“Box pruning revisited” – an optimization project by Pierre Terdiman – 2017

Part 9b – better SIMD

Last time we wrote an initial SIMD version that ended up slower than the scalar code. We then identified the *_mm_set_ps* intrinsics as a potential source of issues.

So today we want to replace *_mm_set_ps* with "real" SIMD loads: *_mm_load_ps* (for aligned loads) or *_mm_loadu_ps* (for unaligned loads). These are the intrinsics that each map to a single assembly instruction (respectively *movaps* and *movups*). They load 4 contiguous floats into a SIMD register, so we will need to reorder the data. If we go back to the initial C++ code and rearrange it so that all elements of the same box are on the same side, we get:

```
if( a.mMin.y > b.mMax.y     // (1)
||  a.mMax.y < b.mMin.y     // (2)
||  a.mMin.z > b.mMax.z     // (3)
||  a.mMax.z < b.mMin.z)    // (4)
```

i.e. the a’s are all on the left side, and the b’s are all on the right side.

This is going to be tricky since we have a mix of "greater than" and "lesser than" comparisons in the middle. Let's ignore it for now and focus on the loads.

For a given box, we want to load the y's and z's (both min and max) with a single load. Unfortunately the AABB class so far looks like this:

```
class AABB
{
    public:
    Point   mMin;   //!< Min point
    Point   mMax;   //!< Max point
};
```

That is, switching back to plain floats:

```
struct AABB
{
    // Min point
    float mMinX;
    float mMinY;
    float mMinZ;

    // Max point
    float mMaxX;
    float mMaxY;
    float mMaxZ;
};
```

With this format we cannot load the y's and z's at the same time, since they're interleaved with the x's. So, we change the format, and introduce this new AABB class:

```
struct SIMD_AABB
{
    float mMinX;
    float mMaxX;
    //
    float mMinY;
    float mMinZ;
    float mMaxY;
    float mMaxZ;
};
```

It's the same data as before, we just put the x's first. That way the 4 floats we are interested in are stored contiguously in memory, and we can load them to an SSE register with a single assembly instruction. Which gives us this in-register layout, for two boxes:

__m128 box0 (a)	mMinY	mMinZ	mMaxY	mMaxZ
__m128 box1 (b)	mMinY	mMinZ	mMaxY	mMaxZ

Go back to the C++ code and you'll see that we need to compare minimums of (a) to maximums of (b). So we need to reorder the elements for one of the two boxes.

Since one of the box (say box0) is constant for the duration of the loop, it makes more sense to reorder the constant box (only once), instead of doing the reordering inside the loop. This is done with a single shuffle instruction (*_mm_shuffle_ps*), and we end up with:

__m128 box0 (a)	mMaxY	mMaxZ	mMinY	mMinZ
__m128 box1 (b)	mMinY	mMinZ	mMaxY	mMaxZ

Just to see where we stand, let's do a SIMD comparison (*_mm_cmpgt_ps*) between (a) and (b). This computes:

```
a.mMaxY      > b.mMinY
a.mMaxZ      > b.mMinZ
a.mMinY      > b.mMaxY
a.mMinZ      > b.mMaxZ
```

The two last lines are actually directly what we need:

```
a.mMinY      > b.mMaxY      => that's (1)
a.mMinZ      > b.mMaxZ      => that's (3)
```

The two other lines compute almost what we want, but not quite.

```
a.mMaxY      > b.mMinY      => that's the opposite of (2)
a.mMaxZ      > b.mMinZ      => that's the opposite of (4)
```

Well the opposite is not bad, it also gives us the information we need, just by flipping the result. The flipping is virtual and free: we are not going to actually do it. It just means that we test the *movemask* result against a different expected value. That is, if the previous expected value was, say, 1111b in binary, we can just flip two of the bits for which we get the opposite results, and test against, say 1100b instead. It gives us the same information, and the proper overlap test results…

…except in the equality case.

