Spike: Task 24.P
Title: Measuring Performance & Optimisations

Author: Mitchell Wright, 100595153

Goals / deliverables:

To demonstrate an understanding of code performance and collision systems, while being able to optimise pre-existing code.

Items created during task:

- Code, see: \24 - Spike – Measuring Performance and Optimisations \SDL2

Technologies, Tools, and Resources used:

- Visual Studio 2022
- SourceTree
- GitHub
- Lecture 3.2 – Data Structures

Tasks undertaken:

- Finding the Most Efficient Collision Detection
- Adding Optimizations
- Commit to Git

What we found out:

1. Finding the Most Efficient Collision Detection:

Method A1/A2: These tests take an int as a paremeter, declare 8 ints as box corner positions, 2 boxes copied from the box collection, then assign the corner positions based on box position, width and height, then proceeds to do the actual check. The end result is that I hate looking at it. It should be noted that A1 checks every single box for a collision with every other box. A2, however, skips a large number of repeat checks by only starting each secondary loop from +1 of the main loop index.

Method B: Method B takes a copy of the boxes as parameters, but thereafter performs the same as Method A2. Getting better but still makes me sad.

Method C: Similar to Method B, though now with a reference to the boxes as parameters instead. No unnecessary copying happening here.

Method D: Cuts out all of the nonsense, and just accesses the box's stats rather than assigning them to new variables. This one is nice. I can look at this and not feel sad.

For performance testing, I used the provided program with the render portion of the loop set to false, so that render overhead was not a factor in performance. The results of the testing are shown below.

Number of Boxes	Test A1	Test A2	Test B	Test C	Test D
1	5586	11657	11460	36680	50850
3	17053	34632	34012	111056	155295
10	58690	118099	117150	368600	506316
30	182868	370138	363825	1101031	1529114
60	345302	693399	685906	2230920	3135166

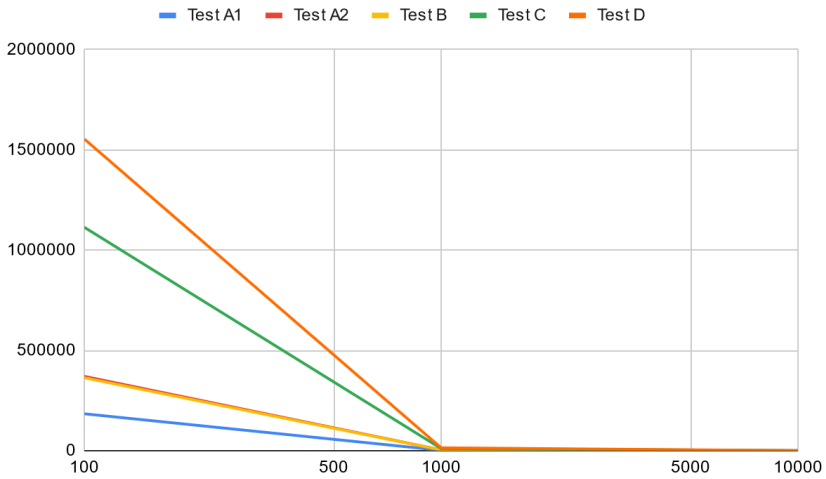
As is quite evident already from the numbers, Test D is by far and away the most efficient of the algorithms. At no point does it even get close to being matched by the second placed Test C.

The chart above shows executions over a number of seconds, and as can be seen over a range of timeframes, Test D is far, far more efficient than any of the other methods. It should be noted that Test A2 is barely visible because it's aligned almost exactly with B.

Please note that all testing was done using 'Release' settings with compiler optimisations turned off. Each run was done without user input or other programs being used.

For posterity's sake, I also created a second table with executions and number of boxes at a runtime of 30 seconds.

Number of Boxes	Test A1	Test A2	Test B	Test C	Test D
100	183992	370592	363515	1114120	1554522
1000	1758	3561	3456	10686	14281
10000	18	36	35	97	123



No notable changes to the previous chart, but the testing of a larger number of objects is probably more applicable than just time limit.

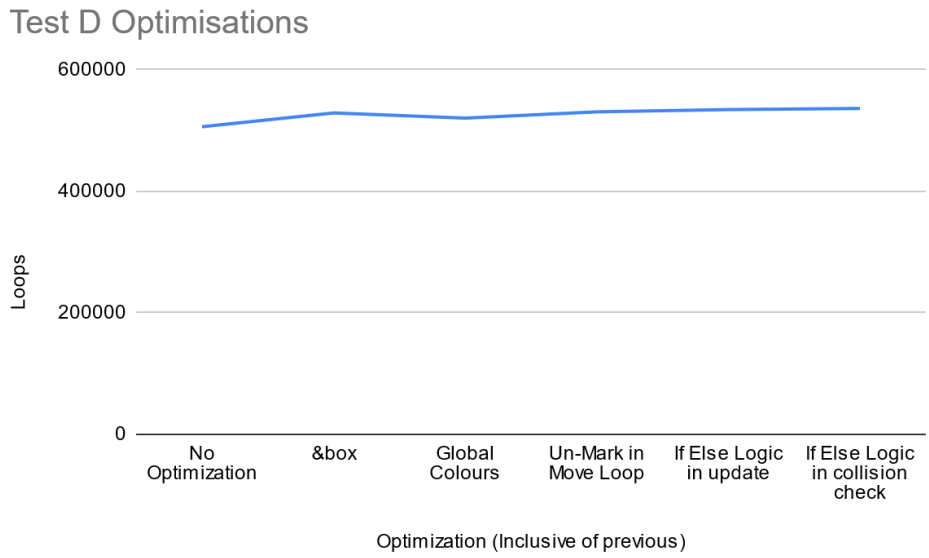
2. Adding Optimizations:

I had quite a bit of trouble with this consistency when attempting to test any optimizations I was adding. But here's a list of slight adjustments I made.

- Moved Colours to Globals
- Used If-Else logic in update_boxes
- Moved box collision reset to movement loop
- Pass box by reference in render

Wasn't really sure how to go about caching rects, I assume this would involve a map, but I've not gone back and done that, due to time constraints.

The results of the optimizations are as shown below.



3. Commit to Git: