"Box pruning revisited" – an optimization project by Pierre Terdiman – 2017

Part 5 – hardcoding

Last time we saw how making the code less generic could also make it faster. This trend continues in this post.

There is another part of the box-pruning function that looks suspiciously over-generic. In the original code the sorting axes are defined by the user and sent as a parameter to the function:

```
bool CompleteBoxPruning(..., const Axes& axes);
```

In the initial code from 2002, I passed the following axes to the function:

```
Axes axes;
axes.Axis0 = 0;
axes.Axis1 = 2;
axes.Axis2 = 1;
```

Inside the function they are copied to local variables:

```
// Catch axes
udword Axis0 = axes.Axis0;
udword Axis1 = axes.Axis1;
udword Axis2 = axes.Axis2;
```

After that, the boxes are sorted along the main axis (*Axis0*), which essentially replicates the X-related part of the `AABB::Intersect(const AABB& a)` function. In other words, the sorting and scanning procedure emulates this line from `AABB::Intersect()`:

```
if(mMax.x < a.mMin.x || a.mMax.x < mMin.x
```

And the remaining `AABB::Intersect()` tests for Y and Z are implemented using the remaining *Axis1* and *Axis2* variables:

```
if(list[Index0]->Intersect(*list[Index1], Axis1))
{
    if(list[Index0]->Intersect(*list[Index1], Axis2))
    {
```

This is all good and fine. Unfortunately it is also sub-optimal.

It was invisible to me when I first wrote the code, but I know now that fetching the axes from user-defined variables has a measurable impact on performance. I saw it happen multiple times in multiple algorithms. The cost is not always large, but it is always real: fetching the axes is going to be either an extra read from the stack, or one less register available to the optimizer to do its job - and there aren't a

lot of registers in the first place on x86. Either way it makes the addressing mode more complex, and if all these things happen in the middle of an inner loop, it quickly accumulates and becomes measurable.

So a much better idea is simply to hardcode the axes.

Well, a more efficient idea at least. Whether it is "better" or not is debatable.

Let's see the effect on performance first, and debate later.

So the function sorted the boxes along X, and then did extra overlap tests on Y and Z within the most inner loop. The rationale for this default choice was that the up axis is usually either Y or Z, "never" X. You typically don't want to sort along your up axis, because most games have a flat, essentially 2D world, even if they're 3D games. If you project the bounds of such a world along the up axis, the projections will pretty much all overlap each-other, and this will greatly reduce the "pruning power" of the main sort. Basically the algorithm would then degenerate to the O(n^2) brute-force version.

Thus, the safest main sorting axis is X, because it is usually not the vertical axis. After that, and for similar reasons, it is usually slightly better to use the remaining non-vertical axis for "Axis1" and keep the vertical axis last in "Axis2" (the 2D overlap test can then early-exit more quickly).

One can also analyze the distribution of objects at runtime and select a different main axis each frame, depending on how objects are spatially organized at any given time. That's what I did in my old Z-Collide library, eons ago.

Similarly, one could try to make the code *more* generic by removing the requirements that the input axes are the regular X, Y, Z coordinate axes. After all it is perfectly possible to project the boxes along an arbitrary 3D axis, so the main sorting axis could be anything, and the remaining other two axes could be derived from it in the same way a frame can be derived from just a view vector. Unfortunately making the code more generic in such a way, while "better" in theory (because the main sorting axis could be "optimal"), is a *terrible idea* in practice. You need to introduce dot-products to project the boxes instead of directly reading the bounds' coordinates, it adds a lot of overhead, and the resulting code is ultimately slower than what you started with - even with a "better" main sorting axis.

No, as we discussed in part 4, what you need to do for performance is the opposite: make the code less generic.

And hardcode the axes.

Do this, and then that part goes away:

```
const float MaxLimit = list[Index0]->GetMax(Axis0);
```

It's replaced with simply:

```
const float MaxLimit = list[Index0]->mMax.x;        // MODIFIED
```

Similarly, and especially, this also goes away:

```
if(list[Index0]->Intersect(*list[Index1], Axis1))
{
    if(list[Index0]->Intersect(*list[Index1], Axis2))
    {
```

It's replaced with:

```
if(intersects2D(*list[Index0], *list[Index1]))
{
```

Which is implemented this way (note the hardcoding of axes):

```
static __forceinline int intersects2D(const AABB& a, const AABB& b)
{
        if(     b.mMax.y < a.mMin.y || a.mMax.y < b.mMin.y
        ||      b.mMax.z < a.mMin.z || a.mMax.z < b.mMin.z)
                return 0;
        return 1;
}
```

So we completely eliminate the need for passing axes to the function, or reading them from local variables. By reading ".x", ".y" and ".z" directly, we give the compiler the freedom to simply hardcode the offsets in the address calculation. Let's look at the resulting disassembly.

