Task 24 Short Report
Ryan Chessum 102564760

There are 4 different functions that test for collision between two rects.

```cpp
bool crash_test_A(int i, int j) // via index
{    //yeap - lazyfoo style!

    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    CrashBox A, B;
    A = boxes[i];
    B = boxes[j];

    //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;

    //If any of the sides from A are outside of B
    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;

    //If none of the sides from A are outside B
    return true;
}
```

The first function, test A, gets the two boxes it wants to get collisions for by having the indexes for the global array of boxes passed to it. It uses the indexes to get each box and then use each bot to assign values to each integer variable for use in the collision test.

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

    //If any of the sides from A are outside of B
    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;
    //If none of the sides from A are outside B
    return true;
}
```

Crash test b passes in two boxes to be compared as variable to use in the test. They are passed as regular variables, so the boxes passed in are copied and reconstructed for use in the function scope.

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

    //If any of the sides from A are outside of B
    if (bottomA <= topB) return false;
    if (topA >= bottomB) return false;
    if (rightA <= leftB) return false;
    if (leftA >= rightB) return false;

    //If none of the sides from A are outside B
    return true;
}
```

Crash test C uses references to each box, meaning they do not have to be reconstructed.

```cpp
bool crash_test_D(CrashBox& A, CrashBox& B)
{
    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;
}
```

Crash test D also uses constant references but instead of creating local variables and assigning them to the data needed for the calculations the math to work out the bottom and right of the rect is done in the if statement.

I ran the program 30 times and recorded the average number of times the game loop can be called in a second.

I ran the tests for 3 seconds, 5 seconds and 10 seconds, 10 times per duration.

	A1	A2	B	C	D
3000	2396.333218	4995.666504	4174.666882	5449.6665	8919.66629
	2433.333397	4907.666683	4105.333328	5425.666809	8869.667053
	2470.999956	4980.666637	4164.333344	5432.333469	8882.333755
	2447.333336	4954.666615	4128.666878	5415.333271	8876.000404
	2470.999956	4935.99987	4156.000137	5428.666592	8871.00029
	2461.999893	4986.666679	4169.333458	5435.99987	8932.999611
	2463.666677	4978.666782	4174.333096	5433.000088	8966.333389
	2470.000029	4890.999794	4038.000107	5389.333248	8892.000198
	2460.666656	4933.000088	3964.333296	5326.666832	8750.666618
	2459.33342	4976.333141	4135.000229	5386.000156	8855.999947
5000	2544.60001	5185.200214	4199.399948	5489.999771	9014.5998
	2547.199965	5080.999851	4159.599781	5486.599922	8951.000214
	2545.200109	5135.60009	4177.599907	5379.000187	8811.599731
	2498.399973	5085.000038	4108.200073	5289.999962	8888.199806
	2530.600071	5105.59988	4164.199829	5375.199795	8799.599648
	2515.1999	5101.60017	4140.600204	5435.599804	8927.200317
	2536.600113	5128.799915	4144.40012	5408.400059	8808.199883
	2515.1999	5118.800163	4166.800022	5474.400043	9013.400078
	2526.99995	5154.799938	4147.799969	5460.400105	8924.599648
	2517.800093	5095.399857	4120.200157	5418.000221	8867.799759
10000	2717.099905	5538.199902	4103.199959	5450.799942	8835.300446
	2753.900051	5568.200111	4062.399864	5457.099915	8943.099976
	2776.900053	5595.699787	4127.500057	5476.200104	9008.500099
	2785.79998	5596.600056	4110.70013	5468.299866	8977.499962
	2774.300098	5611.700058	4118.000031	5480.29995	8961.400032
	2784.300089	5592.700005	4122.900009	5492.499828	8963.80043
	2774.499893	5564.799786	4110.499859	5449.60022	8911.800385
	2781.899929	5620.299816	4124.499798	5497.600079	8948.300362
	2784.499884	5600.599766	4087.39996	5513.000011	8959.799767
	2795.599937	5605.700016	4139.100075	5485.799789	8988.200188

The program the results of each test to the console. So I added a loop to the main function to run the tests 10 times. I then copied the data into a spreadsheet table. Test A is ran twice as the second time it is called it runs faster.

