

```
ics
& (depth < MAXDEPTH)
{
    if (nt < 0)
        nt = inside ? 1 : -1;
    nt = nt / nc; ddn = dot(N, R);
    cos2t = 1.0f - nnt * nnt;
    D, N );
    )
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * sqrt(f));
        Tr) R = (D * nnt - N * (ddn < 0 ? 1 : -1));
    }
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    {
        survive = SurvivalProbability( diffuse );
        estimation - doing it properly, closely following
        if;
        radiance = SampleLight( &rand, I, &L, &lightPdf );
        e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
        {
            w = true;
            at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
            at3 factor = diffuse * INVPI;
            at weight = Mis2( directPdf, brdfPdf );
            at cosThetaOut = dot( N, L );
            E * ((weight * cosThetaOut) / directPdf) * (radiance
        }
        random walk - done properly, closely following Small's
        (survive)
    }
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

# 4

# Ray Tracing for Games

Dr. Jacco Bikker - IGAD/BUAS, Breda, January 31

# Welcome!



# Ray Tracing for Games

Thursday  
09:00 – 14:00

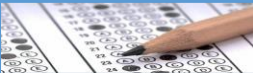
advanced Whitted  
audio, AI & physics  
faster Whitted  
Heaven7

Wednesday  
13:00 – 17:00

course intro  
LH2  
template  
Whitted  
refactoring  
RT-centric games

LAB 1

LAB 2



work @ home

End result day 2:

A solid Whitted-style  
ray tracer, as a basis  
for subsequent work.

Friday  
09:00 – 17:00

optimization  
profiling, rules of  
engagement  
threading

LAB



SIMD  
applied SIMD  
SIMD triangle  
SIMD AABB

LAB 4

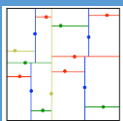
End result day 3:  
A 5x faster tracer.

## GLOBAL GAME JAM

Monday  
09:00 – 17:00

acceleration  
grid, BVH, kD-tree  
SAH  
binning

LAB 5



refitting  
top-level BVH  
threaded building

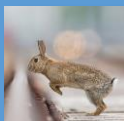
LAB 6

End result day 4:  
A real-time tracer.

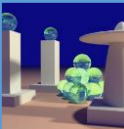
Tuesday  
09:00 – 17:00

Monte-Carlo  
Cook-style  
glossy, AA  
area lights, DOF

LAB 7



path tracing



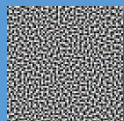
LAB 8

End result day 5:  
Cook or Kajiya.

Thursday  
09:00 – 17:00

random numbers  
stratification  
blue noise

LAB 9



importance  
sampling  
next event  
estimation

LAB 10

End result day 6:  
Efficiency.

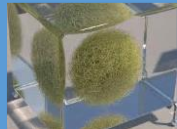
Friday  
09:00 – 17:00

future work

LAB 11



path guiding



LAB 10



End result day 6:  
Great product.

## Agenda:

- Introduction
- SSE / AVX
- Streams
- Vectorization

```
ics
& (depth < MAXDEPTH))
{
    if (nt < 0)
        nt = inside ? 1 : 1.01;
    nt = nt / nc, ddn = ddn * nt;
    float r2t = 1.0f - nnt * nnt;
    float r0, N );
    r0 = 0;
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * r2t);
Tr) R = (D * nnt - N * (ddn < 0 ? 1 : -1));

E * diffuse;
= true;

-
refl + refr)) && (depth < MAXDEPTH))
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

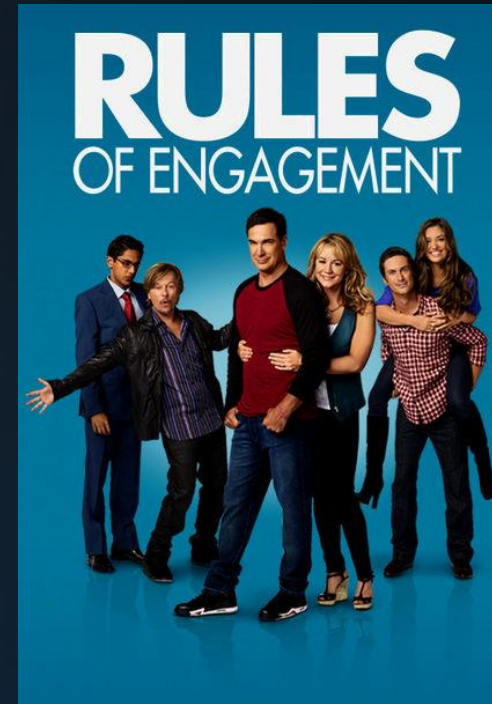
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following Smallman
df;
radiance = SampleLight( &rand, I, &L, &lightPdf );
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH))
{
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH))
{
    random walk - done properly, closely following Smallman
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
```



## Consistent Approach

(0.) Determine optimization requirements

1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 7 and 8 until time runs out
9. Report.



## Rules of Engagement

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two
7. Do Things Simultaneously

## S.I.M.D.

Single Instruction Multiple Data:

*Applying the same instruction to several input elements.*

In other words: if we are going to apply the same sequence of instructions to a large input set, this allows us to do this in parallel (and thus: faster).

SIMD is also known as *instruction level parallelism*.

Examples:

