DS2 Project Report

Rayyan Ul Haq, Rizwan Niaz Khajar, Shehryar Mughal
Abdullah Iqbal, Bilal Mohiuddin

April 2019

Introduction

Our project is collision detection of axis aligned 2 dimensional rectangles. Collision detection is a very wide field where is there is a constant lookout for better and better algorithms. Visually it is very easy for a person to decide whether two objects are colliding or not but to construct an algorithm is a different challenge altogether. Therefore many algorithms have evolved over the years for collision detection. We will develop and test algorithms for a very simple case, one where our objects are rectangles which are parallel to the x and y axes, as the problem becomes rapidly more difficult as the orientation, shape and dimension of objects changes. The five algorithms along with their data structures we will use are: Narrow Phase collision detection using arrays, Narrow Phase collision detection using Skiplist also known as 'sweep and prune', Spatial Hashing/Uniform grid broad phase collision, Quad Tree broad phase collision and Binary Tree broad phase collision.

Array based narrow phase

This approach is very naive and has a simple algorithm: each object is compared to every other object in the list to determine every pair of colliding objects. The complexity of this will therefore be $O(n^2)$. The collision in our case of axis aligned rectangles is simply done through checking if the x projection of both rectangles overlap and then the y projection, returning true if both are overlapping. This collision detection mechanism however is critical as every other algorithm makes use of it after narrowing down the list of objects to check so that not every pair is compared.

Skiplist based narrow phase

This approach is an improvement over the array based approach as in a SkipList, the add operation adds in sorted order. The add operation in a SkipList is expected to be $O(\log n)$ Since each add operation takes $O(\log n)$ time, the total complexity for adding n elements will be $O(n\log n)$. We are adding and sorting rectangles based on the x coordinate of their top left corner. Hence we will

have a linked list sorted in terms of the x coordinate of the top left corner of each rectangle at each frame of the program. The collision detection is done by comparing each rectangle with every rectangle to its right in the SkipList up until we reach a rectangle it does not collide with. This is done because since the SkipList is sorted in terms of the x coordinate of the top left corner of each rectangle, if we reach a rectangle our current rectangle does not collide with, we will be sure that all the remaining rectangles will also not collide as they are at a larger x coordinate. Keeping this in mind, if our rectangles are dispersed on the screen, then on average each rectangle will collide with one other rectangle meaning we will have n comparisons. Hence, the total complexity of both add and checking collisions will be $O(n\log\, n + n)$. If more gather closer and closer, then the complexity will converge to $O(n^2)$ but this does not happen very often.

Hash based broad phase

In our algorithm, we use a spatial hashing based broad phase to partition our space, the broad phase refers to identifying all of the objects that are potentially colliding. By doing this we ensure that we do not have to perform a large number of collision checks. The primary purpose of this phase is to identify for certain which objects are not colliding, so that we do not have to consider them when we perform our collision check.The Spatial Hashing method, also known as the Uniform Grid Method, works by partitioning our space into equal sized blocks. Each block represents a "bucket" which corresponds to the number of the block on the grid. We then iterate over a list of all the rectangles in our space and perform a hash function on each of them that maps them into a "bucket" based on their position in the grid (If an object lies in more than one block it is simply mapped onto all of the "buckets" that correspond to them). Thus objects that are in the same block in the space are mapped onto the same bucket. After that, we simply have to perform collision checks between objects that lie in the same bucket. Since our hashing function takes constant time, the addition of one element has a complexity of $O(1)$. Since we must add n elements, the time complexity of mapping all of the rectangles into their corresponding buckets is $O(n)$. Assuming all of the elements are evenly spread out i.e only one object is mapped onto each bucket, our best case time complexity becomes $O(n)$. If all of the objects are concentrated into a single region, i.e they are all mapped onto the same bucket, the number of collision checks we must perform converges to $O(n^2)$ since each object must be checked for collisions with every other object. Therefore, our best case time complexity turns out to be $O(n)$ while our worst case time complexity is $O(n^2)$.

