for 3x3 or 3x4 or 4x4

Like positions are subtracted together

Matrix - position

- Positions

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \; 2x3\ matrix$$

$$a\ is\ (1,1) position$$
$$d\ is\ (2,1) position$$
$$c\ is\ (1,3) position$$

$$(row, col)\ position$$

DePaul University

Matrix - transpose

- Swap row and column positions

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \; 2x3\; matrix$$

$$c\ is\ (1{,}3) is\ now\ (3{,}1) \qquad A^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \; 3x2\; matrix$$

- Note:
  - T – super script, to indicate transpose
  - The rows and columns are swapped
  - The dimension of the matrix is swapped

Matrix – transpose 4x4

- Swap row and column positions

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \quad A^T = \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

- Swap columns and rows inside of matrix

Matrix - Multiplication

- Matrix 2x2

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \quad B = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

$$C = AB$$

$$C = \begin{bmatrix} a_1 b_1 + a_2 b_3 & a_1 b_2 + a_2 b_4 \\ a_3 b_1 + a_4 b_3 & a_3 b_2 + a_4 b_4 \end{bmatrix}$$

Matrix - Multiplication

- Matrix 2x2
  - Across then down

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \quad B = \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix}$$

$$C = AB$$

$$C = \begin{bmatrix} a_1 b_1 + a_2 b_3 & a_1 b_2 + a_2 b_4 \\ a_3 b_1 + a_4 b_3 & a_3 b_2 + a_4 b_4 \end{bmatrix}$$

Matrix - Multiplication

- Matrix 3x3

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \quad B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix}$$

$$C = AB$$

$$C = \begin{bmatrix} a_1b_1 + a_2b_4 + a_3b_7 & a_1b_2 + a_2b_5 + a_3b_8 & a_1b_3 + a_2b_6 + a_3b_9 \\ a_4b_1 + a_5b_4 + a_6b_7 & a_4b_2 + a_5b_5 + a_6b_8 & a_4b_3 + a_5b_6 + a_6b_9 \\ a_7b_1 + a_8b_4 + a_9b_7 & a_7b_2 + a_8b_5 + a_9b_8 & a_7b_3 + a_8b_6 + a_9b_9 \end{bmatrix}$$

Matrix – Multiplication pattern

- (dot product) Across then down

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix}$$

$$C = AB$$

$$C = \begin{bmatrix} a_1 b_1 + a_2 b_4 + a_3 b_7 & a_1 b_2 + a_2 b_5 + a_3 b_8 & a_1 b_3 + a_2 b_6 + a_3 b_9 \\ a_4 b_1 + a_5 b_4 + a_6 b_7 & a_4 b_2 + a_5 b_5 + a_6 b_8 & a_4 b_3 + a_5 b_6 + a_6 b_9 \\ a_7 b_1 + a_8 b_4 + a_9 b_7 & a_7 b_2 + a_8 b_5 + a_9 b_8 & a_7 b_3 + a_8 b_6 + a_9 b_9 \end{bmatrix}$$

Vector * Matrix

- Vector * matrix
  - Size of matrix needs to agreement
    - Inside dimensions only
      - (1 x n) vector
        - 1 – row, n –columns
        - Just a specialized matrix with 1 row, n columns
      - (n x m) matrix
        - n – rows, m – columns
    - **V_{out} = V * M**
      - Dimensionally
        - (1 x n) (n x m)
        - Inside are both ***n*** so it's allowed

DePaul University

Vector * Matrix

- Vector * Matrix
  - $V_{out} = V * M$

$$V_{out} = V * M$$

$$V_{out} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_9 \end{bmatrix}$$

$$V_{out} = \begin{bmatrix} v_x m_1 + v_y m_4 + v_z m_7 & v_x m_2 + v_y m_5 + v_z m_8 & v_x m_3 + v_y m_6 + v_z m_9 \end{bmatrix}$$

  - Row Major
    - Vector (1x4) ← that's a Row

Matrix * Vector

- Matrix * Vector
  - Size of matrix needs to agreement
    - Inside dimensions only
      - (m x n) matrix
        - m – rows, n – columns
      - (n x 1) vector
        - n – rows, 1 –column
        - Just a specialized matrix with 4 rows, 1 column
    - $\mathbf{V_{out} = M * V}$
      - Dimensionally
        - (m x n) (n x 1)
        - Inside are both ***n*** so it's allowed

