**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:

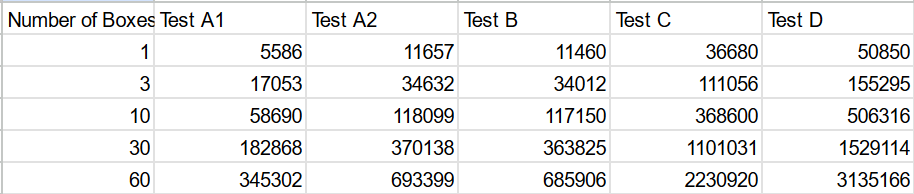
Method A1/A2: These tests take an int as a paremeter, declare 8 ints as box corner positions, 2 boxes copied from the box collection, then assign the corner positions based on box position, width and height, then proceeds to do the actual check. The end result is that I hate looking at it. It should 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.

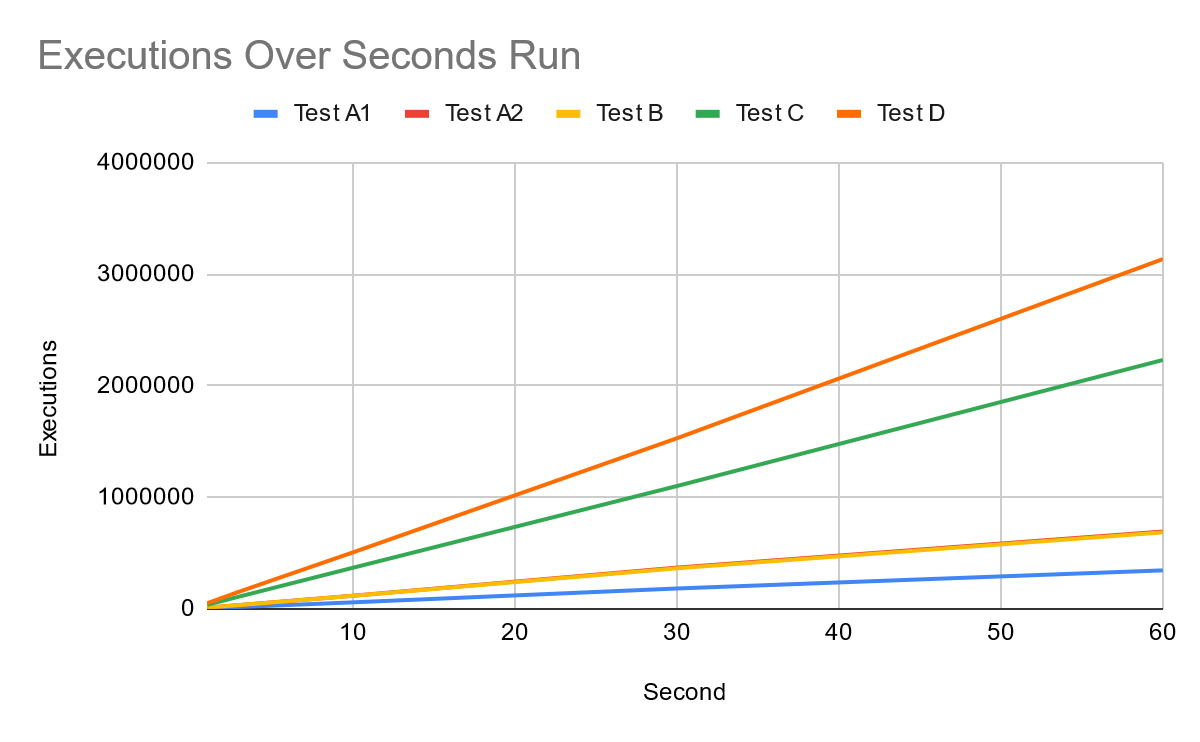
Method C: Similar to Method B, though now with a reference to the boxes as parameters instead. No unnecessary copying happening here.

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.



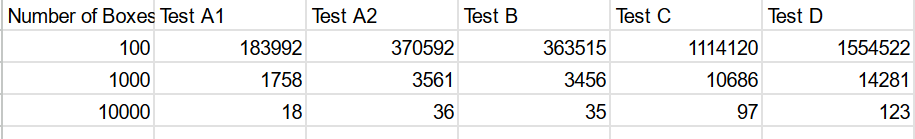
As is quite evident already from the numbers, Test D is by far and away the most efficient of the algorithms. At no point does it even get close to being matched by the second placed Test C.

