Profiling, Performance, Optimization

Name: Shiva Gupta
Student ID: 102262514
Unit Code: COS30031
Task Number: 24
Date: 31/10/2021
Repo link: https://bitbucket.org/ShivaGupta_/cos30031-102262514/src/master/24%20-%20Spike%20-%20Profiling%2C%20Performance%20and%20Optimsation/

Environment

This section contains information about the factors that may affect the performance.

- Battery: Connected
- Programs Running: VS Studio
- Compiler Optimizations: Off
- IDE Configuration: Debug 32-bit

Collecting Data

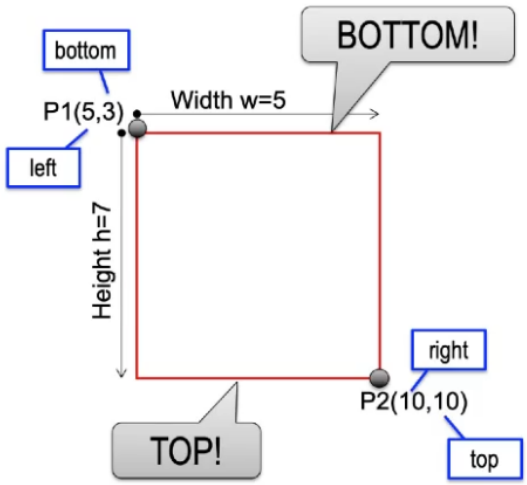
The method for collecting data has been documented in the spike report.

Collision Checking Approach

[Resource: 2D Collisions Lecture]

The collision checking approach that is being used in the task is the same throughout. The performance differences occur because of the method of handling memory for boxes. In this section we will learn about the collision checking approach used and then focus on the memory handling.

The boxes that are being used are AABB (axis-aligned bounding boxes). First, we calculate the edges of a box using the following approach.



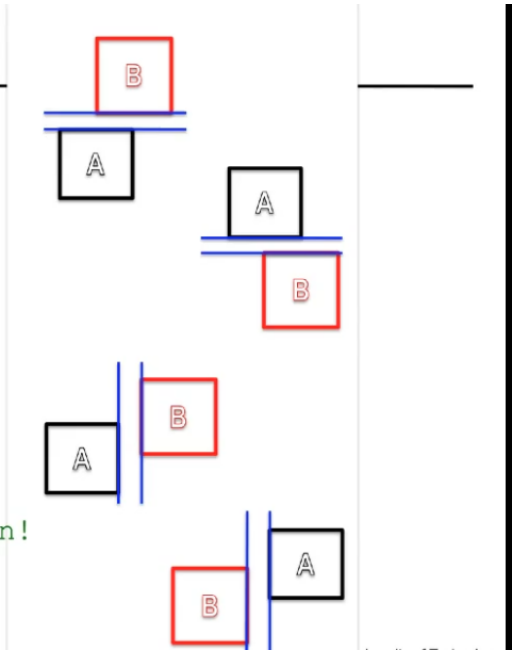

```
// So if box A is represented by
// a point (x,y), width (w) and
// height (h), we want sides...
int left, right, top, bottom;

bottom = A.y;      // = 3
top = A.y + A.h;   // 3+7 = 10
left = A.x;        // = 5
right = A.x + A.w;// 5+5 = 10
```

Then we check if any of the edges of A are outside B. If the edges are inside then there must be some collision.

AABB Collision Test

```
bool collision(AABB A, AABB B)
{
  // Any edges of A outside of B ?
  if( A.bottom <= B.top )
    return false;
  if( A.top >= B.bottom )
    return false;
  if( A.right <= B.left )
    return false;
  if( A.left >= B.right )
    return false;
  // Must be some overlap - collision!
  return true;
}
```



Methods of Collision Checking

In the task there are 5 different methods that test the collisions between AABB boxes. The boxes are initialized and stored inside a global array. The maximum number of boxes stored is 100. Before testing we initialize the boxes and store them in the `boxes` array. The boxes have random spawn position, velocity and dimensions.