Quad Tree based broad phase

This is also a broad phased approach which means it works by narrowing down and partitioning the list of objects that need to be checked against each other. This algorithm does this by taking a list of rectangles and creates a root node which covers the entire space and contains all the rectangles, representing that

all the rectangles part of the node lie strictly within the space defined by the node. If the number of rectangles in a node exceeds a certain threshold, in our case 7 then the node creates four children and assigns each of them a quadrant of the space it occupies it then shift as many rectangles as it can from its self to its children. However the rectangle that is to be shifted must lie strictly in the region of space defined by the child. If the rectangle that is to be shifted happens to span a space such that it cannot strictly lie in the space of any one child then the rectangle remains at the parent node. Their is also a limit to the minimum space a node can have. Thus the space is recursively divided into many sections with greater division where there are a larger number of rectangles. Each section contains potentially colliding rectangles which must be checked using narrow phase detection. When this is accomplished, to check for collisions we only need to check the rectangles that lie at a node with each other and against the rectangles lying in the path to the root from the node.Adding all the n rectangles of the tree will have a expected theoretical complexity of $O(n\log\ n)$. We also expect that after completing add the rectangles will be spread out with in the tree and each leaf will contain one or two rectangles and with hardly any rectangles on the path to the root as most of the rectangles will lie with in the space and rarely on a partition. Thus the checking for collisions will take $O(n)$ time and the overall complexity becomes $O(n\ \log\ n + n)$ giving $O(n\log\ n)$ as the final complexity. The worst case can occur if the rectangles are too concentrated and all end up in the same leaf node in which case the complexity will be $O(n^2 + n\ \log\ n)$ equivalent to $O(n^2)$.

Binary Tree based broad phase

Binary trees broad phase is very similar to quad trees but has two important differences that we will discuss a bit later. Firstly let see how it divides the space. In a similar fashion as quad trees the node of a binary tree stores the space it spans and the list of rectangles but in this case the space is divided in two parts instead of four when the node decides to subdivide. Further the division can occur along the y or the x axis, for simplicity it parts based on which ever it spans more. This leads to the binary tree having greater height about 2 to 4 times as compared to a quad tree but it has to do less comparisons, only against two children, when splitting a node, in contrast to the four children of quad tree. The point of this was to check if given these two differences will the increase in depth outweigh the performance cost or will the less number of comparisons result in it being faster. The theoretical complexity is still $O(n\log\ n)$ although the log may have different base in binary and quad. The worst case is the same as a quad tree.

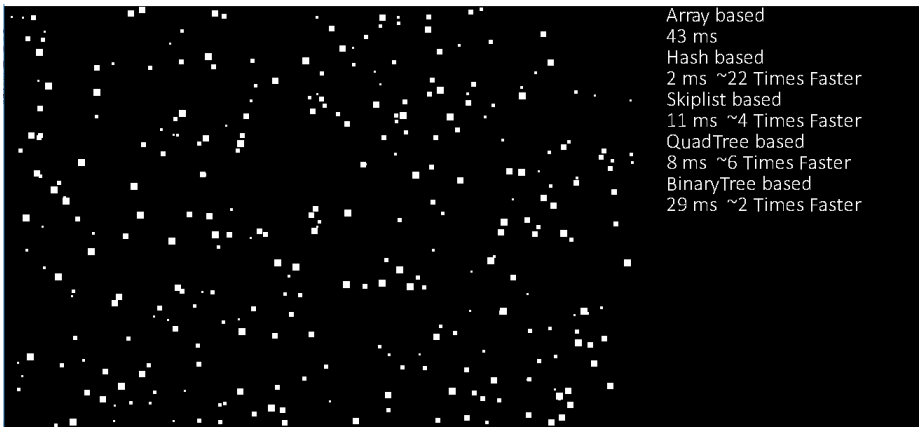
Scenario

Each of the data structures inherits from a collision detection class which already contains a function to check the collision of two rectangles which all of the