```
union { uint a4; unsigned char a[4]; };
do
{
    GetFourRandomValues( a );
}
while (a4 != 0);
```

```
unsigned char a[4] = { 1, 2, 3, 4 };
unsigned char b[4] = { 5, 5, 5, 5 };
unsigned char c[4];
*(uint*)c = *(uint*)a + *(uint*)b;
// c is now { 6, 7, 8, 9 }.
```





```
void Game::Tick( float deltaTime )
```

```
{
```

```
00000000140002C40  movss      dword ptr [rsp+10h],xmm1
00000000140002C46  mov        qword ptr [rsp+8],rcx
00000000140002C4B  push      rdi
00000000140002C4C  sub        rsp,90h
00000000140002C53  mov        rdi,rsp
00000000140002C56  mov        ecx,24h
00000000140002C5B  mov        eax,0CCCCCCCCh
00000000140002C60  rep stos   dword ptr [rdi]
00000000140002C62  mov        rcx,qword ptr [this]
    unsigned char a[4] = { 1, 2, 3, 4 };
00000000140002C6A  mov        byte ptr [a],1
00000000140002C6F  mov        byte ptr [rsp+35h],2
00000000140002C74  mov        byte ptr [rsp+36h],3
00000000140002C79  mov        byte ptr [rsp+37h],4
    unsigned char b[4] = { 5, 5, 5, 5 };
00000000140002C7E  mov        byte ptr [b],5
00000000140002C83  mov        byte ptr [rsp+55h],5
00000000140002C88  mov        byte ptr [rsp+56h],5
00000000140002C8D  mov        byte ptr [rsp+57h],5
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
00000000140002C92  mov        eax,dword ptr [b]
00000000140002C96  mov        ecx,dword ptr [a]
00000000140002C9A  add        ecx,eax
00000000140002C9C  mov        eax,ecx
00000000140002C9E  mov        dword ptr [c],eax
```

out

same  
, this  
ster).

Examples:

```
union { uint a4; unsigned char a[4]; };
do
{
    GetFourRandomValues( a );
}
while (a4 != 0);
```

```
unsigned char a[4] = { 1, 2, 3, 4 };
unsigned char b[4] = { 5, 5, 5, 5 };
unsigned char c[4];
*(uint*)c = *(uint*)a + *(uint*)b;
// c is now { 6, 7, 8, 9 }.
```

```
void Game::Tick( float deltaTime )
```

```
{
0000000140002250  movss      dword ptr [rsp+10h],xmm1
0000000140002256  mov        qword ptr [rsp+8],rcx
000000014000225B  sub        rsp,38h
    unsigned char a[4] = { 1, 2, 3, 4 };
    unsigned char b[4] = { 5, 5, 5, 5 };
000000014000225F  mov        dword ptr [rsp+40h],5050505h
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
0000000140002267  mov        edx,dword ptr [b]
000000014000226B  mov        dword ptr [rsp+48h],4030201h
0000000140002273  add        edx,dword ptr [a]
0000000140002277  mov        ecx,edx
0000000140002279  mov        eax,edx
```

allows us to do this in parallel (and thus: faster).

SIMD is also known as *instruction level parallelism*.

Examples:

```
union { uint a4; unsigned char a[4]; };
do
{
    GetFourRandomValues( a );
}
while (a4 != 0);
```

```
unsigned char a[4] = { 1, 2, 3, 4 };
unsigned char b[4] = { 5, 5, 5, 5 };
unsigned char c[4];
*(uint*)c = *(uint*)a + *(uint*)b;
// c is now { 6, 7, 8, 9 }.
```

# Ray Tracing for Games

uint = unsigned char[4]

Pinging google.com yields: 74.125.136.101

Each value is an unsigned 8-bit value (0..255).

Combing them in one 32-bit integer:

101 +  
256 \* 136 +  
256 \* 256 \* 125 +  
256 \* 256 \* 256 \* 74 = 1249740901.

Browse to: <http://1249740901> (works!)

Evil use of this:

We can specify a user name when visiting a website, but any username will be accepted by google. Like this:

<http://infomov@google.com>

Or:

<http://www.ing.nl@1249740901>

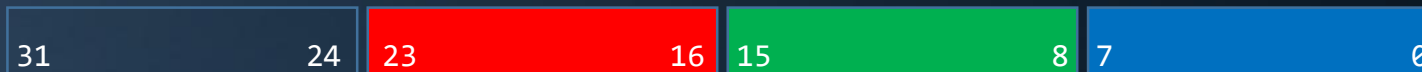
Replace the IP address used here by your own site which contains a copy of the ing.nl site to obtain passwords, and send the link to a 'friend'.





## Example: color scaling

Assume we represent colors as 32-bit ARGB values using unsigned ints:



To scale this color by a specified percentage, we use the following code:

```
uint ScaleColor( uint c, float x ) // x = 0..1
{
    uint red = (c >> 16) & 255;
    uint green = (c >> 8) & 255;
    uint blue = c & 255;
    red = red * x, green = green * x, blue = blue * x;
    return (red << 16) + (green << 8) + blue;
}
```



|    |    |    |    |    |   |   |   |
|----|----|----|----|----|---|---|---|
| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
|----|----|----|----|----|---|---|---|

## Example: color scaling

```
uint ScaleColor( uint c, float x ) // x = 0..1
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = red * x, green = green * x, blue = blue * x;
    return (red << 16) + (green << 8) + blue;
}
```

## Improved:

```
uint ScaleColor( uint c, uint x ) // x = 0..255
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8;
    green = (green * x) >> 8;
    blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```

# Ray Tracing for Games

Example: color scaling

|    |    |    |    |    |   |   |   |
|----|----|----|----|----|---|---|---|
| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
| 31 | 24 | 23 | 16 | 15 | 8 | 7 |   |

```
uint ScaleColor( uint c, uint x ) // x = 0..255
```

```
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8, green = (green * x) >> 8, blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```

*7 shifts, 3 ands, 3 muls, 2 adds*

Improved:

```
uint ScaleColor( const uint c, const uint x ) // x = 0..255
```

```
{
    uint redblue = c & 0x00FF00FF;
    uint green    = c & 0x0000FF00;
    redblue = ((redblue * x) >> 8) & 0x00FF00FF;
    green = ((green * x) >> 8) & 0x0000FF00;
    return redblue + green;
}
```

*2 shifts, 4 ands, 2 muls, 1 add*

# Ray Tracing for Games

Example: color scaling

|    |    |    |    |    |   |   |   |
|----|----|----|----|----|---|---|---|
| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
| 31 | 24 | 23 | 16 | 15 | 8 | 7 |   |

```
uint ScaleColor( uint c, uint x ) // x = 0..255
```

```
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8, green = (green * x) >> 8, blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```

*7 shifts, 3 ands, 3 muls, 2 adds  
(15 ops)*

Further improved:

```
uint ScaleColor( const uint c, const uint x ) // x = 0..255
```

```
{
    uint redblue = c & 0x00FF00FF;
    uint green    = c & 0x0000FF00;
    redblue = (redblue * x) & 0xFF00FF00;
    green = (green * x) & 0x00FF0000;
    return (redblue + green) >> 8;
}
```

*1 shift, 4 ands, 2 muls, 1 add  
(8 ops)*

## Other Examples

### Rapid string comparison:

```
char a[] = "optimization skills rule";
char b[] = "optimization is so nice!";
bool equal = true;
int l = strlen( a );
for ( int i = 0; i < l; i++ )
{
    if (a[i] != b[i])
    {
        equal = false;
        break;
    }
}
```

Likewise, we can copy byte arrays faster.

```
char a[] = "optimization skills rule";
char b[] = "optimization is so nice!";
bool equal = true;
int q = strlen( a ) / 4;
for ( int i = 0; i < q; i++ )
{
    if (((int*)a)[i] != ((int*)b)[i])
    {
        equal = false;
        break;
    }
}
```

## SIMD using 32-bit values - Limitations

Mapping four chars to an int value has a number of limitations:

$$\{ 100, 100, 100, 100 \} + \{ 1, 1, 1, 200 \} = \{ 101, 101, 102, 44 \}$$

$$\{ 100, 100, 100, 100 \} * \{ 2, 2, 2, 2 \} = \{ \dots \}$$

$$\{ 100, 100, 100, 200 \} * 2 = \{ 200, 200, 201, 144 \}$$

In general:

- Streams are not separated (prone to overflow into next stream);
- Limited to small unsigned integer values;
- Hard to do multiplication / division.

```
...ics
& (depth < MAXDEPTH))
{
    if (nt > 0)
    {
        nt = inside ? 1 + 1.25 * nt : 1 + 0.75 * nt;
        nt = nt / nc; ddn = ddn * nt;
        ps2t = 1.0f - nnt * nnt;
        D, N );
    }
}
```

```
at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * nt);
Tr) R = (D * nnt - N * (ddn
```

```
E * diffuse;
= true;
```

```
efl + refr)) && (depth < MAXDEPTH))
```

```
D, N );
refl * E * diffuse;
= true;
```

```
MAXDEPTH)
```

```
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely
```

```
if;
```

```
radiance = SampleLight( &rand, I, &L, &lightDir;
```

```
e.x + radiance.y + radiance.z) > 0) && (acc
```

```
w = true;
```

```
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
```

```
at3 factor = diffuse * INVPI;
```

```
at weight = Mis2( directPdf, brdfPdf );
```

```
at cosThetaOut = dot( N, L );
```

```
E * ((weight * cosThetaOut) / directPdf) * (radiant
```

```
random walk - done properly, closely following Small's
```

```
survive)
```

```
;
```

```
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
```

```
survive;
```

```
pdf;
```

```
n = E * brdf * (dot( N, R ) / pdf);
```

```
sion = true;
```



