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

Part 8 – branchless overlap test

For this post, we start again from version 6b, ignoring our experiments with integer comparisons in version 7. We are going to stay focused on comparisons though, and looking at this function:

```
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;
}
```

It contains four comparisons, and four branches. The box values are going to be arbitrary, so these branches are the worst you can get: branch prediction will essentially never work for them. If we check the disassembly, it looks like this:

```
                if(intersects2D(BoxList[Index0], BoxList[Index1]))
012F2EAD  movss       xmm0,dword ptr [edi+4]
012F2EB2  comiss      xmm0,dword ptr [eax+10h]
012F2EB6  ja          CompleteBoxPruning+293h (012F2F33h)
012F2EB8  movss       xmm0,dword ptr [eax+4]
012F2EBD  comiss      xmm0,dword ptr [edi+10h]
012F2EC1  ja          CompleteBoxPruning+293h (012F2F33h)
012F2EC3  movss       xmm0,dword ptr [edi+8]
012F2EC8  comiss      xmm0,dword ptr [eax+14h]
012F2ECC  ja          CompleteBoxPruning+293h (012F2F33h)
012F2ECE  movss       xmm0,dword ptr [eax+8]
012F2ED3  comiss      xmm0,dword ptr [edi+14h]
012F2ED7  ja          CompleteBoxPruning+293h (012F2F33h)
                {
```

Pretty much what you'd expect from the C++ code, and all the comparisons and branches are there indeed.

Comparisons and branches. We often talk about these two as if they would always go hand in hand and come together, but they are very different things and it's important to distinguish between the two. In terms of performance, one of them is more expensive than the other.

Let me demonstrate. We can rewrite the overlap function this way:

```
static __forceinline int intersects2D(const AABB& a, const AABB& b)
{
    const bool b0 = b.mMax.y < a.mMin.y;
    const bool b1 = a.mMax.y < b.mMin.y;
    const bool b2 = b.mMax.z < a.mMin.z;
    const bool b3 = a.mMax.z < b.mMin.z;
    const bool b4 = b0||b1||b2||b3;
    return !b4;
}
```

Compare to the previous function, and you easily see that the comparisons are still there in the C++ code. We still compare max floats to min floats, four times, none of that has changed. We just wrote the code slightly differently.

Compile it, run it, and... You get exactly the same timings as before. In fact, you get exactly the same disassembly. Thus, the branches are also all there. Well really, that makes sense.

Now, however, for something that perhaps does not make as much sense, remove three characters from this function and write it this way instead:

```
static __forceinline int intersects2D(const AABB& a, const AABB& b)
{
    const bool b0 = b.mMax.y < a.mMin.y;
    const bool b1 = a.mMax.y < b.mMin.y;
    const bool b2 = b.mMax.z < a.mMin.z;
    const bool b3 = a.mMax.z < b.mMin.z;
    const bool b4 = b0|b1|b2|b3;
    return !b4;
}
```

Pretty much the same, right?

Well, not quite. The compiler begs to differ, and produces the following disassembly:

```
                if(intersects2D(BoxList[Index0], BoxList[Index1]))
002D2ECD  movss       xmm0,dword ptr [list]
                if(intersects2D(BoxList[Index0], BoxList[Index1]))
002D2ED2  comiss      xmm0,dword ptr [edx+14h]
002D2ED6  movss       xmm0,dword ptr [nb]
002D2EDB  seta        cl
002D2EDE  comiss      xmm0,dword ptr [edx+10h]
002D2EE2  movss       xmm0,dword ptr [edx+8]
002D2EE7  seta        al
002D2EEA  or          cl,al
002D2EEC  comiss      xmm0,dword ptr [ebp+14h]
002D2EF0  movss       xmm0,dword ptr [edx+4]
002D2EF5  seta        al
002D2EF8  or          cl,al
002D2EFA  comiss      xmm0,dword ptr [ebp+10h]
002D2EFE  seta        al
002D2F01  or          cl,al
002D2F03  jne         CompleteBoxPruning+28Bh (02D2F5Bh)
```

The *comparisons* are still here (the "comiss") but the *branches* are mostly gone: there's only one branch instead of four. That's all thanks to the "seta" instructions, which can store the results of our comparisons directly in a register, in a branchless way. Technically the final branch is not even in the overlap function itself, it's just part of the calling code that uses the returned value. The overlap function itself became branchless.

Using | instead of || told the compiler it was necessary to perform all comparisons to create the final bool, while the previous version had early-exits (and sometimes only performed one comparison instead of four). The new code is also larger, since it needs to concatenate the bools to create the final return value. But using more instructions does not always mean slower results:

Home PC

Complete test (brute force): found 11811 intersections in 778666 K-cycles.
32497 K-cycles.
31411 K-cycles.
31402 K-cycles.
31637 K-cycles.
31415 K-cycles.
31440 K-cycles.
31404 K-cycles.
31412 K-cycles.
31388 K-cycles.
31596 K-cycles.
31411 K-cycles.
31383 K-cycles.
31595 K-cycles.
31405 K-cycles.
31383 K-cycles.
31392 K-cycles.

Complete test (box pruning): found 11811 intersections in 31383 K-cycles.

Office PC:

Complete test (brute force): found 11811 intersections in 819311 K-cycles.
29897 K-cycles.
29563 K-cycles.
29620 K-cycles.
29430 K-cycles.
29641 K-cycles.
29352 K-cycles.
29363 K-cycles.
29305 K-cycles.
30214 K-cycles.
29538 K-cycles.
31417 K-cycles.
30416 K-cycles.
30112 K-cycles.
30443 K-cycles.
30105 K-cycles.
29083 K-cycles.
Complete test (box pruning): found 11811 intersections in 29083 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

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

Well, we are now 3 times faster than when we started. That's another milestone I suppose.

It is important to note that going branchless does *not* always pay off. Branches are expensive if mispredicted, but cheap if correctly predicted. *Also, a branch that early-exits and skips an expensive part of the code can still be cheaper overall than running the expensive part of the code*. It's obvious, but multiple times I've seen people rewrite functions in a branchless way, and just assume that the result was better - *without benchmarking*. Unfortunately the results were often slower. There is absolutely no guarantee here. If you blindly follow a rule or a recipe, removing branches just for the sake of removing branches, you won't go very far. This is a case where you need to test, profile, use your brain.

What we learnt:

Comparisons and branches are two different things.

Mispredicted branches are costly, rewriting in a branchless way can help here.

Don't assume the results will be faster. Profile the difference.

The C++ versions can look very similar (only 3 extra characters) but have a very different disassembly, and very different performance. You need to look through the C++, what matters is the generated code.

Are we done?

No!

Next time we will *finally* attack the SIMD version.