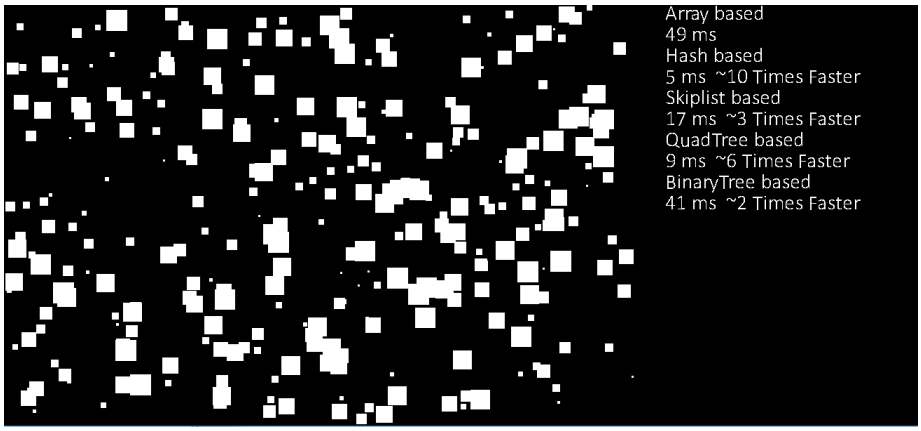
algorithms use in or after broad phase to perform narrow collision checks. A list of rectangles is passed to the function and the function returns a set of sets for every pair of colliding objects. The return pattern is set so that the output of all algorithms can be compared to each other. The algorithms were run on varying number of rectangles where the width and height, and position are randomly determined for each rectangle. The time was noted in milliseconds for each data structure. The default python random number generator was used for this. The choice of the generator we believe will not greatly affect our result and therefore we used the default. We further experimented by raising both the min and max length limits for the rectangles and observed the resulting changes. Another factor was to control the spread of the rectangles, concentrating them for some cases and spreading them out for others.

Results

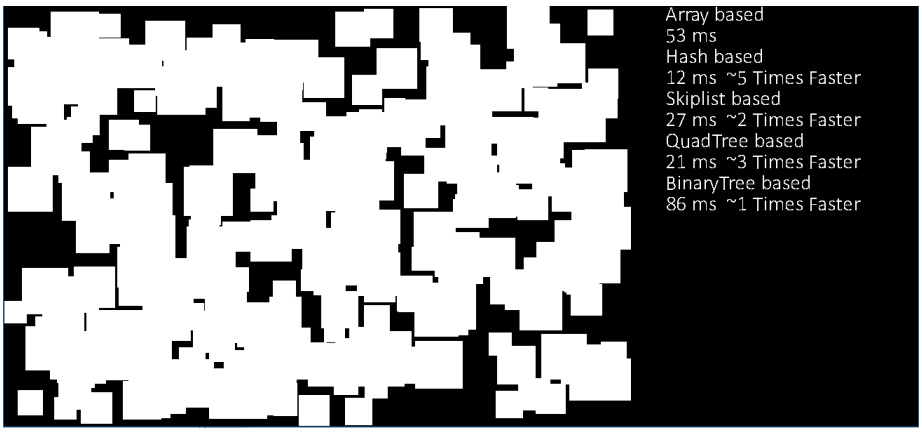
The first case we ran was that the rectangles are small sized and well spread out through out the space as shown below: small_spread



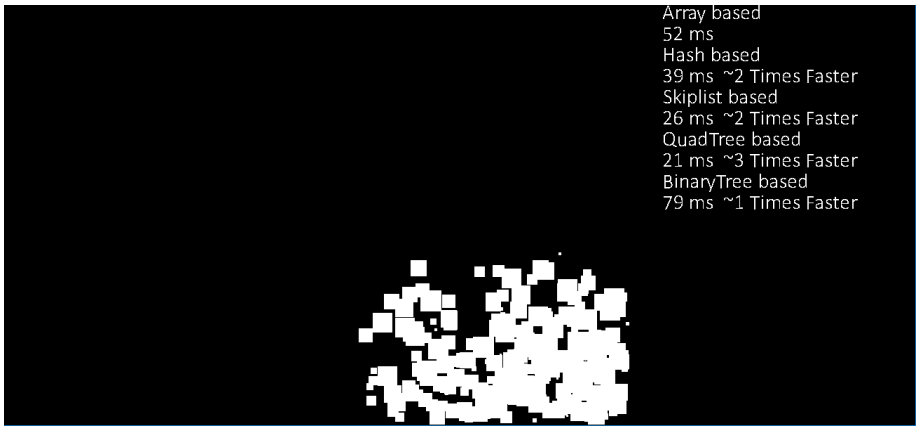
We see that all the other algorithms are faster than n^2 array based. Let us try doing this using different box sizes: medium_spread