# Ray Tracing for Games

## SIMD using 32-bit values - Limitations

Ideally, we would like to see:

- Isolated streams
- Support for more data types (char, short, uint, int, float, double)
- An easy to use approach

# Meet SSE!

```
survive = SurvivalProbability( diffuse, L, N );  
// estimation - doing it properly, closely following Seal's  
// paper  
if;  
radiance = SampleLight( &rand, I, &L, &lights, &N );  
r.x + radiance.y + radiance.z ) > 0) && (dot( N,  
w = true;  
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;  
at3 factor = diffuse * INVP1;  
at weight = MixS2( directPdf, brdfPdf );  
at cosThetaOut = dot( N, L );  
E * ((weight * cosThetaOut) / directPdf) * (radiance
```

random walk - done properly, closely following Seal's  
live)

```
mat3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sign = true;
```

## Agenda:

- Introduction
- SSE / AVX
- Streams
- Vectorization

```
ics
& (depth < MAXDEPTH)
{
    // Inside or outside?
    bool t = inside ? 1 : -1;
    Vec nt = nt / nc, ddn = dot(N, D);
    double nnt = nnt * ddn, nnt2 = 1.0f - nnt * nnt;
    Vec D, N );
    Vec R = (D * nnt - N * (ddn < 0 ? 1 : -1));
    Vec E * diffuse;
    bool refl = true;
    Vec refl + refr)) && (depth < MAXDEPTH)
    {
        Vec D, N );
        Vec refl * E * diffuse;
        bool refl = true;
    }
    MAXDEPTH)
{
    survive = SurvivalProbability( diffuse );
    // estimation - doing it properly, closely following Small's
    if(
    {
        radiance = SampleLight( &rand, I, &L, &lightPdf );
        Vec e.x + radiance.y + radiance.z ) > 0) && (depth <
    {
        w = true;
        Vec brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        Vec3 factor = diffuse * INVPI;
        Vec weight = Mis2( directPdf, brdfPdf );
        Vec cosThetaOut = dot( N, L );
        Vec E * ((weight * cosThetaOut) / directPdf) * (radiance
    }
    // random walk - done properly, closely following Small's
    survive)
    {
        Vec3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        survive;
        Vec pdf;
        Vec n = E * brdf * (dot( N, R ) / pdf);
        // estimation = true;
    }
}
```



## A Brief History of SIMD

Early use of SIMD was in vector supercomputers such as the CDC Star-100 and TI ASC (image).

Intel's MMX extension to the x86 instruction set (1996) was the first use of SIMD in commodity hardware, followed by Motorola's AltiVec (1998), and Intel's SSE (P3, 1999).

SSE:

- 70 assembler instructions
- Operates on 128-bit registers
- Operates on vectors of 4 floats.



# Ray Tracing for Games

## SIMD Basics

C++ supports a 128-bit vector data type: `__m128`  
Henceforth, we will pronounce to this as ‘quadfloat’. ☺

`__m128` literally is a small array of floats:

```
union { __m128 a4; float a[4]; };
```

Alternatively, you can use the integer variety `__m128i`:

```
union { __m128i a4; int a[4]; };
```

# Ray Tracing for Games

## SIMD Basics

We operate on SSE data using *intrinsics*: in the case of SSE, these are keywords that translate to a single assembler instruction.

Examples:

```
__m128 a4 = _mm_set_ps( 1, 0, 3.141592f, 9.5f );
__m128 b4 = _mm_setzero_ps();
__m128 c4 = _mm_add_ps( a4, b4 ); // not: __m128 = a4 + b4;
__m128 d4 = _mm_sub_ps( b4, a4 );
```

Here, ‘\_ps’ stands for *packed scalar*.



## SIMD Basics

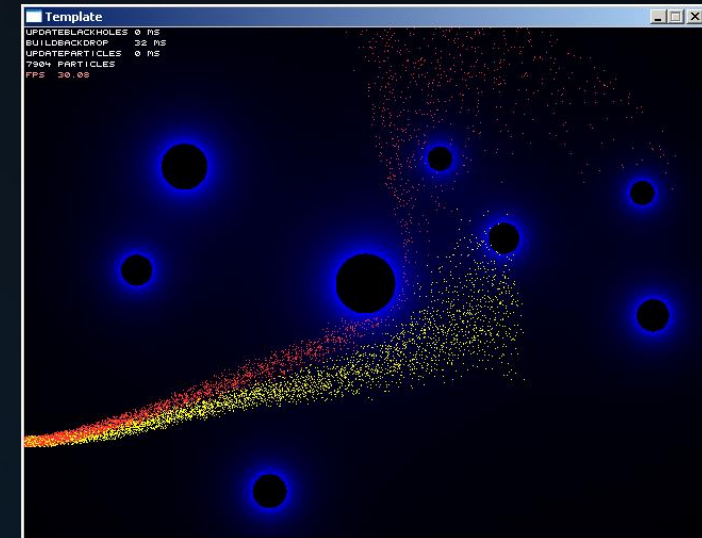
### Other instructions:

```
__m128 c4 = _mm_div_ps( a4, b4 ); // component-wise division
__m128 d4 = _mm_sqrt_ps( a4 );    // four square roots
__m128 d4 = _mm_rcp_ps( a4 );      // four reciprocals
__m128 d4 = _mm_rsqrt_ps( a4 );   // four reciprocal square roots (!)

__m128 d4 = _mm_max_ps( a4, b4 );
__m128 d4 = _mm_min_ps( a4, b4 );
```