Method 1: `crash_test_all_A1()`

1. In this method we iterate through the `boxes` array and check for collision. But there is an extra test to check if the two boxes being checked do not refer to the same box. If they are not then we check for collision and save their state.

```
if (crash_test_A(i,j)) {
                if (i != j) { //extra test
                    boxes[i].state = CONTACT_YES;
                    boxes[j].state = CONTACT_YES;
                }
            }
```

2. Then the index is passed in to the actual test and then tested.

```
bool crash_test_A(int i, int j) // via index
```

This is the **slowest** approach because of the extra test and the passing in the index of the boxes. This approach also double checks the boxes already checked against each other because the index always starts from 0. The only boxes it does not double check is the boxes with the same index (45, 45), (90. 90) etc.
If box A collides with box B then box B also collides with box A but this approach does not implement this logic.

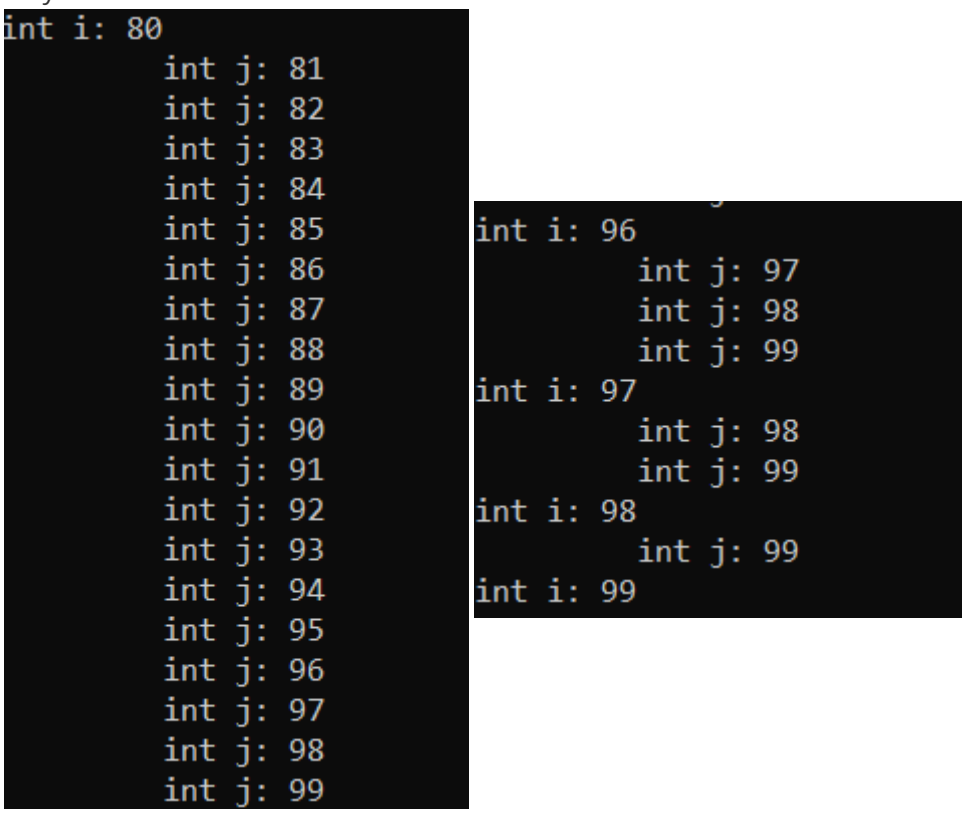
Method 2: `crash_test_all_A2()`

This method uses the same crash test method, that is, testing the collision by passing in the index but it removes the need for checking for different indexes.

1. In the for loop it initializes the index for box B by always taking the next index. This ensures that the boxes checked against each other are not tested again.

```
for (int i = 0; i < BOX_COUNT; i++) {
    for (int j = i+1; j < BOX_COUNT; j++) {
        //j = i+1
```

This approach decreases the test conducted by a large number because in the later tests only boxes that have not been tested are tested.



