Welcome!

**Lecture 5 - “SIMD recap”**

-INFOMAGR – Advanced Graphics-

Jacco Bikker - November 2017 - February 2018

\[
\begin{align*}
I(x, x') &= g(x, x') \left[ \epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right]
\end{align*}
\]
Today’s Agenda:

- Introduction
- C++ / SSE & AVX
- Parallel Data Streams
- Practical
Introduction

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.
Advanced Graphics – SIMD Recap

Introduction

Hardware – VLIW

Vector instructions:

Vector4 a = { 1, PI, e, \sqrt{4} };  
Vector4 b = { 4, 4, 4, 4 };  
Vector4 c = a * b;

Concept:

- function A4 consists of instructions operating on 4 items
- executing A4 requires the same number of instructions required to execute A on a single item
- throughput of A4 is four times higher.

The ‘4’ in the above is known as the vector width. Modern processors support 4-wide vectors (Pentium 3 and up), 8-wide (i3/i5/i7), 16-wide (Larrabee / Xeon Phi) and 32-wide (NVidia and AMD GPUs).
SIMD Using Integers

An integer is a 32-bit value, which means that it stores 4 bytes:

```
char[] a = { 1, 2, 3, 4 };
uint a4 = (1 << 24) + (2 << 16) + (3 << 8) + 4;
```

In C++ we can directly exploit this:

```
union
{
    char a[4];
    uint a4;
};
a4 = (1 << 24) + (2 << 16) + (3 << 8) + 4;
a4 += 0x01010101;
```
SIMD Using Integers

An integer is a 32-bit value, which means that it stores 4 bytes:

```csharp
char[] a = { 1, 2, 3, 4 };
uint a4 = (1 << 24) + (2 << 16) + (3 << 8) + 4;
```

C# also allows this, although it is a bit of a hack:

```csharp
[StructLayout(LayoutKind.Explicit)]
struct byte_array
{
    [FieldOffset(0)] public byte a;
    [FieldOffset(1)] public byte b;
    [FieldOffset(2)] public byte c;
    [FieldOffset(3)] public byte d;
    [FieldOffset(0)] public unsigned int abcd;
}
```
SIMD using 32-bit values - Limitations

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

\[
\{ 100, 100, 100, 100 \} + \{ 1, 1, 1, 200 \} = \{ 101, 101, 102, 44 \}
\]
\[
\{ 100, 100, 100, 100 \} * \{ 2, 2, 2, 2 \} = \{ \ldots \}
\]
\[
\{ 100, 100, 100, 200 \} * 2 = \{ 200, 200, 201, 144 \}
\]

In general:

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

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!
Introduction

SIMD / SSE

SSE was first introduced with the Pentium-3 processor in 1999, and adds a set of 128-bit registers, as well as instructions to operate on these registers.

32-bit:

```
{ char, char, char, char } = int
```

128-bit:

```
{ float, float, float, float } = __m128
{ int, int, int, int } = __m128i
```

Apart from storing 4 floats or ints, the registers can also store two 64-bit values, eight 16-bit values or sixteen 8-bit values.
Introduction

SIMD / SSE

Problems when working with 32-bit integers:

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

Ideal situation:

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

SSE offers these benefits, except for one (guess which 😊).
Today’s Agenda:

- Introduction
- C++ / SSE & AVX
- Parallel Data Streams
- Practical
C++/SSE

Basic SSE

Any PC since the Pentium 3 will support SSE (even Atom processors). It is safe to assume a system has at least SSE4.

Basic operations:

```cpp
__m128 a4 = _mm_set_ps( 1.0f, 2.0f, 3.14159f, 1.41f );
__m128 b4 = _mm_set1_ps( 2.0f ); // broadcast
__m128 c4 = _mm_add_ps( a4, b4 );
__m128 d4 = _mm_div_ps( a4, b4 );
__m128 e4 = _mm_sqrt_ps( a4 );
```
C++/SSE

Basic SSE

Any PC since the Pentium 3 will support SSE (even Atom processors). It is safe to assume a system has at least SSE4.