Keep the assembler-like syntax in mind:

```
__m128 d4 = dx4 * dx4 + dy4 * dy4;
__m128 d4 = _mm_add_ps(
    _mm_mul_ps( dx4, dx4 ),
    _mm_mul_ps( dy4, dy4 )
);
```



# CODING TIME

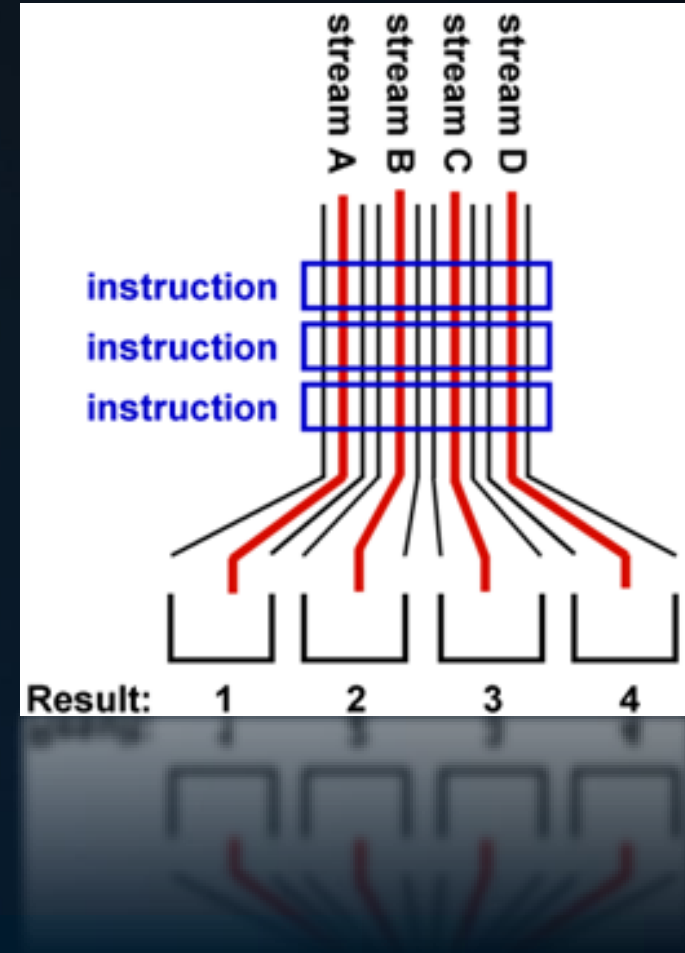


## SIMD Basics

In short:

- Four times the work at the price of a single scalar operation (if you can feed the data fast enough)
- Potentially even better performance for min, max, sqrt, rsqrt
- Requires four independent streams.

And, with AVX we get `__m256...`



## Agenda:

- Introduction
- SSE / AVX
- Streams
- Vectorization

```
ics
& (depth < MAXDEPTH))
{
    if (nt < 0)
        nt = inside ? 1 : 1.25;
    nt = nt / nc; ddn = ddn * nt;
    rnt2t = 1.0f - nnt * nnt;
    D, N );
    )
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * rnt2t);
        Tr) R = (D * nnt - N * (ddn < 0) ? 1 : -1);
        E * diffuse;
        = true;
        -
        refl + refr)) && (depth < MAXDEPTH))
        {
            D, N );
            refl * E * diffuse;
            = true;
        }
    }
    MAXDEPTH)
    {
        survive = SurvivalProbability( diffuse );
        estimation - doing it properly, closely following Smallman
        if;
        radiance = SampleLight( &rand, I, &L, &lightDir );
        e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH))
        {
            w = true;
            at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
            at3 factor = diffuse * INVPI;
            at weight = Mis2( directPdf, brdfPdf );
            at cosThetaOut = dot( N, L );
            E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH))
        }
        random walk - done properly, closely following Smallman
        (survive)
        {
            ;
            at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
            survive;
            pdf;
            n = E * brdf * (dot( N, R ) / pdf);
            sion = true;
        }
    }
}
```