Matrix * Vector

- Matrix * Vector
  - **V_{out} = M * V**

$$V_{out} = M * V$$

$$V_{out} = \begin{bmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_8 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

$$V_{out} = \begin{bmatrix} m_1 v_x + m_2 v_y + m_3 v_z \\ m_4 v_x + m_5 v_y + m_6 v_z \\ m_7 v_x + m_8 v_y + m_9 v_z \end{bmatrix}$$

  - Column Major
    - Vector (4x1) ← that's a Row

Row Major vs Column Major

- Vector * Matrix
  - If vector is (1 x n) size
    - It is a row vector
- Row major orientation
  - Matrices A, B, C are concatenated left to right
    - M = A*B*C
    - For row major (preferred for performanc)
- Column major orientation
  - A, B, C are concatenated Right to Left.
    - M = C * B * A
    - For column major (vectors that are nx1 – dimensions)

Row Major

- Graphics Cards
  - OpenGL or DirectX
    - Use Row major internally
- Physics engines
  - Use row major
- You should use row major
  - Be cool, use row major

4D vectors

- 4D Vectors
  - (1x4) dimensions
  - **V** = [x, y, z, 1]
- Why do we need them:
  - Matrix multiplication reasons
    - Future class, allows mixture of translation and rotation
    - Quaternions
    - Cache alignment

4D Row Vectors

- Since vectors are (1x4) dimension
  - Matrices need to be (4xn)
    - $V_{out} = V * M$
      - Dimensionally
        - (1 x n) (n x m)
        - Inside are both ***n*** so it's allowed
- All the math today can easily extended to 4D
  - Just follow the pattern
  - Exception is inverse, look that up.

DePaul University

Vector 1x4

- Vectors we use are 4 dimension homogeneous vectors
  - Components: X,Y,Z,W
- What is W?
  - It is an extension to make the matrix multiplies work for both rotations and translations.

How do you use a matrix?

- Matrices will allow points to be transformed from one location to another.
- Points are in the form of vectors
  - p = [x,y,z,w]
    - W = 1, for our discussion.
- Multiply a point with a matrix you transform the point.
  - p_out = p * M;

Translation

- Move points in space, along some vector.
- Points can be moved in the x,y,z direction.
  - Any direction, + or -, doesn't matter.
- Translation Matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

  - ***Tx, Ty, Tz*** is the translation direction applied

Using Translation Matrix

- Point
  - p = [x,y,z,w]
- Output
  - p_out = p * M;

$$p_{out} = p * T$$

$$p_{out} = \begin{bmatrix} 10 & 20 & 30 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 5 & 7 & 1 \end{bmatrix}$$

$$p_{out} = \begin{bmatrix} 12 & 25 & 37 & 1 \end{bmatrix}$$

- Example:
  - p = [10,20,30,1]
  - Translation = [2, 5, 7]

DePaul University

Very important (many points)

- In previous example we processed one point
  - You can process a collections of points
    - Did someone say array?

$$p_{out} = p * T$$

$$p_{out}[i] = p[i]\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\2 & 5 & 7 & 1\end{bmatrix}$$

  - Process a collection of points through the matrix.

DePaul University

Same for operations

- You can transform points by
  - Scaling,
  - Translation
  - Rotation
- Quick summary of transformations
  - Covering the next several slides
  - Stay with me… Your getting it.

Scale

- Scaling
  - You can scale in the x, y, z directions
  - You can scale uniformly or independently
- Scale Matrix:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations

- Rotations can be rotate points around an axis.
- Many rotations can be thought of a series of individual rotations.
  - Rx – rotation about the x-axis
  - Ry – rotation about the y-axis
  - Rz – rotation about the z-axis
- Rotations can be about an arbitrary axis.
  - Not in this class (Quaternions are better)

Rotation Matrices, Rx, Ry, Rz

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Putting it all together

- A matrix can transform points
- You can scale, rotate or translate points
- If you do several operations together you can concatenate matrices!
  - That the big take away.
- Concatenating matrices saves
  - times and operations

Transform Example:

- Take a bag of points (array)
  - Scale the points
  - Rotate the points
  - Translate points
- Piece meal you would:

$$p_{out_A}[i] = p[i] * S$$
$$p_{out_B}[i] = p_{out_A}[i] * R$$
$$p_{out_C}[i] = p_{out_B}[i] * T$$

- 3 separate transformations
- A lot of multiplies, and adds… we can do better

Continued:

- One matrix can represent all those transformations
  - Scale, then Rotate, then translate the points
    - $S \rightarrow R \rightarrow T$
  - One concatenated matrix can represent all
    - $M = S*R*T$

