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

Part 6a – data-oriented design

There has been a lot of talks and discussions in the past about "data-oriented programming" and how it leads to faster code. Just Google the term along with "Mike Acton" and you will find plenty of it. I don't need to repeat everything here: Mike is right. Listen to Mike.

In practical terms: we need to think about cache misses now. What we do with the data is one thing, but how we access the data is another equally important thing.

As far as our box pruning function is concerned, the data is pretty simple: we read an array of AABB pointers in input, and we write an array of pairs in output.

```
bool CompleteBoxPruning(udword nb, const AABB** list, Container& pairs);
```

Reading from and writing to linear arrays is the most cache-friendly thing one can do, so we're not starting from a bad place (there is no linked lists or complicated structures here). However we are reading an array of AABB *pointers*, not an array of AABBs. And that extra indirection is probably costly.

I remember using this design for practical reasons: I wanted the function to be user-friendly. At the time I was using a classical "object-oriented" design for my game objects, and I had something like this:

```
class GameObject
{
    public:
        ...
        AABB    mBounds;
        ...
};
```

The AABB of each object was embedded within the game object class, vanilla C++ design. And I didn't want the library to force users to first gather and copy the bounds in a separate bounds array just to be able to call the function. So instead, the function accepts an array of AABB pointers, so that it can read the AABBs directly from the game objects. The AABBs can neatly stay as a member of the objects, there is no need to duplicate them in a separate array and use more memory, no need to break up the neat objects into disjoint classes, it all looked quite nice.

And I suppose it was. Nice.

But fast, no, that, it was not.

Reading the bounds directly from the game objects is an extraordinarily bad idea: it makes you read some scattered data from some random location for each box, it's the perfect recipe for cache misses and basically the worst you can do.

You can solve it with *prefetch* calls, to some extents. But prefetching is a fragile solution that tends to break when code gets refactored, and making it work in a cross-platform way is tedious (the different platforms may not even have the same cache line size, so you can easily imagine the complications).

A better idea is to switch to "data-oriented design" instead of "object-oriented design". In our case it simply means that the bounds are extracted from the game objects and gathered in an array of bounds, typically called "bounds manager". Something along these lines:

```
class GameObject
{
    public:
        ...
        int     mBoundsIndex;   // Index within BoundsManager::mBounds
        ...
};

class BoundsManager
{
    public:
        AABB*   mBounds;    // Linear array of bounds for all GameObjects
};
```

Vanilla data-oriented design, that one.

There are pros & cons to both approaches, which are beyond the scope of this blog post. All I am concerned about here is the performance of the box pruning function, and in this respect the data-oriented design wins big time.

The modifications to the code are minimal. The function signature simply becomes:

```
bool CompleteBoxPruning(udword nb, const AABB* list, Container& pairs);
```

And the corresponding indirections are removed from the code. That's it. That's the only difference. We don't "optimize" the code otherwise. But the effect on performance is still dramatic:

Home PC:

Complete test (brute force): found 11811 intersections in 782034 K-cycles.
64260 K-cycles.
60884 K-cycles.
60645 K-cycles.
60586 K-cycles.
60814 K-cycles.
62650 K-cycles.
60591 K-cycles.
60697 K-cycles.
60833 K-cycles.
60579 K-cycles.
60601 K-cycles.
60905 K-cycles.
60596 K-cycles.
60836 K-cycles.
60673 K-cycles.
60585 K-cycles.
Complete test (box pruning): found 11811 intersections in 60579 K-cycles.

Office PC:

Complete test (brute force): found 11811 intersections in 810333 K-cycles.
59420 K-cycles.
58451 K-cycles.
59005 K-cycles.
65448 K-cycles.
64930 K-cycles.
62007 K-cycles.
59005 K-cycles.
58847 K-cycles.
58904 K-cycles.
58462 K-cycles.
58843 K-cycles.
58655 K-cycles.
58583 K-cycles.
59056 K-cycles.
58776 K-cycles.
58556 K-cycles.
Complete test (box pruning): found 11811 intersections in 58451 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

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

The modifications to the code have been the smallest so far (really just replacing "->" with "." here and there), and yet this gave the biggest speedup until now. In other words: our current best optimization did not come from modifying how we process the data (==roughly, "the code"), or from changing the data itself, but instead it came from changing how we access the data. That's a pretty powerful result.

What we learnt:

Choose data-oriented design over object-oriented design for performance.

The data access pattern is often more important than anything else.

We are not done yet though. There is a lot more to do w.r.t. optimizing our data access, as we will see next time. But this change was so small and yet so effective that it was worth isolating in a separate blog post to drive the point home.