If a.mMaxY==b.mMinY for example:

- the C++ code tests “a.mMaxY < b.mMinY” and returns 0, which is the desired reference result.
- the SIMD code tests “a.mMaxY > b.mMinY” and also returns 0, which gives us a 1 after flipping.

Replacing the "greater than" comparison with "greater or equal" wouldn't work, because what we would really need is a SIMD comparison that performs the former on two of the components, and the later on the two others. So, the way it stands, the SIMD code is equivalent to this:

```
if( a.mMin.y > b.mMax.y     // => (1)
||  a.mMax.y <= b.mMin.y    // => ~(2)
||  a.mMin.z > b.mMax.z     // => (3)
||  a.mMax.z <= b.mMin.z)   // => ~(4)
```

Almost what we wanted!

And in fact, we could very well accept the differences and stop here.

After all, the equality case is often a bit fuzzy. Don't forget that we are dealing with floats here. It is bad form to use exact equality when comparing floats. When using the box pruning function for the broadphase in a physics engine, the equality case is meaningless since the bounding boxes are constantly evolving, recomputed each frame, and there is always a bit of noise in the results due to the nature of floating point arithmetic. The results will depend on how the compiler reorganized the instructions, whether x87 or SIMD is used, it will depend on the internal FPU accuracy of x87 registers, i.e. the state of the FPU control word, and so on. Basically the case where the resulting bounding boxes are exactly touching is so ill-defined that it really doesn't make any difference if the SIMD code uses ">" or ">=" on some of the components.

If we accept this limitation, we can run the test again and see the results:

Home PC:

```
Complete test (brute force): found 11811 intersections in 781912 K-cycles.
 33554 K-cycles.
 32514 K-cycles.
 32803 K-cycles.
 32498 K-cycles.
 32486 K-cycles.
 32487 K-cycles.
 32499 K-cycles.
 32488 K-cycles.
 32669 K-cycles.
 32672 K-cycles.
 32855 K-cycles.
 32537 K-cycles.
 32504 K-cycles.
 32477 K-cycles.
 32674 K-cycles.
 32505 K-cycles.
Complete test (box pruning): found 11725 intersections in 32477 K-cycles.
```

Office PC:

```
Complete test (brute force): found 11811 intersections in 808659 K-cycles.
 15775 K-cycles.
 15266 K-cycles.
 15097 K-cycles.
 15302 K-cycles.
 15259 K-cycles.
 15554 K-cycles.
 15560 K-cycles.
 16137 K-cycles.
 15709 K-cycles.
 15109 K-cycles.
 15317 K-cycles.
 15149 K-cycles.
 15238 K-cycles.
 15239 K-cycles.
 15525 K-cycles.
 15097 K-cycles.
Complete test (box pruning): found 11725 intersections in 15097 K-cycles.
```

The gains are summarized here:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(101662)			
Version2 - base	98822	0	0%	1.0
Version3	93138	~5600	~5%	~1.06
Version4	81834	~11000	~12%	~1.20
Version5	78140	~3600	~4%	~1.26
Version6a	60579	~17000	~22%	~1.63
Version6b	41605	~18000	~31%	~2.37
(Version7)	(40906)	-	-	-
(Version8)	(31383)	(~10000)	(~24%)	(~3.14)
Version9a	34486	~7100	~17%	~2.86
Version9b	32477	~2000	~5%	~3.04

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(96203)			
Version2 - base	92885	0	0%	1.0
Version3	88352	~4500	~5%	~1.05
Version4	77156	~11000	~12%	~1.20
Version5	73778	~3300	~4%	~1.25
Version6a	58451	~15000	~20%	~1.58
Version6b	45634	~12000	~21%	~2.03
(Version7)	(43987)	-	-	-
(Version8)	(29083)	(~16000)	(~36%)	(~3.19)
Version9a	31864	~13000	~30%	~2.91
Version9b	15097	~16000	~52%	~6.15

WO-AH.

Ok, pause, that's too much weirdness at the same time.

First, the reported number of intersections is wrong now (11725 vs 11811). That code is broken!

Second, the code is not much faster than the naive version 9a at home. Why?!

Third, the code is now 2X faster than the naive version on the office PC. What's going on?!

Well, that's SIMD for you. Things can become a little bonkers.

Let's talk about the number of overlapping pairs first. It should be expected that the numbers are going to be different between the brute-force and optimized versions now. The brute-force version still uses this:

```
if( a.mMin.y > b.mMax.y     // (1)
||  a.mMax.y < b.mMin.y     // (2)
||  a.mMin.z > b.mMax.z     // (3)
||  a.mMax.z < b.mMin.z)    // (4)
```

While the optimized version effectively uses this:

```
if( a.mMin.y > b.mMax.y     // => (1)
||  a.mMax.y <= b.mMin.y    // => ~(2)
||  a.mMin.z > b.mMax.z     // => (3)
||  a.mMax.z <= b.mMin.z)   // => ~(4)
```

It doesn't compute the same thing, as we previously discussed, so, obviously, the reported number of overlapping pairs can be different. It simply means that some of the boxes in the set are exactly touching.

Beyond this, there is however a much more subtle reason at play here. If you change the `AABB::Intersect()` function so that it handles the equality case like the SIMD code, you *still* don't get the same number of pairs. That's because the SIMD code has another issue we didn't mention so far: it is not symmetric anymore. That is, *Intersect(A, B) != Intersect(B, A)* in the equality case. Imagine two boxes for which a.mMaxY == b.mMinY. Since the code tests "a.mMaxY <= b.mMinY", we get a positive result here. Now swap a and b. We now have b.mMaxY == a.mMinY, but the code tests "a.mMinY > b.mMaxY", which returns a negative result. What this means is that the results will depend on the order in which the boxes are tested by the functions.

For the brute-force function we always use the same order, and who is "a" or "b" never changes. But the optimized code sorts the boxes according to X, so who is "a" or "b" changes depending on the box positions. So we cannot easily compare the two versions anymore, which makes testing and validating the code a bit problematic. One way to do it is to explicitly avoid the equality case in the input set, in which case the returned number of intersections becomes the same again for the two functions. Another way to do it is to pre-sort the boxes along X, to ensure that the brute-force code will process the boxes in the same order as the SIMD code. If you do all that (change the `AABB::Intersect()` function, pre-sort the boxes), the validity tests always pass and the number of pairs is always the same for the two functions. The code is not broken!

However, you may find these difficulties a bit too much to swallow. Fair enough. Since the SIMD function is harder to test and validate that before, we labeled it "unsafe". And before discussing the performance results, we're going to see how to make it "safe", i.e. how to make it compute exactly the same overlaps as the scalar code, in all cases.

Let's go back to the initial scalar code:

```
if( a.mMin.y > b.mMax.y
||  a.mMax.y < b.mMin.y
||  a.mMin.z > b.mMax.z
||  a.mMax.z < b.mMin.z)
```

Right, so the problem was that with a's and b's on the same side, we get a mix of "<" and ">" operators. To fix this, we could just multiply some of the components by -1. Which would give:

```
if(  a.mMin.y >  b.mMax.y
||  -a.mMax.y > -b.mMin.y
||   a.mMin.z >  b.mMax.z
||  -a.mMax.z > -b.mMin.z)
```

And… that just works. It adds some overhead though, in particular one extra load and one extra multiply within the inner loop (since we need to fix both boxes).

But we can improve this. Since version 6b we are parsing internal bounds arrays in the main loop, rather than the user's input array. So we are free to store the bounds in any form we want, and in particular we can pre-multiply mMin.y and mMin.z by -1.0. That way we don't need to do the multiply on box (b) within the inner loop.