Example: normalizing four vectors:

```c++
__m128 x4 = _mm_set_ps( A.x, B.x, C.x, D.x );
__m128 y4 = _mm_set_ps( A.y, B.y, C.y, D.y );
__m128 z4 = _mm_set_ps( A.z, B.z, C.z, D.z );
__m128 sqX4 = _mm_mul_ps( x4, x4 );
__m128 sqY4 = _mm_mul_ps( y4, y4 );
__m128 sqZ4 = _mm_mul_ps( z4, z4 );
__m128 sqlen4 = _mm_add_ps( _mm_add_ps( sqX4, sqY4 ), sqZ4 );
__m128 len4 = _mm_sqrt_ps( sqlen4 );
x4 = _mm_div_ps( x4, len4 );
y4 = _mm_div_ps( y4, len4 );
z4 = _mm_div_ps( z4, len4 );
```
Intermediate SSE

SSE includes powerful functions that prevent conditional code, as well as specialized arithmetic functions.

\[ \text{\_m128 min4} = \text{\_mm\_min\_ps}(a4, b4); \]
\[ \text{\_m128 max4} = \text{\_mm\_max\_ps}(a4, b4); \]
\[ \text{\_m128 one\_over\_sq4} = \text{\_mm\_rsqrt\_ps}(a4); // reciprocal square root \]
\[ \text{\_m128i int4} = \text{\_mm\_cvtps\_epi32}(a4); // cast to integer \]
\[ \text{\_m128 f4} = \text{\_mm\_cvtepi32\_ps}(\text{int4}); // cast to float \]
**Advanced SSE**

Comparisons and masking.

```c
__m128 mask4a = _mm_cmple_ps(a4, b4); // less or equal
__m128 mask4b = _mm_cmpgt_ps(a4, b4); // greater than
__m128 mask4c = _mm_cmpne_ps(a4, b4); // not equal
__m128 mask4d = _mm_cmpeq_ps(a4, b4); // equal
```

```c
__m128 combined = _mm_and_ps(mask4a, mask4b);
__m128 inverted = _mm_andnot_ps(mask4a, mask4b);
__m128 either = _mm_or_ps(mask4a, mask4b);
__m128 blended = _mm_blendv_ps(a4, b4, mask4a);
```

A good source of additional information is MSDN: [https://msdn.microsoft.com/en-us/library/bb892950(v=vs.90).aspx](https://msdn.microsoft.com/en-us/library/bb892950(v=vs.90).aspx)
AVX

Recent CPUs support 8-wide SIMD through AVX.

Simply replace __m128 with __m256, and add 256 to each function:

```c++
__m256 a8 = __mm256_set_ps1( 0 );
```
Alignment

SSE and AVX data must be properly aligned:

- **__m128** must be aligned to 16 bytes;
- **__m256** must be aligned to 32 bytes.

Visual Studio will do this for you for variables on the stack. When allocating buffers of these values, make sure you use an aligned malloc / free:

```c
__m128* data = _aligned_malloc( 1024 * sizeof( __m128 ), 16 );
```
C++/SSE

Debugging

The Visual Studio debugger considers __m128 and __m256 to be basic types.

In the debugger you can inspect them as arrays of floats, ints, shorts, bytes etc.
Today’s Agenda:

- Introduction
- C++ / SSE & AVX
- Parallel Data Streams
- Practical
Concepts

Streams

Consider the following scalar code:

```csharp
Vector3 D = Vector3.Normalize(T - P);
```

This is quite high-level. What the processor needs to do is:

```csharp
Vector3 tmp = T - P;
float length = sqrt(tmp.x * tmp.x + tmp.y * tmp.y + tmp.z * tmp.z);
D = tmp / length;
```
Streams

Consider the following scalar code:

Vector3 D = Vector3.Normalize( T - P );

This is quite high-level. What the processor needs to do is:

```cpp
float tmp_x = T.x - P.x;
float tmp_y = T.y - P.y;
float tmp_z = T.z - P.z;
float sqlen = tmp_x * tmp_x + tmp_y * tmp_y + tmp_z * tmp_z;
float length = sqrt( sqlen );
D.x = tmp_x / length;
D.y = tmp_y / length;
D.z = tmp_z / length;
```
Concepts

Streams

Consider the following scalar code:

Vector3 D = Vector3.Normalize( T - P );

Using vector instructions:

```csharp
__m128 A = T - P          // 75%

float B = dot( A, A )      // 75%

__m128 C = { B, B, B }     // 75%, overhead

__m128 D = A / C          // 75%
```
Streams