Answer to the Universe…

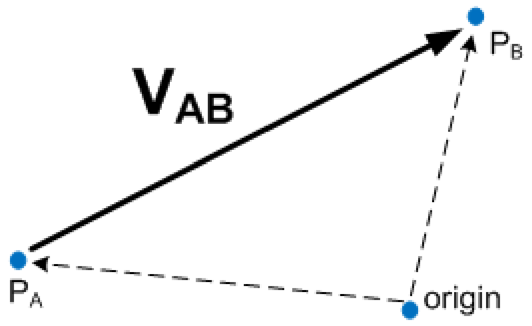
- Not exactly, but close
  - Use 1x4 vectors to allow the concatenation to happen mathematically.
  - Otherwise - need to stage transform our points.
    - Very wasteful and slow.
  - Points are in [x,y,z,w]
    - W is 1 for this class
      - Physics
    - In Computer Graphics it has other uses
      - Perspective Correction.

DePaul University

Form a vector from 2 points

- Vector has a magnitude and a direction
  - Vectors can be formed from 2 pts
- Simply subtract the points.
  - 1st pt is the head of the vector
  - 2nd pt is the tail of the vector
  - V_{AB} forms a vector from A to B

$$P_A = [a_x \quad a_y \quad a_z \quad 1]$$
$$P_B = [b_x \quad b_y \quad b_z \quad 1]$$



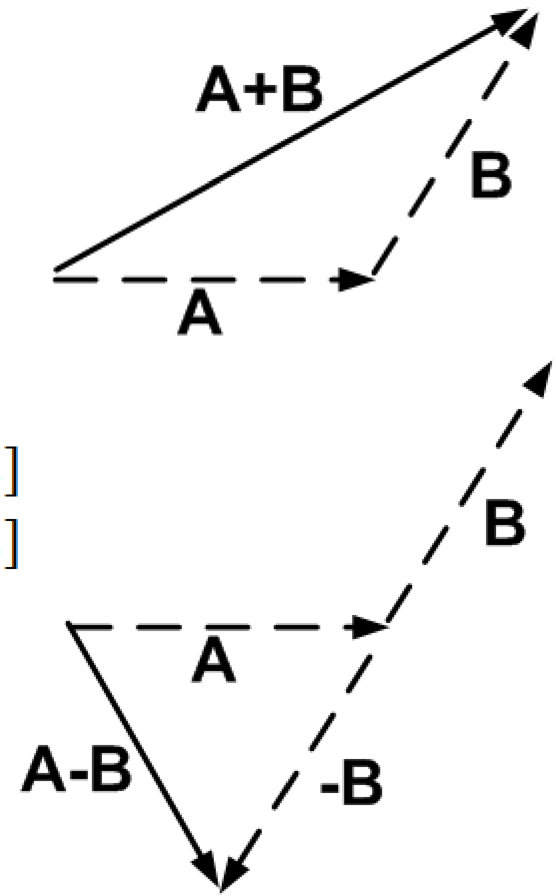
$$V_{AB} = P_B - P_A = [b_x - a_x \quad b_y - a_y \quad b_z - a_z \quad 1]$$

DePaul University

1x4 Vector math

- Vectors A & B

$$A = [a_x \quad a_y \quad a_z \quad 1]$$
$$B = [b_x \quad b_y \quad b_z \quad 1]$$



- Add, Subtract

$$A + B = [a_x + b_x \quad a_y + b_y \quad a_z + b_z \quad 1]$$
$$A - B = [a_x - b_x \quad a_y - b_y \quad a_z - b_z \quad 1]$$

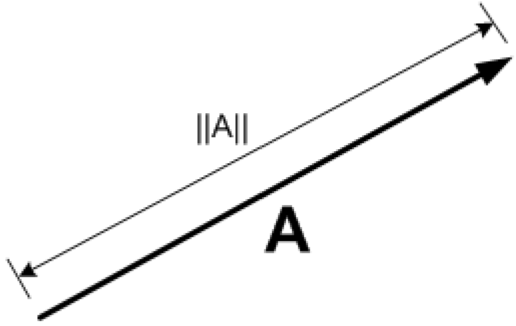
DePaul University

1x4 Magnitude

- Magnitude (length)