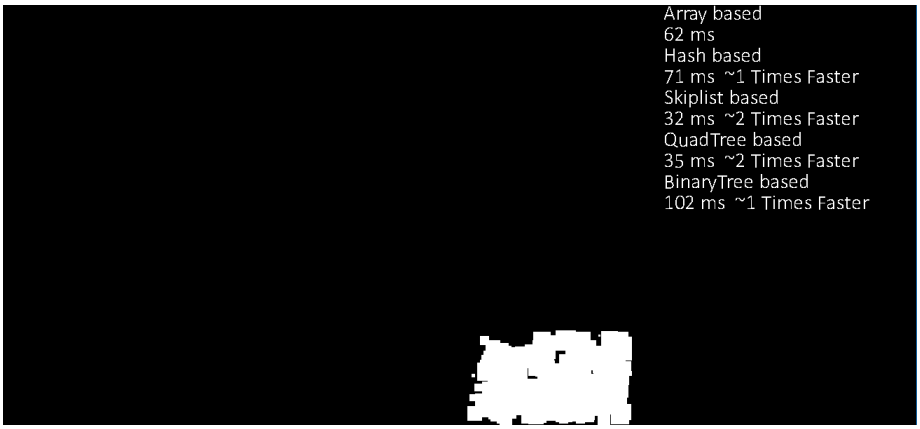
Here we see that all the algorithms except Array and QuadTree have suffered. Let us try increasing the sizes even further: big_spread



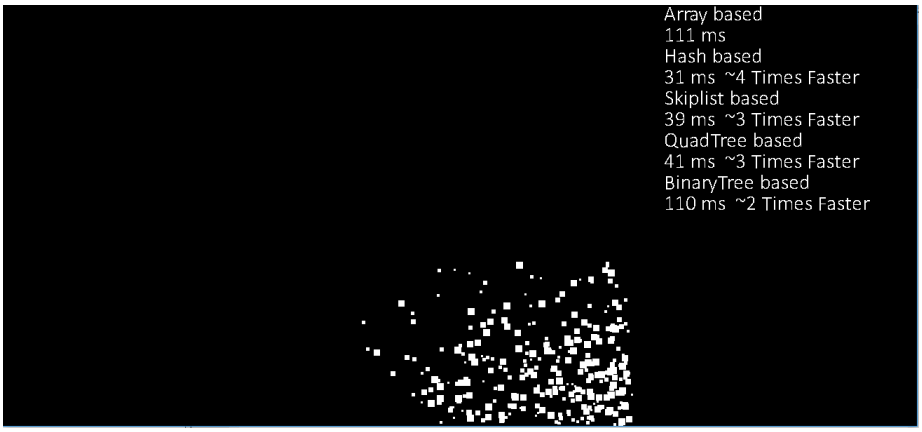
We see that all the algorithms except Array based have suffered even more with the BinaryTree even taking more time than the naive approach of Array. Since this looks very crude and we have already established that the increase in size of rectangles adversely affects the algorithms we will now continue to experiment on medium and small sized rectangles. Next follows a shot of medium sized blocks concentrated in a quadrant: medium_concentrated



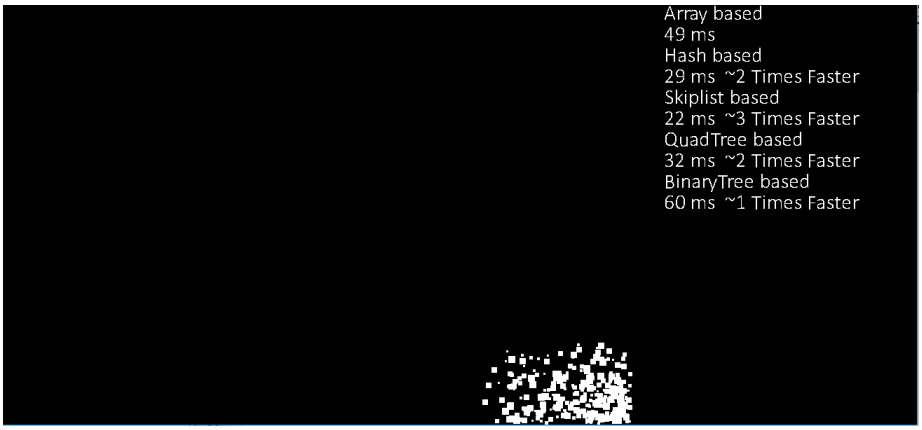
As we can see, concentrating the boxes into a quadrant is even worse for the algorithms than increasing the size of the boxes. Next follows a shot of further concentrating the medium boxes: medium_concentrated_super