Consider the following scalar code:

```csharp
Vector3 D = Vector3.Normalize(T - P);

A = T.X - P.X
B = T.Y - P.Y
C = T.Z - P.Z
D = A * A
E = B * B
F = C * C
G = sqrt(F)
D.X = A / G
D.Y = B / G
D.Z = C / G
```

---

Advanced Graphics – SIMD Recap

Concepts

Streams

Consider the following scalar code:

```csharp
Vector3 D = Vector3.Normalize(T - P);

A = T.X - P.X
B = T.Y - P.Y
C = T.Z - P.Z
D = A * A
E = B * B
F = C * C
G = sqrt(F)
D.X = A / G
D.Y = B / G
D.Z = C / G
```
Concepts

Streams

Optimal utilization of SIMD hardware is achieved *when we run the same algorithm four times in parallel*. This way, the approach also scales naturally to 8-wide, 16-wide and 32-wide SIMD.
Concepts

Streams – Data Organization

Consider the following data structure:

```c
struct Ray {
    float ox, oy, oz;
    float dx, dy, dz, t;
};
Ray rp[256];
```

- **AoS (Array of Structures)**: Each element of the array contains all the fields of the structure.
- **SoA (Structure of Arrays)**: Each element of the array is a separate structure that contains a single field.

```c
union { float ox[256]; __m128 ox4[64]; }; union { float oy[256]; __m128 oy4[64]; }; union { float oz[256]; __m128 oz4[64]; }; union { float t[256]; __m128 t4[64]; }
```
Concepts

Streams – Ray Tracing

Leveraging SIMD for ray tracing:

1. One ray, four primitives
2. One ray, four nodes
3. Four rays, one primitive / node

Option 3 is the least intrusive:

```cpp
class Ray4
{
    public:
        __m128 ox4, oy4, oz4;
        __m128 dx4, dy4, dz4;
        __m128 t4;
};
```

```
vec3 e1 = tri.V2 - tri.V1;
vec3 e2 = tri.V3 - tri.V1;
vec3 P = cross( D, e2 );
float det = dot( e1, P );
if (det > -EPS && det < EPS) return NOHIT;
float inv_det = 1 / det;
vec3 T = 0 - tri.V1;
float u = dot( T, P ) * inv_det;
if (u < 0 || u > 1) return NOHIT;
vec3 Q = cross( T, e1);
float v = dot( D, Q ) * inv_det;
if (v < 0 || u + v > 1) return NOHIT;
float t = dot( e2, Q ) * inv_det;
if (t > EPSILON)
{
    *out = t;
    return HIT;
}
return NOHIT;
```
Concepts

Streams – Flow Divergence

Like other instructions, comparisons between vectors yield a vector of booleans.

```c
__m128 mask = _mm_cmpeq_ps( v1, v2 );
```

The mask contains a bitfield: 32 x ‘1’ for each **TRUE**, 32 x ‘0’ for each **FALSE**.

The mask can be converted to a 4-bit integer using _mm_movemask_ps:

```c
int result = _mm_movemask_ps( mask );
```

Now we can use regular conditionals:

```c
if (result == 0) { /* false for all streams */ }
if (result == 15) { /* true for all streams */ }
if (result < 15) { /* not true for all streams */ }
if (result > 0) { /* not false for all streams */ }
```
Concepts

Streams – Masking

More powerful than ‘any’, ‘all’ or ‘none’ via movemask is *masking*.

```c
if (det > -EPS && det < EPS) return NOHIT;
```

Translated to SSE:

```c
__m128 mask1 = _mm_cmple_ps( det4, MINUSEPS4 );
__m128 mask2 = _mm_cmpge_ps( det4, EPSILON4 );
__m128 det4mask = _mm_or_ps( mask1, mask2 );
if (_mm_movemask_ps( det4mask ) == 0) return NOHIT; // all rays missed
```

Note that if only one ray survives, we continue executing the algorithm.