This method is the **second slowest** method because it still passes indexes of the boxes array to the collision testing function.

Method 3: `crash_test_all_B()`

This method uses the same approach for accessing positions of the boxes but it makes a duplicate of every box for testing and then returns the result.

```
if (crash_test_B(boxes[i],boxes[j])) {

bool crash_test_B(CrashBox A, CrashBox B) // struct (copy)
```

This approach is the **third slowest** because it **duplicates the boxes** but it still checks only for different boxes.

Method 4: `crash_test_all_C()`

This method uses a better way of using the already initialized boxes and then testing them for collisions.

1. It achieves this by passing the boxes by reference.

```
bool crash_test_C(CrashBox &A, CrashBox &B) // via struct (ref!)
```

2. But, it still calculates the edges separately for box A and box B.

```
    //The sides of the rectangles
    int leftA, leftB;
```

```
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Calculate the sides of rect A
    leftA = A.x;
    rightA = A.x + A.w;
    topA = A.y;
    bottomA = A.y + A.h;

    //Calculate the sides of rect B
    leftB = B.x;
    rightB = B.x + B.w;
    topB = B.y;
    bottomB = B.y + B.h;
```

This method is the **second fastest** method because it uses the same boxes for testing and does not test the same boxes again but it still calculates the edges first and then checks for collision which can be removed.

Method 5: `crash_test_all_D()`

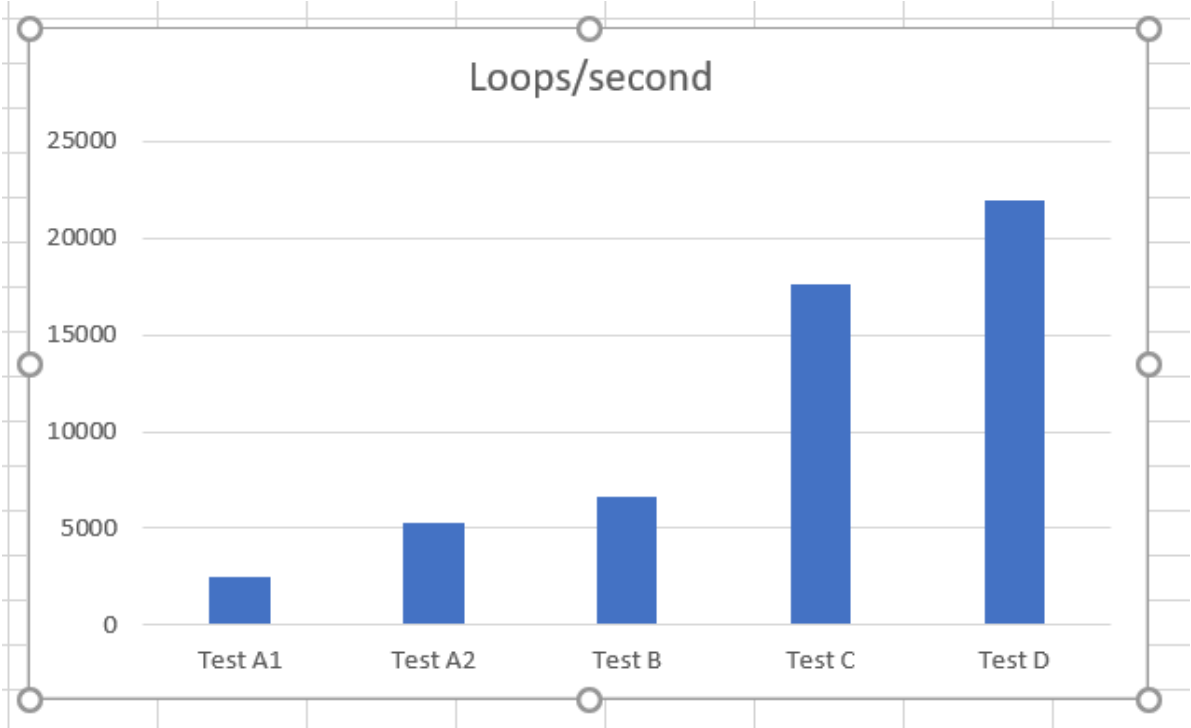
This method uses all the positives of the other method and calculates the edges while testing for collisions and not before.

```
bool crash_test_D(CrashBox &A, CrashBox &B) //references
{
    //calculate the edges while testing
    if ((A.y + A.h) <= B.y) return false;
    if (A.y >= (B.y + B.h)) return false;
    if ((A.x + A.w) <= B.x) return false;
    if (A.x >= (B.x + B.w)) return false;
    return true;
}
```