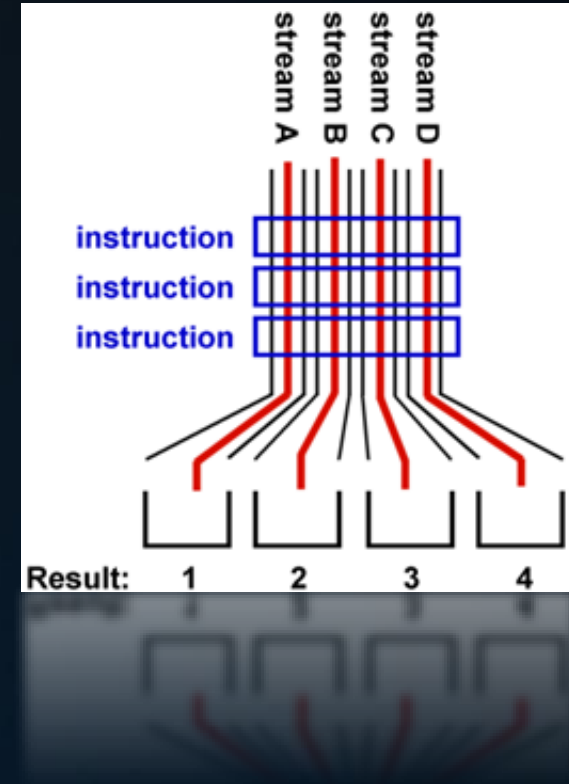


## SIMD According To Visual Studio

```
vec3 A( 1, 0, 0 );  
vec3 B( 0, 1, 0 );  
vec3 C = (A + B) * 0.1f;  
vec3 D = normalize( C );
```

The compiler will notice that we are operating on 3-component vectors, and it will use SSE instructions to speed up the code. This results in a modest speedup. Note that one lane is never used at all.

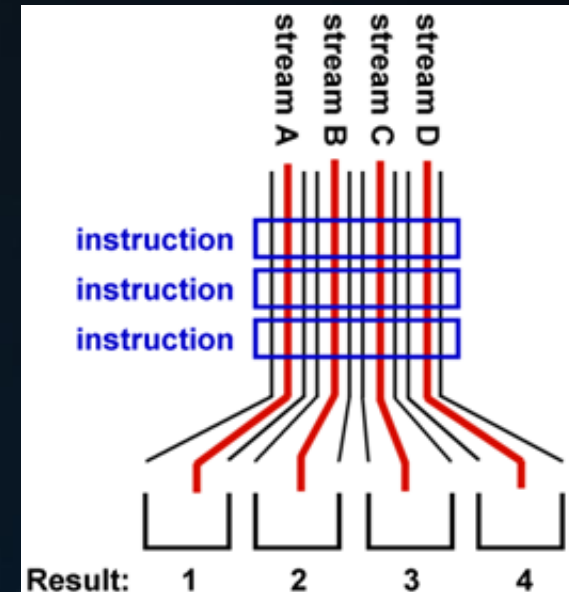
To get maximum throughput, we want four independent streams, running in parallel.



## SIMD According To Visual Studio

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nt;
        float2t = 1.0f - nnt * ddn;
        D, N );
    }
    if (a = nt - nc, b = nt - nc)
    {
        Tr = 1 - (R0 + (1 - R0) * t);
        R = (D * nnt - N * (ddn * t));
    }
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &lightPdf );
    e.x + radiance.y + radiance.z > 0) && (depth < MAXDEPTH)
    {
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
    }
    random walk - done properly, closely following Small's
    (survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

```
float Ax = 1, Ay = 0, Az = 0;
float Bx = 0, By = 1, Bz = 0;
float Cx = (Ax + Bx) * 0.1f;
float Cy = (Ay + By) * 0.1f;
float Cz = (Az + Bz) * 0.1f;
float l = sqrtf( Cx * Cx + Cy * Cy + Cz * Cz );
float Dx = Cx / l;
float Dy = Cy / l;
float Dz = Cz / l;
```

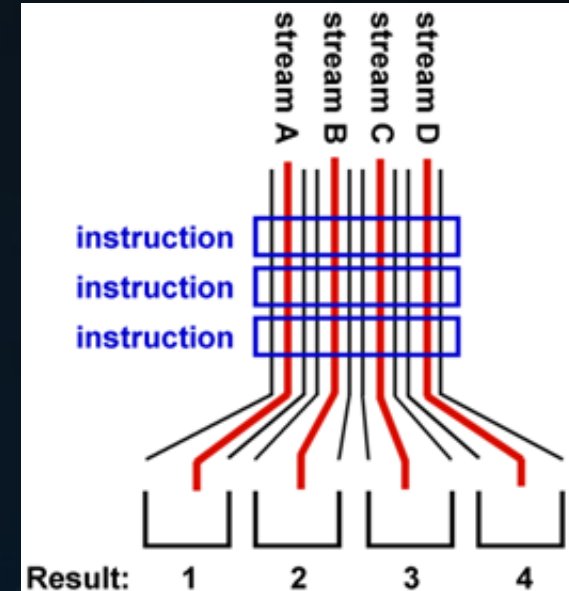


# Ray Tracing for Games

## SIMD According To Visual Studio

```
ics
& (depth < MAXDEPTH)
{
    t = inside ? 1.0f : 0.0f;
    nt = nt / nc; ddn = dot(n, d);
    cos2t = 1.0f - nnt * ddn;
    D, N );
    0)
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    -
    efl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    df;
    radiance = SampleLight( &rand, I, &L, &lightPos,
    e.x + radiance.y + radiance.z ) > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Small
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