This has proved very fatal for Hash and BinaryTree both of which are now taking more time than naive Array. The skiplist and QuadTree seem to be holding out a little better. Next follow pictures of trying to concentrate small sized rectangles: small_concentrated

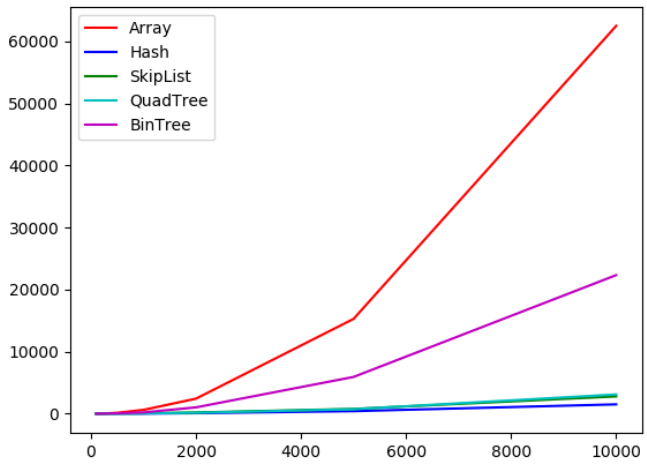


small_concentrated_super

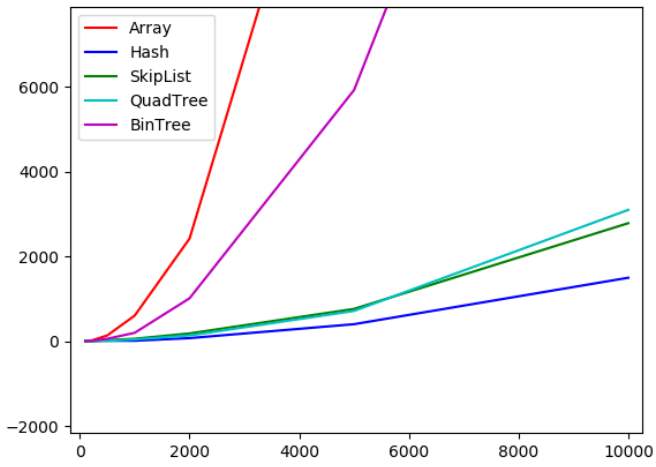


small_concentrated_super_super

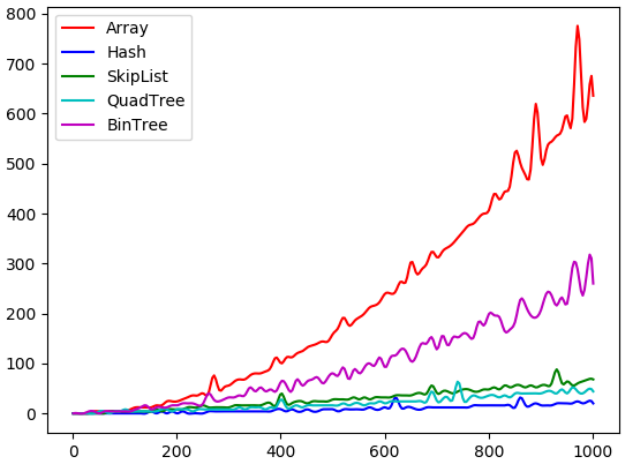
As can be seen from the last picture for smaller sized rectangles it took a much smaller area for the concentration of rectangles to become so high that the algorithms began failing as in the case medium_concentrated_super. Now we will perform a general analysis using varying numbers of small spread out rectangles so that the true potential of the algorithms can be measured. The following graph shows the number of milliseconds the algorithm took on the y axis against the number of rectangles on the x axis. The curve has smoothed using python.