A few lines later we have another check:

```c
if (u < 0 || u > 1) return NOHIT;
```
Concepts

Streams – Masking

Like last time, we translate

```c
if (u < 0 || u > 1) return NOHIT;
```

to

```c
mask1 = _mm_cmpge_ps( u4, ZERO4 );
mask2 = _mm_cmple_ps( u4, ONE4 );
umask = _mm_and_ps( mask1, mask2 );
```

Some rays may have ‘died’ in the previous conditional statement, so we include the mask produced by that condition:

```c
combinedmask = _mm_and_ps( det4mask, umask );
if (_mm_movemask_ps( combinedmask ) == 0) return;
```
Streams – Masking

Particularly interesting is the last conditional:

```c
if (t > EPSILON)
{
    *out = t;
    return HIT;
}
```

For four rays, we only want to change the distance we return for those rays that are still ‘alive’. For this, we use a blend operation:

```c
__m128 t4_out = _mm_blendv_ps( t4_in, t4, finalmask );
```

The beauty here is that, to the processor, this is not conditional code.
Concepts

Streams – Summary

Practical use of SSE / AVX:

- Translate your algorithm to a pure scalar flow (write out all vector operations).
- Use vectors of four or eight elements (__m128, __m256).
- Run the scalar flow four or eight times in parallel.
- Reorganize data so that each line in the algorithm can fetch 128 or 256 consecutive bits.
- Use 128-bit masks to store results of comparisons.
- Convert these to useful integers using _mm_movemask_ps.
- Continue the algorithm as long as at least one stream is alive; combine masks.
- Use _mm_blendv_ps to overwrite some values in a __m128 register.

These concepts apply to SSE, AVX and SIMD in C#.
Today's Agenda:

- Introduction
- C++ / SSE & AVX
- Parallel Data Streams
- Practical
Practical

Ray4/Tri Intersection

Scalar flow:

```c
vec3 e1 = tri.V2 - tri.V1;
vec3 e2 = tri.V3 - tri.V1;
vec3 P = cross( D, e2 );
float det = dot( e1, P );
if (det > -EPS && det < EPS) return NOHIT;
float inv_det = 1 / det;
vec3 T = O - tri.V1;
float u = dot( T, P ) * inv_det;
if (u < 0 || u > 1) return NOHIT;
vec3 Q = cross( T, e1 );
float v = dot( D, Q ) * inv_det;
if (v < 0 || u + v > 1) return NOHIT;
float t = dot( e2, Q ) * inv_det;
if (t > EPSILON) {
  *out = t;
  return HIT;
}
return NOHIT;
```

```c
Scalar flow:
float e1x = v2.x - v1.x;
float e1y = v2.y - v1.y;
float e1z = v2.z - v1.z;
float e2x = v3.x - v1.x;
float e2y = v3.y - v1.y;
float e2z = v3.z - v1.z;
float Px = r.D.y * e2z - r.D.z * e2y;
float Py = r.D.z * e2x - r.D.x * e2z;
float Pz = r.D.x * e2y - r.D.y * e2x;
float det = e1x * Px + e1y * Py + e1z * Pz;
if (det > -EPSILON && det < EPSILON) return;
float inv_det = 1 / det;
float Tx = r.O.x - v1.x;
float Ty = r.O.y - v1.y;
float Tz = r.O.z - v1.z;
float u = (Tx * Px + Ty * Py + Tz * Pz) * inv_det;
if (u < 0 || u > 1) return;
float Qx = Ty * e1z - Tz * e1y;
float Qy = Tz * e1x - Tx * e1z;
float Qz = Tx * e1y - Ty * e1x;
float v = (r.D.x * Qx + r.D.y * Qy + r.D.z * Qz) * inv_det;
if (v < 0 || u + v > 1) return;
float t = (e2x * Qx + e2y * Qy + e2z * Qz) * inv_det;
if (t > 0) r.t = min( r.t, t );
```
Ray4/Tri Intersection