This makes it the **fastest** method amongst all others.

Results

The graph below documents the results when all the methods are ran for 10 seconds. The approach is how many times a method can run in a given time t. Here t is 10 seconds.



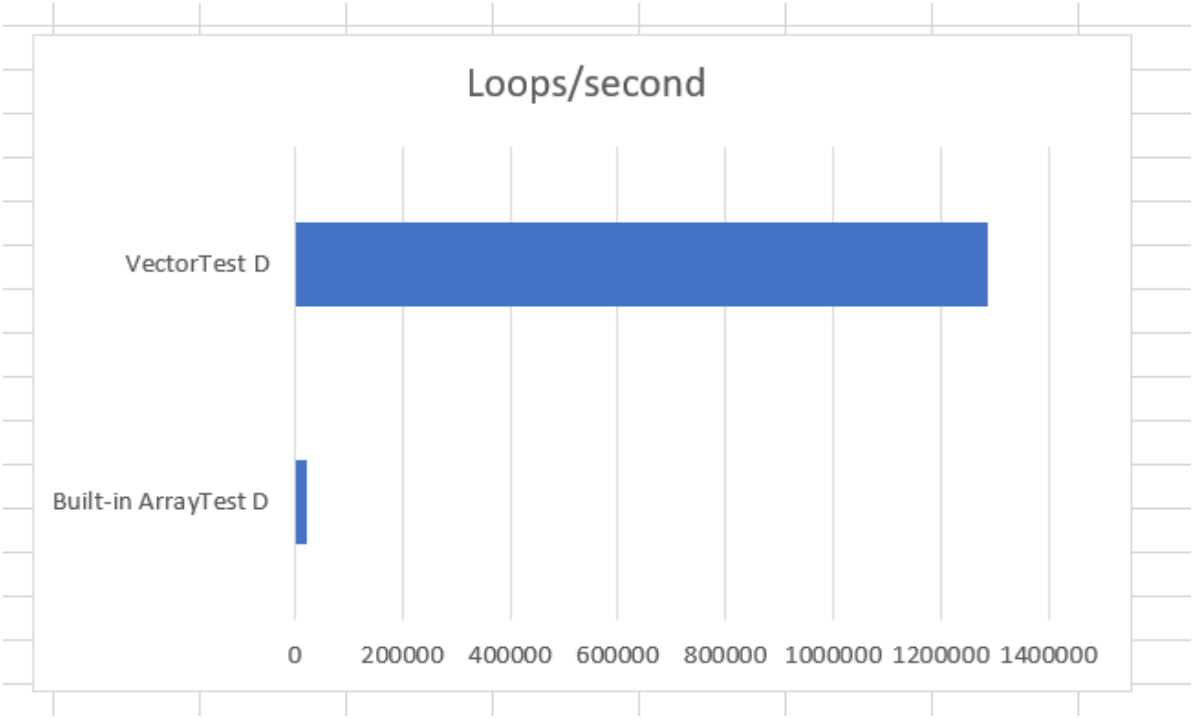
Title	Loops	Time(ms)	Loops/second
Test A1	26031	10000	2603.1
Test A2	53866	10000	5386.6
Test B	67123	10000	6712.3
Test C	176429	10000	17642.9
Test D	225740	10000	22574
Test E	254203	10000	25420.3

1. Using references while testing gives us great performance results. (Difference between Test B and Test C).
2. calculating the edges while testing also greatly improves the performance. (Difference between Test C and Test D).