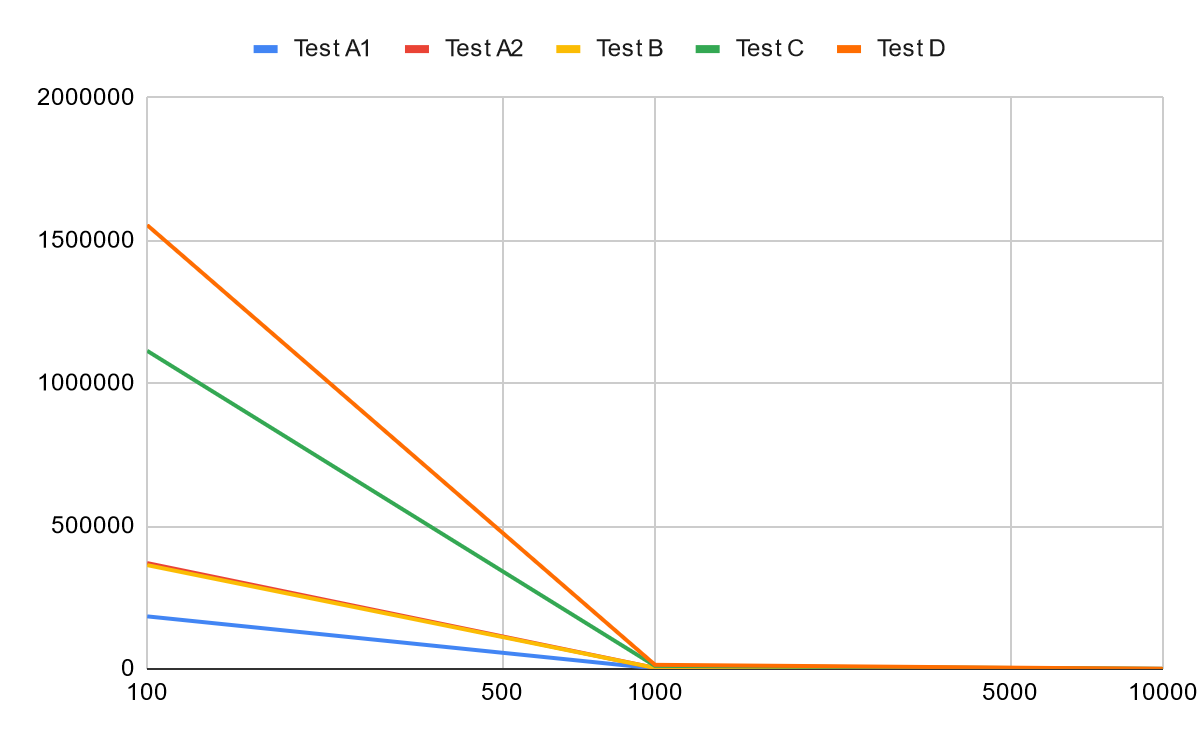


The chart above shows executions over a number of seconds, and as can be seen over a range of timeframes, Test D is far, far more efficient than any of the other methods. It should be noted that Test A2 is barely visible because it’s aligned almost exactly with B.

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.





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

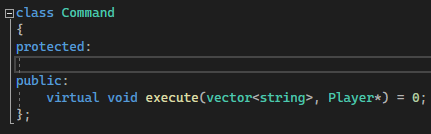
1. Implementing the Command Manager:

Llkj;klj;lkj.

1. Implementing Commands:

Commands are pretty easy, but there are a lot of them. So let's get to explaining these commands, one by one. Valus Ta'aurc. From what I can gather he commands the Siege Dancers from an Imperial Land Tank outside of Rubicon. He's well protected, but with the right team, we can punch through those defences, take this beast out, and break their grip on Freehold. Surely this won’t get me into any trouble with plagiarism detection.

The base command class is very simple.



It’s a virtual function. Wow!

Look: Gets the player’s current location and outputs the description.

A computer screen shot of a program code

Description automatically generated

Look At: Gets an item within the player’s current location and outputs the description.

A black screen with white text

Description automatically generated

Help: This is one of the commands that requires the use of the command manager. This command calls a function on the command manager that outputs all the commands available in the list.

A screen shot of a computer

Description automatically generated

Alias: Takes 2 inputs, then if the command exists, copies the command to a new entry in the list of commands.

A computer screen with text

Description automatically generated

Debug: Outputs as much information as is available and possible for me to access easily.

A computer screen shot of code

Description automatically generated

Go: Checks if the direction has been entered, if it has, then it checks if there’s anything in that direction. Once checked, it moves the player to that location via a function previously used in other tasks.

A computer screen with text

Description automatically generated

Inventory: Get’s the player’s inventory and outputs it to console.

A screen shot of a computer code

Description automatically generated

1. Commit to Git:

A blue background with white text

Description automatically generated