On the other hand we still need to flip the signs of mMax.y and mMax.z for box (a), and since we pre-multiplied the mins (which has an effect on box (a) as well, it comes from the same buffer), the net result is that we now need to multiply the whole box (a) by -1.0 at runtime. This makes the code simpler, and since box (a) is constant for the duration of the inner loop, the impact on performance is minimal. In the end, the fix to go from the unsafe version to the safe version is just two additional instructions (*_mm_load1_ps* + *_mm_mul_ps*) on the constant box. That's all!

And the impact on performance is invisible:

Home PC:

```
Complete test (brute force): found 11811 intersections in 781499 K-cycles.
 34077 K-cycles.
 32779 K-cycles.
 32584 K-cycles.
 33237 K-cycles.
 32656 K-cycles.
 32618 K-cycles.
 33197 K-cycles.
 32662 K-cycles.
 32757 K-cycles.
 32582 K-cycles.
 32574 K-cycles.
 32579 K-cycles.
 32592 K-cycles.
 32565 K-cycles.
 32867 K-cycles.
 32591 K-cycles.
Complete test (box pruning): found 11811 intersections in 32565 K-cycles.
```

Office PC:

```
Complete test (brute force): found 11811 intersections in 822472 K-cycles.
 16558 K-cycles.
 16051 K-cycles.
 15127 K-cycles.
 15324 K-cycles.
 15667 K-cycles.
 15169 K-cycles.
 15117 K-cycles.
 15324 K-cycles.
 15117 K-cycles.
 15902 K-cycles.
 16729 K-cycles.
 15860 K-cycles.
 15937 K-cycles.
 15218 K-cycles.
 15356 K-cycles.
 15116 K-cycles.
Complete test (box pruning): found 11811 intersections in 15116 K-cycles.
```

The gains are summarized here:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(101662)			
Version2 - base	98822	0	0%	1.0
Version3	93138	~5600	~5%	~1.06
Version4	81834	~11000	~12%	~1.20
Version5	78140	~3600	~4%	~1.26
Version6a	60579	~17000	~22%	~1.63
Version6b	41605	~18000	~31%	~2.37
(Version7)	(40906)	-	-	-
(Version8)	(31383)	(~10000)	(~24%)	(~3.14)
Version9a	34486	~7100	~17%	~2.86
Version9b - unsafe	32477	~2000	~5%	~3.04
Version9b - safe	32565	~1900	~5%	~3.03

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(96203)			
Version2 - base	92885	0	0%	1.0
Version3	88352	~4500	~5%	~1.05
Version4	77156	~11000	~12%	~1.20
Version5	73778	~3300	~4%	~1.25
Version6a	58451	~15000	~20%	~1.58
Version6b	45634	~12000	~21%	~2.03
(Version7)	(43987)	-	-	-
(Version8)	(29083)	(~16000)	(~36%)	(~3.19)
Version9a	31864	~13000	~30%	~2.91
Version9b - unsafe	15097	~16000	~52%	~6.15
Version9b - safe	15116	~16000	~52%	~6.14

This is basically the same speed as the unsafe version. Yay!

We are still going to keep the two unsafe and safe versions around for a while, in case further optimizations work better with one of them.

Note how all of this came together thanks to previous optimizations:

- We were able to use a SIMD comparison because we dropped the user-defined axes in version 5.
- We were able to create a SIMD-friendly AABB class because we now parse internal buffers instead of the user's buffers, after version 6b.
- For the same reason we were able to go from the unsafe version to the safe version "for free" because we could store our boxes pre-flipped. This was made possible by version 6b as well.

That's quite enough for one post so we will investigate the remaining mystery next time: why are the timings so different between the home PC and the office PC?

What we learnt:

The same SIMD code can have very different performance on different machines.

Therefore it is important to test the results on different machines. If you would only test this on the home PC, after writing two SIMD versions (9a / 9b) that end up slower than a scalar version (8), you might be tempted to give up. You need to see the results on the office PC to remain motivated.

There's a ripple effect with optimizations: some of them may have value not because of their limited performance gains, but simply because they open the door for more optimizations further down the road.