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

Part 1 - The setup

Back in 2002 I released a small "box pruning" library on my website. Basically, this is a "broadphase" algorithm: given a set of input bounds (AABBs - for *Axis Aligned Bounding Boxes*), it finds the subset of overlapping bounds (reported as an array of overlapping pairs). There is also a "bipartite" version that finds overlapping boxes between two sets of input bounds. For more details about the basics of box pruning, please refer to this document.

Note that "box pruning" is not an official term. That's just how I called it some 15 years ago.

Since then, the library has been used in a few projects, commercial or otherwise. It got benchmarked against alternative approaches: for example you could still find it back in *Bullet 2.82*'s "Extras\CDTestFramework" folder, tested against Bullet's internal "dbVt" implementation. People asked questions. People sent suggestions. And up until recently, people sent me results showing how their own alternative broadphase implementation was faster than mine. The last time that happened, it was one of my coworkers, somebody sitting about a meter away from me in the office, who presented his hash-grid based broadphase which was "2 to 3 times faster" than my old box pruning code.

That's when I knew I had to update this library. Because, of course, I've made that code quite a bit faster since then.

Even back in 2002, I didn't praise that initial implementation for its performance. The comments in there mention the "sheer simplicity", explicitly saying that it is "probably not faster" than the other competing algorithms from that time. This initial code was also not using optimizations that I wrote about later in the SAP document, e.g. the use of "sentinels" to simplify the inner loop. Thus, clearly, it was not optimal.

And then of course, again: *it's been 15 years*. I learnt a few things since then. The code I write today is often significantly better and faster than the code I wrote 15 years ago. Even if my old self would have had a hard time accepting that.

So, I thought I could write a series of blog posts about how to optimize that old implementation from 2002, and make it a good deal faster. Like, I don't know, let's say an order of magnitude faster as a target.

Sounds good?
Ok, let's start.

For this initial blog post I didn't do any real changes, I am just setting up the project. The old version was compiled with VC6, which I unfortunately don't have anymore. For this experiment I randomly picked VC11.

Here is a detailed list of what I did:

- Converted the project from Visual Studio 6 to Visual Studio 2012. I used the automatic conversion and didn't tweak any project settings. I used whatever was enabled after the conversion. We will investigate the compiler options in the next post.

- Made it compile again. There were some missing loop indices in the radix code, because earlier versions of Visual Studio were notoriously not properly handling the scope in for loops (see */Zc::forScope*). Other than that it compiled just fine.

- Moved the files that will not change in these experiments to a shared folder outside of the project. That way I can have multiple Visual Studio projects capturing various stages of the experiment without copying all these files in each of them.

- Added a new benchmark. In the old version there was a single test using the "CompleteBoxPruning" function, with a configuration that used 5000 boxes and returned 200 overlapping pairs. This is a bit too small to properly stress test the code, so I added a new test that uses 10000 boxes and returns 11811 intersections. That's enough work to make our optimizations measurable. Otherwise they can be invisible and lost in the noise.

- In this new test ("RunPerformanceTest" function) I loop multiple times over the "BruteForceCompleteBoxTest" and "CompleteBoxPruning" tests to get more stable results (recording the 'min'). Reported timing values are divided by 1024 to get results in "K-Cycles".

- I also added a validity test ("RunValidityTest" function) to make sure that the two functions, brute-force and optimized, keep reporting the same pairs.

And that's about it. There are no modifications to the code otherwise for now; it is the same code as in 2002.

The initial results are as follows, on my home PC and my office PC. The *CPU-Z* profiles for both machines are available here:

Home PC: pierre-pc

Office PC: pterdiman-dt

Note that they aren't really "killer" machines. They are kind of old and not super fast. That's by design. Optimizing things on the best & latest PC often hides issues that the rest of the world may suffer from. So you ship your code thinking it's fast, and immediately get reports about performance problems from everybody. Not good. I do the opposite, and (try to) make the code run fast on old machines. That way

there's no bad surprises. For similar reasons I test the results on at least two machines, because sometimes one optimization only works on one, and not on the other. Ideally I could / should have used entirely different platforms here (maybe running the experiments on consoles), but I'm not sure how legal it is to publish benchmark results from a console, so, maybe next time.

Home PC:

```
Complete test (brute force): found 11811 intersections in 795407 K-cycles.
102583 K-cycles.
102048 K-cycles.
101721 K-cycles.
101906 K-cycles.
101881 K-cycles.
101662 K-cycles.
101768 K-cycles.
101693 K-cycles.
102094 K-cycles.
101924 K-cycles.
101696 K-cycles.
101964 K-cycles.
102000 K-cycles.
101789 K-cycles.
101982 K-cycles.
101917 K-cycles.
Complete test (box pruning): found 11811 intersections in 101662 K-cycles.
```

Office PC:

```
Complete test (brute force): found 11811 intersections in 814615 K-cycles.
106695 K-cycles.
96859 K-cycles.
97934 K-cycles.
99237 K-cycles.
97394 K-cycles.
97002 K-cycles.
96746 K-cycles.
96856 K-cycles.
98473 K-cycles.
97249 K-cycles.
96655 K-cycles.
102757 K-cycles.
96203 K-cycles.
96661 K-cycles.
107484 K-cycles.
104195 K-cycles.
Complete test (box pruning): found 11811 intersections in 96203 K-cycles.
```

The "brute force" version uses the unoptimized O(N^2) "BruteForceCompleteBoxTest" function, and it is mainly there to check the returned number of pairs is valid. I only report one performance number for this case - we don't care about it.

The "box pruning" version uses the "CompleteBoxPruning" function, that we are going to optimize. As we can see here, this initial implementation offered quite decent speedups already - not bad for something that was about 50 lines of vanilla C++ code.

For the "complete box pruning case" I make the code run 16 times and record the minimum time. I am going to report the 16 numbers from the 16 runs though, to show how stable the machines are (i.e. do we get reproducible results or is it just random?), and to show the cost of the first run (which is usually more expensive than the subsequent runs, so it's a worst-case figure).

These results are out-of-the-box after the automatic project conversion. Needless to say, this is compiled in *Release* mode.

If you want to replicate these numbers and get stable benchmark results from one run to the next, make sure your PC is properly setup for minimal interference. See for example the first paragraphs in this post .

For now I will only focus on the "CompleteBoxPruning" function, completely ignoring its "BipartiteBoxPruning" counterpart in my reports. All optimizations will be equally valid for the bipartite case, but there will be enough to say and report with just one function for now. The companion code might update and optimize the bipartite function along the way though - I'll just not talk about it.

You can find the initial disassembly in CompleteBoxPruning_Disassembly1.txt (202 instructions). We are going to keep a close eye to the disassembly in this project.

That's it for now.

Next time, we will play with the compiler options and see what kind of speedup we can get "for free".