```
float Ax[4] = {...}, Ay[4] = {...}, Az[4] = {...};
float Bx[4] = {...}, By[4] = {...}, Bz[4] = {...};
float Cx[4] = ...;
float Cy[4] = ...;
float Cz[4] = ...;
float l[4] = ...;
float Dx[4] = ...;
float Dy[4] = ...;
float Dz[4] = ...;
```



## SIMD According To Visual Studio

```
ics  
& (depth < MAXDEPTH)  
  
t = inside ? 1.0f : 0.0f;  
nt = nt / nc; ddn = ddn * t;  
ps2t = 1.0f - nnt * ddn;  
D, N );  
0)
```

```
at a = nt - nc, b = nt + nc;  
at Tr = 1 - (R0 + (1 - R0) * t);  
Tr) R = (D * nnt - N * (ddn
```

```
E * diffuse;  
= true;
```

```
efl + refr)) && (depth < MAXDEPTH)
```

```
D, N );  
refl * E * diffuse;  
= true;
```

```
MAXDEPTH)
```

```
survive = SurvivalProbability( diffuse );  
estimation - doing it properly, closely following  
df;
```

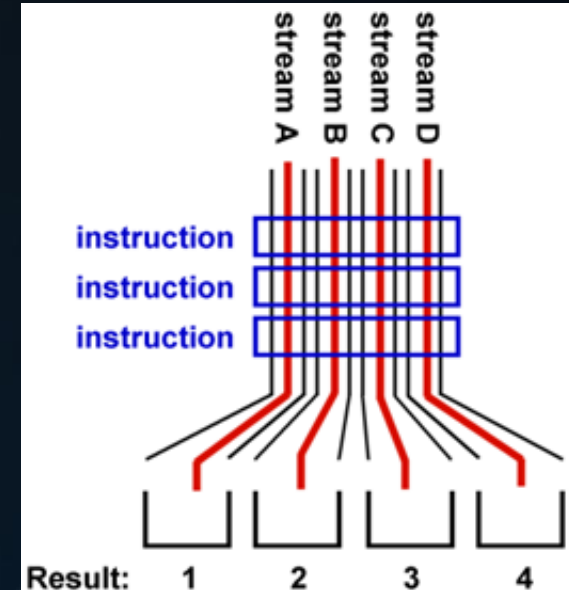
```
radiance = SampleLight( &rand, I, &L, &lightPdf );  
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
```

```
w = true;  
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;  
at3 factor = diffuse * INVPI;  
at weight = Mis2( directPdf, brdfPdf );  
at cosThetaOut = dot( N, L );  
E * ((weight * cosThetaOut) / directPdf) * (radiance
```

```
random walk - done properly, closely following Small's  
ive)
```

```
;  
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );  
survive;  
pdf;  
n = E * brdf * (dot( N, R ) / pdf);  
ion = true;
```

```
__m128 Ax4 = {...}, Ay4 = {...}, Az4 = {...};  
__m128 Bx4 = {...}, By4 = {...}, Bz4 = {...};  
__m128 Cx4 = ...;  
__m128 Cy4 = ...;  
__m128 Cz4 = ...;  
__m128 l4 = ...;  
__m128 Dx4 = ...;  
__m128 Dy4 = ...;  
__m128 Dz4 = ...;
```



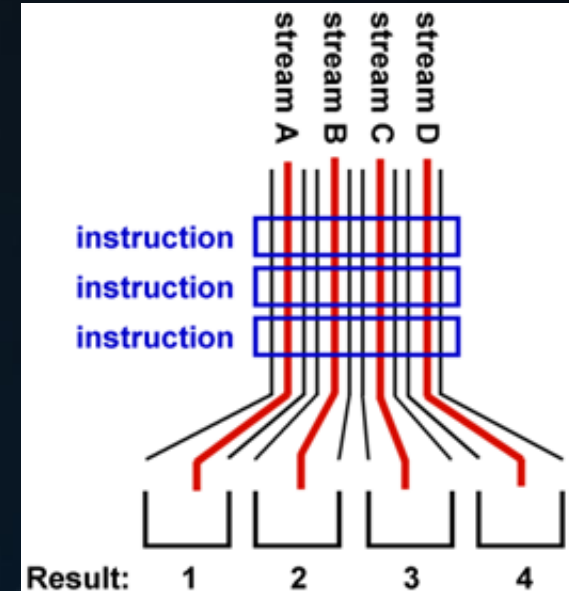


## SIMD According To Visual Studio

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nt;
        r2t = 1.0f - nnt * nnt;
        D, N );
    }
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * r2t);
    r) R = (D * nnt - N * (ddn *
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    -refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    df;
    radiance = SampleLight( &rand, I, &L, &light,
    e.x + radiance.y + radiance.z ) > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following Small
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;