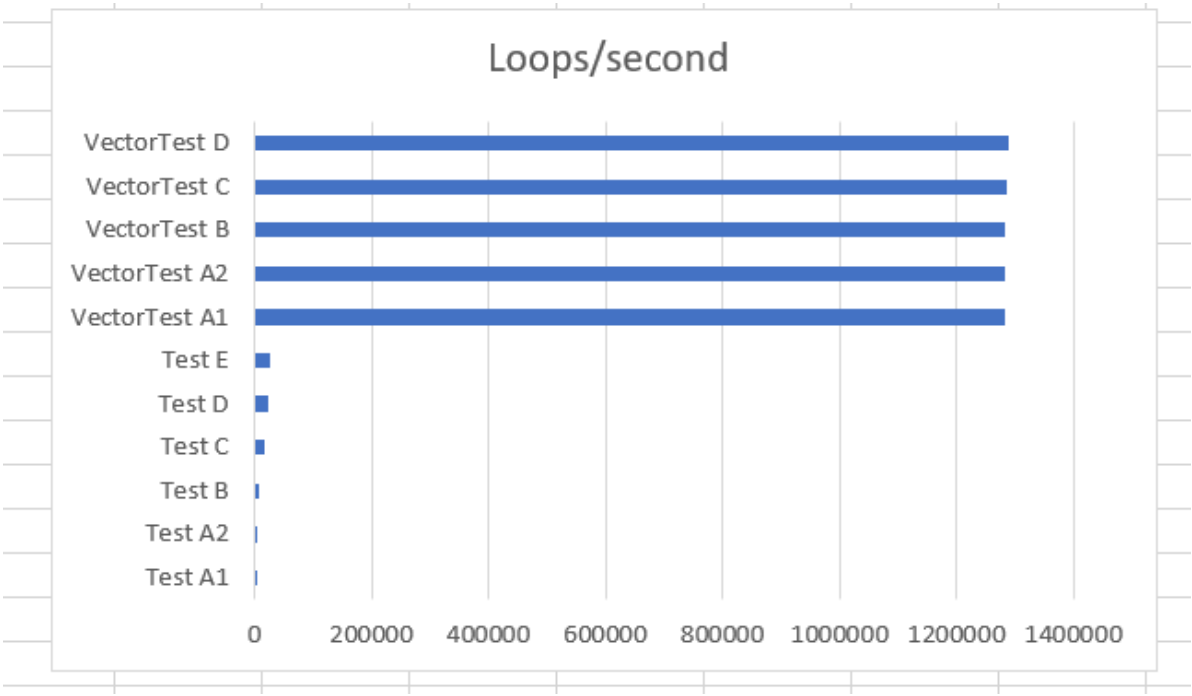
Optimizations

In the methods used approach D produces the most performant results but this method can further be optimized using these approaches:

The current global structure being used is built-in array but if we change it to use **STL vectors** then we can massively improve the performance.



In fact, it improves the performance for all the tests.



Title	Loops	Time(ms)	Loops/second
Test A1	26031	10000	2603.1
Test A2	53866	10000	5386.6
Test B	67123	10000	6712.3
Test C	176429	10000	17642.9
Test D	225740	10000	22574
Test E	254203	10000	25420.3
VectorTest A1	12821875	10000	1.28E+06
VectorTest A2	12836954	10000	1.28E+06
VectorTest B	12846663	10000	1.28E+06
VectorTest C	12853767	10000	1.29E+06
VectorTest D	12901756	10000	1.29E+06

What we found out

After collecting data, measuring the performance and optimizing the collision testing approach we found out that:

1. The STL vectors are faster than built-in arrays.
2. Passing by reference is more efficient than passing by value.
3. implementing additional logic while testing can massively improve performance like `If A collides with B => B collides with A`