It looked like this before the change:

```
        const float MaxLimit = list[Index0]->GetMax(Axis0);
010A2D8A  mov         eax,dword ptr [esp+3Ch]
        const float MaxLimit = list[Index0]->GetMax(Axis0);
010A2D8E  mov         ecx,dword ptr [esp+14h]
010A2D92  mov         eax,dword ptr [eax+edi*4]
010A2D95  movss       xmm1,dword ptr [eax+ecx*4+0Ch]
010A2D9B  movss       dword ptr [esp+24h],xmm1
```

It looks like this after the change:

```
        const float MaxLimit = list[Index0]->mMax.x;        // MODIFIED
013A2C8A  mov         eax,dword ptr [esp+30h]
013A2C8E  mov         eax,dword ptr [eax+edi*4]
013A2C91  movss       xmm1,dword ptr [eax+0Ch]
013A2C96  movss       dword ptr [esp+18h],xmm1
```

Did you spot the difference?

“0Ch” is the offset to go from min to max within the AABB class, so you can easily guess that *eax* in the first *movss* is the AABB address.

“ecx*4” disappeared entirely. That was the part dealing with the user-defined axes.

So we got the expected results:

- the addressing mode became simpler (“eax+0Ch” instead of “eax+ecx*4+0Ch”)
- we got one less read from the stack / one less instruction (“mov ecx, dword ptr [esp+14h]” vanished)
- register pressure decreased (new code doesn’t use *ecx*, while initial code did)

The performance increase is minor in this case, but measurable:

Home PC:

Complete test (brute force): found 11811 intersections in 781460 K-cycles.
78971 K-cycles.
78157 K-cycles.
78216 K-cycles.
78319 K-cycles.
78140 K-cycles.
80163 K-cycles.
78618 K-cycles.
78194 K-cycles.
78182 K-cycles.
78308 K-cycles.
78140 K-cycles.
78252 K-cycles.
78385 K-cycles.
78489 K-cycles.
78404 K-cycles.
78620 K-cycles.
Complete test (box pruning): found 11811 intersections in 78140 K-cycles.

Office PC:

Complete test (brute force): found 11811 intersections in 824010 K-cycles.
75750 K-cycles.
76740 K-cycles.
75640 K-cycles.
75539 K-cycles.
75800 K-cycles.
78633 K-cycles.
80828 K-cycles.
82691 K-cycles.
74491 K-cycles.
74675 K-cycles.
75398 K-cycles.
73778 K-cycles.
75074 K-cycles.
79776 K-cycles.
73916 K-cycles.
74263 K-cycles.
Complete test (box pruning): found 11811 intersections in 73778 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

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

This doesn't look like much but it will have profound effects on further optimizations. This change in itself does not make a big difference, but it opens the door for more drastic optimizations like SIMD. We will come back to this in another blog post.

What we learnt:

Hardcoding things can make the code run faster.

From small beginnings come great things. Small optimizations can be worth doing just because they open the door for further, greater optimizations.

For now, before we wrap up this post, there was a bit of a debate left to deal with.

Is it worth the risk of hitting a case where sorting along X makes the code degenerate to O(n^2)?

Yes! Because all broadphase algorithms have bad cases that degenerate to O(n^2) anyway. Just put all the bounds at the origin (each box overlapping all other boxes) and the fastest broadphase algorithm in the world in this case is the brute-force O(n^2) approach. Everything else will have the overhead of the fancy pruning algorithms, that will not be able to prune anything, plus the exact same number of AABB overlap tests on top of it. So the existence of potential degenerate cases is not something to be overly worried about: they exist for all broadphase algorithms anyway.

Is it worth losing the genericity for such minor gains?

Yes! Because it turns out we didn't entirely lose the genericity. There are two traditional ways to get the performance gains from hardcoding the inputs while keeping the genericity of the non-hardcoded version.

The first way is to use templates. You could rewrite this function with templates, and instantiate a different function for different values of the input axes. This works, but I don't really like the approach. Moving all the code to headers gives me less control over inlining. And generating multiple versions of the same function can quickly bloat the code and silently increase the size of the exe for no valid reason, which in turn can make things slower simply by decreasing locality between callers and called. These things happen, and these things matter - that's why you have optimization options in the linker, to let users define the function order within the final executable.

The second way then, is to realize that X, Y and Z are just offsets within the user-defined array. And nothing forces you to put values there in this traditional order.

That is, if you want to sort along Z instead of X, just switch the X and Z values in your input bounds. You don't even need to duplicate the function: just change the input data. Granted: it is not always possible to modify the input data when the bounds are, say, stored within a game object. Flipping the bounds coordinates there would make the box pruning function work with the preferred sorting axes, but it would also have an effect on every other piece of code accessing the same bounds. Which is not ideal.

So what's the ideal?

Well, we just gave a hint of things to come when we mentioned tweaking the input data. Optimizing the code is one thing, but optimizing the data is equally important, if not more. Next time we will implement a data-oriented optimization that will not only make the function a lot faster, but also give us back our genericity for no extra cost.