```

```
__m128 Ax4 = {...}, Ay4 = {...}, Az4 = {...};
__m128 Bx4 = {...}, By4 = {...}, Bz4 = {...};
__m128 X4 = _mm_set1_ps( 0.1f );
__m128 Cx4 = _mm_mul_ps( _mm_add_ps( Ax4, Bx4 ), X4 );
__m128 Cy4 = _mm_mul_ps( _mm_add_ps( Ay4, By4 ), X4 );
__m128 Cz4 = _mm_mul_ps( _mm_add_ps( Az4, Bz4 ), X4 );
__m128 l4 = ...;
__m128 Dx4 = ...;
__m128 Dy4 = ...;
__m128 Dz4 = ...;
```



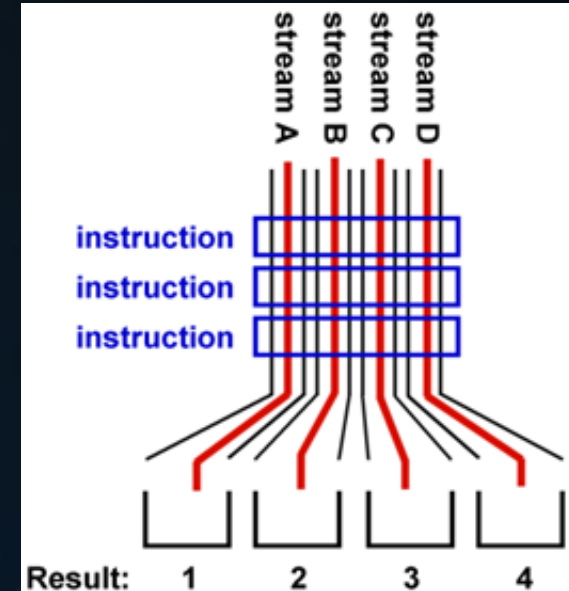
## SIMD Friendly Data Layout

Consider the following data structure:

```
struct Particle
{
    float x, y, z;
    int mass;
};
Particle particle[512];
```

# AoS

# SoA



## SIMD Data Naming Conventions

```
...ics
& (depth < MAXDEPTH)
{
    union { float x[512];    __m128 x4[128]; };
    union { float y[512];    __m128 y4[128]; };
    union { float z[512];    __m128 z4[128]; };
    union { int  mass[512];   __m128i mass4[128]; };
}

...at a = nt - nc, b = nt - nc;
...at Tr = 1 - (R0 + (1 - R0) * r);
...Tr) R = (D * nnt - N * (ddn > 0 ? 1 : -1));

...E * diffuse;
...= true;

...
...efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light );
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
{
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psum;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance.x + radiance.y + radiance.z);

random walk - done properly, closely following Small et al.
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

Notice that SoA is breaking our OO...

Consider adding the struct name to the variables:

```
float particle_x[512];
```

Or put an amount of particles in a struct.

Also note the convention of adding '4' to any SSE variable.

# Agenda:

- Introduction
- SSE / AVX
- Streams
- Vectorization

MAXDEPTH)

```
survive = SurvivalProbability( diff
estimation - doing it properly, cl
df;
```

```
radiance = SampleLight( &rand, I, 8  
e.x + radiance.y + radiance.z) > 0)
```

```

v = true;
at brdfPdf = EvaluateDiffuse( L, N );
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf);

```

```
random walk - done properly, closely  
live)
```

```
at3 brdf = SampleDiffuse( diffuse,
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
```



## Converting your Code

1. Locate a significant bottleneck in your code  
(converting is going to be labor-intensive, be sure it's worth it)
2. Keep a copy of the original code (use `#ifdef`)  
(you may want to compile on some other platform later)
3. Prepare the scalar code  
(add a `'for( int stream = 0; stream < 4; stream++ )'` loop)
4. Reorganize the data  
(make sure you don't have to convert all the time)
5. Union with floats
6. Convert one line at a time, verifying functionality as you go
7. Check MSDN for exotic SSE instructions  
(some odd instructions exist that may help your problem)

Take some time to speed up your ray tracer.

SIMD:

Easy: create four (or eight) primary rays at once, then switch to regular code.

Easy: convert four pixels to integer.

Higher gain: intersect 4 rays with 1 sphere, or 1 ray with 4 spheres.

Advanced SIMD:

Conditional code with SIMD is done with *masking*:

```
__m128 mask = _mm_cmple_ps( a4, b4 );  
__m128 result = _mm_add_ps( total4, _mm_and_ps( a4, mask ) );
```



# End of PART 4.

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 1.25 * nnt)
    {
        nt = nt / nc; ddn = ddn * nc;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * ddn);
        Tr) R = (D * nnt - N * (ddn > 0 ? 1 : -1));
    }
    E * diffuse;
    = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    {
        survive = SurvivalProbability( diffuse );
        estimation - doing it properly, closely following Small's
        if;
        radiance = SampleLight( &rand, I, &L, &lightDir );
        e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
        {
            w = true;
            at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
            at3 factor = diffuse * INVPI;
            at weight = Mis2( directPdf, brdfPdf );
            at cosThetaOut = dot( N, L );
            E * ((weight * cosThetaOut) / directPdf) * (radiance
        }
        random walk - done properly, closely following Small's
        (survive)
    }
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