Scalar flow:

```c
float e1x = v2.x - v1.x;
float e1y = v2.y - v1.y;
float e1z = v2.z - v1.z;
float e2x = v3.x - v1.x;
float e2y = v3.y - v1.y;
float e2z = v3.z - v1.z;

>>> 

__m128 e1x4 = _mm_set1_ps( v2.x - v1.x );
__m128 e1y4 = _mm_set1_ps( v2.y - v1.y );
__m128 e1z4 = _mm_set1_ps( v2.z - v1.z );
__m128 e2x4 = _mm_set1_ps( v3.x - v1.x );
__m128 e2y4 = _mm_set1_ps( v3.y - v1.y );
__m128 e2z4 = _mm_set1_ps( v3.z - v1.z );
```

vec3 e1 = tri.V2 - tri.V1;
vec3 e2 = tri.V3 - tri.V1;
vec3 P = cross( D, e2 );
float det = dot( e1, P );
if (det > -EPS && det < EPS) return NOHIT;
float inv_det = 1 / det;
vec3 T = O - tri.V1;
float u = dot( T, P ) * inv_det;
if (u < 0 || u > 1) return NOHIT;
vec3 Q = cross( T, e1 );
float v = dot( D, Q ) * inv_det;
if (v < 0 || u + v > 1) return NOHIT;
float t = dot( e2, Q ) * inv_det;
if (t > EPSILON) {
    *out = t;
    return HIT;
}
return NOHIT;

Scalar flow:

```c
float e1x = v2.x - v1.x;
float e1y = v2.y - v1.y;
float e1z = v2.z - v1.z;
float e2x = v3.x - v1.x;
float e2y = v3.y - v1.y;
float e2z = v3.z - v1.z;
```
Ray4/Tri Intersection

Scalar flow:

```c
float Px = r.D.y * e2z - r.D.z * e2y;
float Py = r.D.z * e2x - r.D.x * e2z;
float Pz = r.D.x * e2y - r.D.y * e2x;

>>> 
__m128 Px4 = _mm_sub_ps(  
    _mm_mul_ps( r4.dy4, e2z4 ),  
    _mm_mul_ps( r4.dz4, e2y4 )  
); 
__m128 Py4 = _mm_sub_ps(  
    _mm_mul_ps( r4.dz4, e2x4 ),  
    _mm_mul_ps( r4.dx4, e2z4 )  
); 
__m128 Pz4 = _mm_sub_ps(  
    _mm_mul_ps( r4.dx4, e2y4 ),  
    _mm_mul_ps( r4.dy4, e2x4 )  
);```

Practical
Ray4/Tri Intersection

Scalar flow:

```c
float det = e1x * Px + e1y * Py + e1z * Pz;
if (det > -EPSILON && det < EPSILON) return;
float inv_det = 1 / det;

>>>__m128 det4 = _mm_add_ps(
    _mm_add_ps(
    _mm_mul_ps( e1x4, Px4 ),
    _mm_mul_ps( e1y4, Py4 )
    ),
    _mm_mul_ps( e1z4, Pz4 )
);
__m128 mask1 = _mm_or_ps(
    _mm_cmples_ps( det4, MINUSEPS4 ),
    _mm_cmpges_ps( det4, EPS4 )
);
__m128 inv_det4 = _mm_rcp_ps( det4 );
```

```c
vec3 e1 = tri.V2 - tri.V1;
vec3 e2 = tri.V3 - tri.V1;
vec3 P = cross( D, e2 );
float det = dot( e1, P );
if (det > -EPSILON && det < EPSILON) return NOHIT;
float inv_det = 1 / det;
vec3 T = O - tri.V1;
float u = dot( T, P ) * inv_det;
if (u < 0 || u > 1) return NOHIT;
vec3 Q = cross( T, e1 );
float v = dot( D, Q ) * inv_det;
if (v < 0 || u + v > 1) return NOHIT;
float t = dot( e2, Q ) * inv_det;
if (t > EPSILON)
{
    *out = t;
    return HIT;
}
return NOHIT;
```
Practical

Define these at global scope

Option 1: store with the triangle
Option 2: amortize over more rays

Not as accurate as _mm_div_ps(one4, ...);
Ray/AABB Intersection

Intersection of a ray and an AABB can be efficiently calculated using the slab test*:

1. Calculate $t_{\text{min}}, t_{\text{max}}$ using the intersections of the ray with the horizontal planes;
2. Update $t_{\text{min}}, t_{\text{max}}$ using the intersections of the ray with the vertical planes.

If $t_{\text{min}} < t_{\text{max}}$ and $t_{\text{max}} > 0$, the ray intersects the AABB.

---

Boxes

Ray/AABB Intersection

AABB: AxisAligned Bounding Box.

Slab test:

Intersect the ray against pairs of planes;

\[ t_{\text{min}} = +\infty, t_{\text{max}} = -\infty \]

\[ t_{\text{min}} = \max(t_{\text{min}}, \min(t_1, t_2)) \]

\[ t_{\text{max}} = \min(t_{\text{max}}, \max(t_1, t_2)) \]

Intersection if: \( t_{\text{min}} < t_{\text{max}} \)

Since the box is axis aligned, calculating \( t \) is cheap:

\[ t = -(\vec{O} \cdot \vec{N} + d) / (\vec{D} \cdot \vec{N}) \]

\[ = -(O_x \cdot \vec{N}_x + d) / (\vec{D}_x \cdot \vec{N}_x) \]

\[ = (x_{\text{plane}} - O_x) / \vec{D}_x \]

where \( \vec{N} \) is a point on the plane.

In this case, for \( \vec{N} = (1,0,0) \):

\[ d = -(\vec{N} \cdot P), \text{ where } P \text{ is a point on the plane.} \]

\[ t = -(O_x \cdot \vec{N}_x + d) / (\vec{D}_x \cdot \vec{N}_x) \]

\[ = -(O_x - x_{\text{plane}}) / \vec{D}_x \]

\[ = (x_{\text{plane}} - O_x) / \vec{D}_x \]
Ray/AABB Intersection

Scalar code (3D):