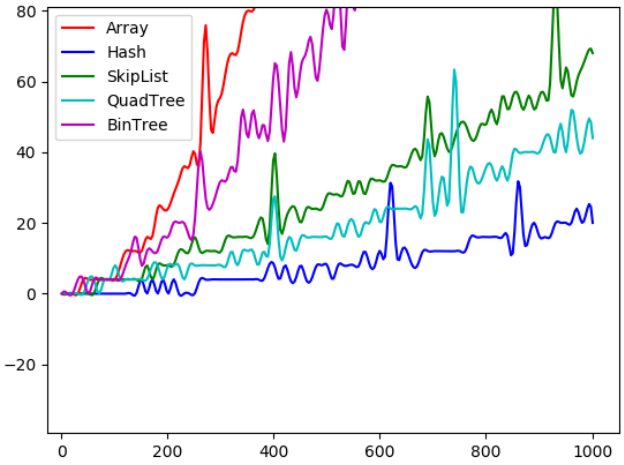
Zoomed to lower section



The upper cap of 10000 was chosen because it proved sufficient enough to measure the performance of the algorithms and beyond it the algorithms took far too long to be feasible. Let use now look at a detailed analysis with a cap of 1000. Curve has been smoothed using python



Zoomed to lower section



Inference

From our initial experiments we know that the quad tree, binary tree and hash perform worse when the objects become concentrated as their buckets contain too many objects for their effectiveness to be utilized. Increasing the size of rectangles has a similar effect as then the rectangles lie in multiple buckets and if we increase the size of the buckets we are faced with a similar problem of concentration evident from the run times of quad trees. A very subtle thing one can note over here is that the skiplist always performs better than the array even though the other structures become slower than the array at bad cases.

Looking at the graph for increasing number of rectangles we see that some of our predictions were true as the graph of array follows a $O(n^2)$ complexity, binary tree follows a $O(n \log n)$ complexity and the hash follows an approximate complexity of $O(n)$. However the quad tree and skiplist seem to be very close to the hash function, deluding us to think that they too are following a complexity of $O(n)$, however, in the zoomed picture we see them separate out from the hash function and we can conjecture that for even larger number of objects both of them will follow $O(n \log n)$ complexity.

From the graph of detailed analysis we see that the complexity we found in the previous graphs still holds in this graph as well. However, we see spikes and dips in this graph and its zoomed section from which we can see that for relatively smaller number of objects, which is usually the case in the foremost application of collision detection: games, the algorithms run quite closely and the random spikes and dips make it difficult to say which algorithm will run better as it is scenario dependent. For larger values however the difference becomes quite clear.

Conclusion

In the case of small evenly spread out objects hashing is the best function with a time complexity of $O(n)$ on average and a space complexity of $O(4n + b)$ where b is the number of buckets supported by out empirical analysis.

In the case of small spread out objects not necessarily evenly quad trees are in theory the best choice with a time complexity of $O(n \log n)$ since hashing should start gaining a larger complexity. However, this is different from our empirical analysis and we believe that is due to a combination of scenario and poor implementation of quad trees. The quad trees need to update every rectangle they contain but because all our rectangles are moving in every frame and need to be updated the complexity becomes akin to creating a quad tree from scratch. The time taken by quad trees is the second lowest in the detailed analysis graph but interestingly looses to skiplist in the broad analysis graph presumably because it starts hitting bad cases The space complexity is $O(2n)$.

The binary tree completely and utterly fails in comparison to the quad tree,

however this does not mean the quad tree is a superior data structure in fact the literature says that for spaces which are not in simple shapes or for objects not in simple shapes the binary tree performs as well and better partitions the space compared to the quad tree and is merely not in home-ground for our scenario.

Another thing we can see is that for none of the cases we tried the skiplist ever performed worse than the array method even though the other data structures performed worse than the array at some point or another. Further the time complexity of skiplist is comparable to our less efficient quad tree and thus may be good alternative in almost every circumstance to the array approach, even when the other structures are not. The space complexity is $O(3n)$