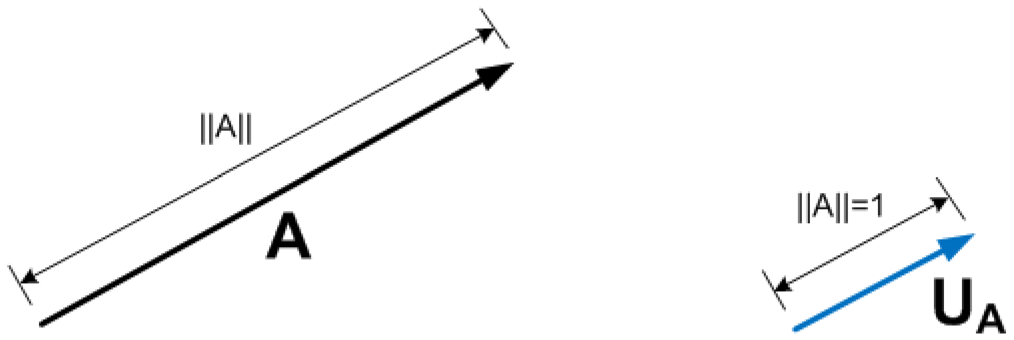
$$A = [a_x \quad a_y \quad a_z \quad 1]$$

$$||A|| = \sqrt{(a_x{}^2 + a_y{}^2 + a_z{}^2)}$$



Normalize

- Change the vector to length of 1



$$A = \begin{bmatrix} a_x & a_y & a_z & 1 \end{bmatrix}$$

$$||A|| = \sqrt{\left(a_x{}^2 + a_y{}^2 + a_z{}^2\right)}$$

$$U_A = \begin{bmatrix} \frac{a_x}{||A||} & \frac{a_y}{||A||} & \frac{a_z}{||A||} & 1 \end{bmatrix}$$

DePaul University

Dot product

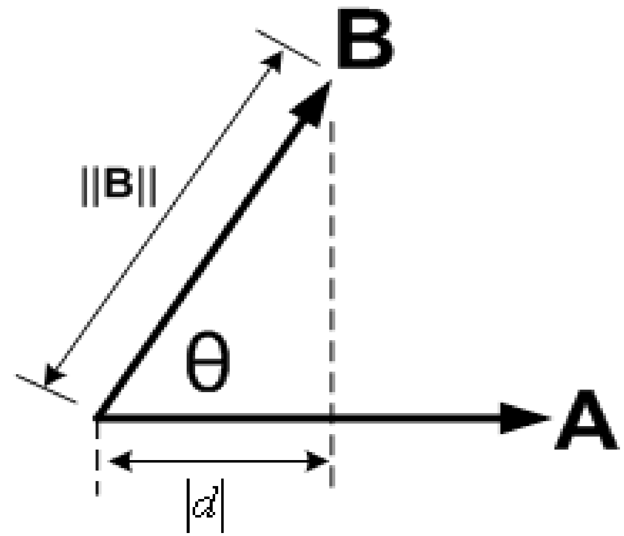
- Dot product returns a scalar.
  - Single float (not a vector)

$$A = [a_x \quad a_y \quad a_z \quad 1]$$
$$B = [b_x \quad b_y \quad b_z \quad 1]$$

$$A \cdot B = \left(a_x b_x + a_y b_y + a_z b_z\right)$$

  - Or if you know the angle between the vectors
    - Not recommended

$$A \cdot B = \|A\| \|B\| \cos\theta$$



$$|d| = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|} = \|\mathbf{B}\| \cos\theta$$

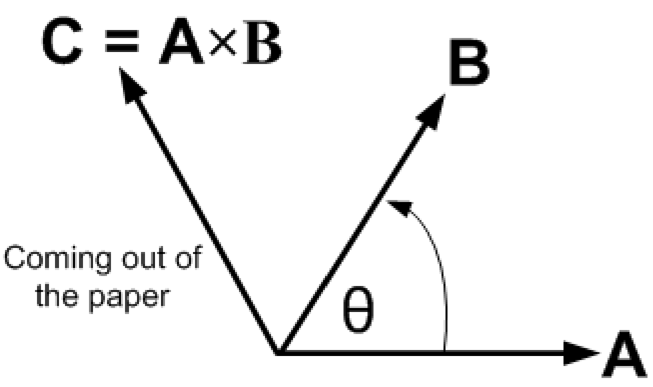
Cross Product

- Cross product returns a vector

$$A \times B = \begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

$$A \times B = [a_y b_z - a_z b_y \quad -(a_x b_z - a_z b_x) \quad a_x b_y - a_y b_x \quad 1]$$

$$A = [a_x \quad a_y \quad a_z \quad 1]$$
$$B = [b_x \quad b_y \quad b_z \quad 1]$$



- Trig form:

$$A \times B = \|A\|\|B\| \sin\theta_{\hat{K}}$$

Matrix 4x4

- All the math we will do will be 4x4 Matrices
- It allows concatenation of different types of transforms.
  - It's Cool and Hip!