```c
bool intersection( box b, ray r )
{
    float tx1 = (b.min.x - r.O.x) * r.rD.x;
    float tx2 = (b.max.x - r.O.x) * r.rD.x;
    float tmin = min(tx1, tx2);
    float tmax = max(tx1, tx2);
    float ty1 = (b.min.y - r.O.y) * r.rD.y;
    float ty2 = (b.max.y - r.O.y) * r.rD.y;
    tmin = max(tmin, min(ty1, ty2));
    tmax = min(tmax, max(ty1, ty2));
    float tz1 = (b.min.z - r.O.z) * r.rD.z;
    float tz2 = (b.max.z - r.O.z) * r.rD.z;
    tmin = max(tmin, min(tz1, tz2));
    tmax = min(tmax, max(tz1, tz2));
    return tmax >= tmin && tmax >= 0;
}
```
Practical

Ray/AABB Intersection

Vector code:

```c
bool intersection( box b, ray r )
{
    __m128 t1 = _mm_mul_ps( _mm_sub_ps( node->bmin4, O4 ), rD4 );
    __m128 t2 = _mm_mul_ps( _mm_sub_ps( node->bmax4, O4 ), rD4 );
    __m128 vmax4 = _mm_max_ps( t1, t2 ), vmin4 = _mm_min_ps( t1, t2 );
    float* vmax = (float*)&vmax4, *vmin = (float*)&vmin4;
    float tmax = min(vmax[0], min(vmax[1], vmax[2]));
    float tmin = max(vmin[0], max(vmin[1], vmin[2]));
    return tmax >= tmin && tmax >= 0;
}
```

```c
struct BVHNode
{
    AABB bounds;
    int leftFirst;
    int count;
};
```

```c
struct BVHNode
{
    float3 bmin;
    int leftFirst;
    float3 bmax;
    int count;
};
```

```c
struct BVHNode
{
    __m128 bmin4;
};
```

```c
struct BVHNode
{
    __m128 bmax4;
};
```
Ray/AABB Intersection

Vector code:

```c
bool intersection( box b, ray r )
{
    __m128 t1 = _mm_mul_ps( _mm_sub_ps( node->bmin4, O4 ), rD4 );
    __m128 t2 = _mm_mul_ps( _mm_sub_ps( node->bmax4, O4 ), rD4 );
    __m128 vmax4 = _mm_max_ps( t1, t2 ), vmin4 = _mm_min_ps( t1, t2 );
    float* vmax = (float*)&vmax4, *vmin = (float*)&vmin4;
    float tmax = min(vmax[0], min(vmax[1], vmax[2]));
    float tmin = max(vmin[0], max(vmin[1], vmin[2]));
    return tmax >= tmin && tmax >= 0;
}
```

Check here for an even faster version:

Today’s Agenda:

- **Introduction**
- **C++ / SSE & AVX**
- **Parallel Data Streams**
- **Practical**
INFOMAGR – Advanced Graphics
Jacco Bikker - November 2017 - February 2018

END of “SIMD recap”
next lecture: “Light Transport”