Distribution of loops persecond
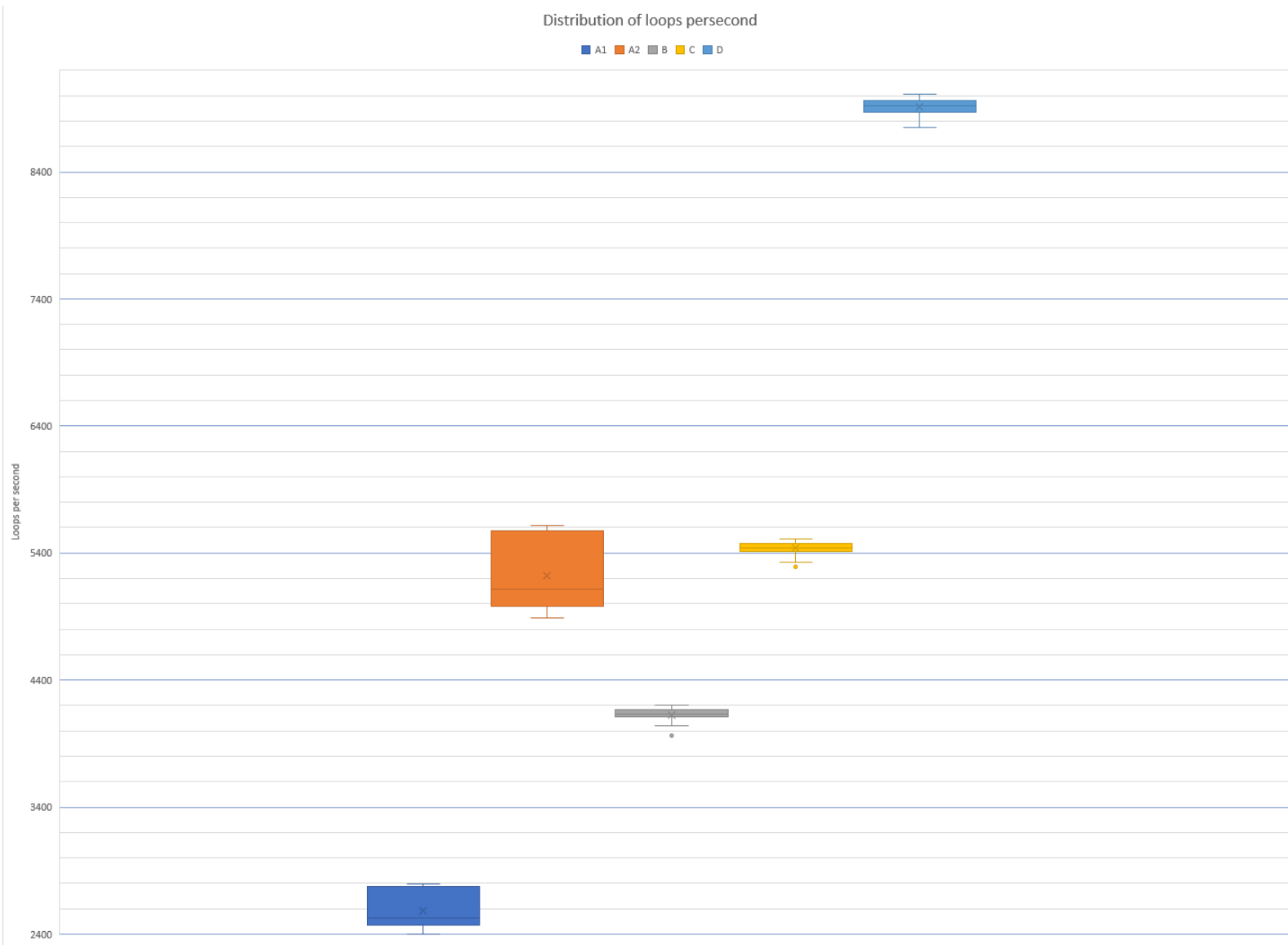
I made a box and whisker plot of each dataset to compare the average performance for each test. Interestingly the A2 tests are almost on par with the C tests, though the c tests are much more consistent. However, the D tests are the most efficient by far, consistently running the loop just below 9000 times per second as nothing has to be initialised. A1 is super slow for some reason, there must be some sort of setup that the program has to do first. B is also pretty slow as the construction must take up a fair bit of time.

From the data we can tell that using pointers where possible and avoiding unnecessary construction is a good way to improve upon performance.