Example matrices

- Assume we are using these matrices for the rest of the examples:

$$M = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \quad N = \begin{bmatrix} n_0 & n_1 & n_2 & n_3 \\ n_4 & n_5 & n_6 & n_7 \\ n_8 & n_9 & n_{10} & n_{11} \\ n_{12} & n_{13} & n_{14} & n_{15} \end{bmatrix}$$

DePaul University

Matrix Add / Subtraction

- Matrix Add

$$M + N = \begin{bmatrix} m_0 + n_0 & m_1 + n_1 & m_2 + n_2 & m_3 + n_3 \\ m_4 + n_4 & m_5 + n_5 & m_6 + n_6 & m_7 + n_7 \\ m_8 + n_8 & m_9 + n_9 & m_{10} + n_{10} & m_{11} + n_{11} \\ m_{12} + n_{12} & m_{13} + n_{13} & m_{14} + n_{14} & m_{15} + n_{15} \end{bmatrix}$$

- Matrix Sub

$$M - N = \begin{bmatrix} m_0 - n_0 & m_1 - n_1 & m_2 - n_2 & m_3 - n_3 \\ m_4 - n_4 & m_5 - n_5 & m_6 - n_6 & m_7 - n_7 \\ m_8 - n_8 & m_9 - n_9 & m_{10} - n_{10} & m_{11} - n_{11} \\ m_{12} - n_{12} & m_{13} - n_{13} & m_{14} - n_{14} & m_{15} - n_{15} \end{bmatrix}$$

Matrix Scale

- Matrix Scale by a scalar

$$sM = \begin{bmatrix} sm_0 & sm_1 & sm_2 & sm_3 \\ sm_4 & sm_5 & sm_6 & sm_7 \\ sm_8 & sm_9 & sm_{10} & sm_{11} \\ sm_{12} & sm_{13} & sm_{14} & sm_{15} \end{bmatrix}$$

Matrix Multiply

$$R = MN = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \begin{bmatrix} n_0 & n_1 & n_2 & n_3 \\ n_4 & n_5 & n_6 & n_7 \\ n_8 & n_9 & n_{10} & n_{11} \\ n_{12} & n_{13} & n_{14} & n_{15} \end{bmatrix}$$

$$\begin{aligned}
r_0 &= m_0n_0 + m_1n_4 + m_2n_8 + m_3n_{12} \\
r_1 &= m_0n_1 + m_1n_5 + m_2n_9 + m_3n_{13} \\
r_2 &= m_0n_2 + m_1n_6 + m_2n_{10} + m_3n_{14} \\
r_3 &= m_0n_3 + m_1n_7 + m_2n_{11} + m_3n_{15} \\
r_4 &= m_4n_0 + m_5n_4 + m_6n_8 + m_7n_{12} \\
r_5 &= m_4n_1 + m_5n_5 + m_6n_9 + m_7n_{13} \\
r_6 &= m_4n_2 + m_5n_6 + m_6n_{10} + m_7n_{14} \\
r_7 &= m_4n_3 + m_5n_7 + m_6n_{11} + m_7n_{15}
\end{aligned}$$

$$\begin{aligned}
r_8 &= m_8n_0 + m_9n_4 + m_{10}n_8 + m_{11}n_{12} \\
r_9 &= m_8n_1 + m_9n_5 + m_{10}n_9 + m_{11}n_{13} \\
r_{10} &= m_8n_2 + m_9n_6 + m_{10}n_{10} + m_{11}n_{14} \\
r_{11} &= m_8n_3 + m_9n_7 + m_{10}n_{11} + m_{11}n_{15} \\
r_{12} &= m_{12}n_0 + m_{13}n_4 + m_{14}n_8 + m_{15}n_{12} \\
r_{13} &= m_{12}n_1 + m_{13}n_5 + m_{14}n_9 + m_{15}n_{13} \\
r_{14} &= m_{12}n_2 + m_{13}n_6 + m_{14}n_{10} + m_{15}n_{14} \\
r_{15} &= m_{12}n_3 + m_{13}n_7 + m_{14}n_{11} + m_{15}n_{15}
\end{aligned}$$

Questions?