An Improvement on Axis-Aligned Bounding Box Collision Detection via Sorting and Axis Intervals

Jamin Eisenberg
ECE Department
Northeastern University
Boston, US
eisenberg.ja@northeastern.edu

***Abstract*—The focus of this paper is improving an algorithm for 2D collision detection that uses axis-aligned bounding boxes. The original algorithm (found in [2], another IEEE paper) works, but has a problematic time complexity, because it makes some comparisons between every shape and every other shape. This makes it very slow with many shapes. The improved algorithm that I created takes advantage of sorting, and turns the problem into a slightly different one, involving shapes' spanning intervals on a particular axis. The improved algorithm performs faster than the original in every case.**

***Keywords*—collision detection, 2D physics, simulation, rigid body, axis-aligned bounding box, optimization, interval**

I. Introduction

The main goal of this project is to implement and optimize a pixel-perfect 2D collision detection algorithm. The algorithm is supposed to be able to evaluate a group of shapes and determine which of the shapes are intersecting. Two shapes are said to intersect if any of their lines are intersecting. The concrete goal of the improved algorithm was to have no significant framerate lag when there were 100 shapes on the screen – a goal that is not met by the original algorithm from [2]. The auxiliary goals of the project include drawing shapes to a screen and simple collision handling – updating the trajectories of shapes after they collide with other shapes.

The hardware used follows:

- DE1-SoC: does all the computation and storage necessary for the project
- Keypad: 16 buttons that allow the user to create shapes and control the program
- Speaker: alerts the user when objects collide
- VGA port: allows the DE1-SoC to communicate with an external monitor for drawing

The work that I do in this paper improves the speed of collision detection greatly. Rapid collision detection has an impact on any field or project that is simulating physics in a reasonable amount of time. A popular application of this is video games, which attempt to simulate realistic physics in real-time. This paper makes a step towards improving the speeds of these simulations.

II. Algorithm Implementation

A. Detecting AABB collisions

From [2], I used the algorithm labelled "algorithm 1": an axis-aligned bounding box (AABB) approach. This will be referred to as the "original algorithm", and my improved version will be referred to as the "improved algorithm". The original algorithm is extremely simple, though it does require an understanding of the term "AABB". An AABB is a rectangle that fully encloses a shape and is oriented such that its lines are parallel to the x and y axes. To calculate the AABB for a given shape, I can simply go through all the shape's points to determine the minimum x and y coordinates, and the maximum x and y coordinates. The two points that these values create are the corners of the AABB. An example is shown in Fig. 1.

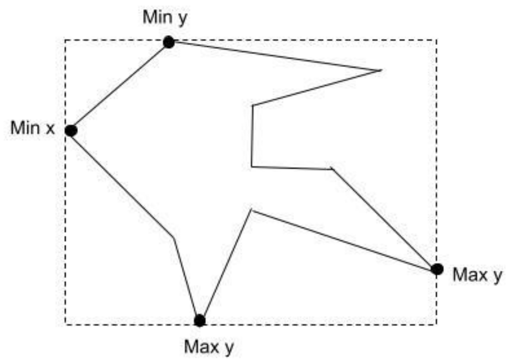

Fig. 1. Example of a complex shape's AABB

Note that this is a reduction of the shape. If two shapes' AABBs are colliding, that does not necessarily mean the shapes themselves are colliding. This will be elaborated on.

This reduction simplifies the problem significantly. Now, according to [2], all I need to do to find out if two AABBs are overlapping is to evaluate two Boolean expressions: if (1) and (2) are both true, then a collision has occurred.

$$Len1 \leq (R\_Width + R'\_Width) / 2 \tag{1}$$

$$Len2 \leq (R\_Height + R'\_Height) / 2 \tag{2}$$

$$Len1 = |R(x) - R'(x)| \tag{3}$$

$$Len2 = |R(y) - R'(y)| \tag{4}$$

Here, Len1 and Len2 are the x and y differences between the two shapes' positions, given by (3) and (4), respectively.

These equations are easier to understand in the context of the program. Equations (1) through (4) basically say that two AABBs are colliding in the x-direction if and only if the sum of the half-widths of the boxes is greater than the x-distance between them. The same idea holds in the y-direction. In every case, the half-width represents the distance from the edge of the shape to the central position of the shape.

At this point, I know how to determine if two shapes' AABBs are colliding. To use the algorithm for our purposes, I simply need to check if any shape is colliding with any other shape. This can be done by iterating over every shape, and for every shape, iterating over every other shape, and using the above expression to determine if they are colliding. In other words, I use a nested for loop to check collisions between every shape and every other shape.

The original algorithm is clearly the best outlined in [2]. From the paper, "It can be easily found that the time consuming of Alg.1 is least." Additionally, it is extremely simple to implement, which is part of why it is fast. It only uses cheap operations, like addition, subtraction, and Boolean operators. The sole weakness of the algorithm is not in checking whether two AABBs are colliding; instead, it is the fact that I need to check every shape with every other shape. If I have *n* shapes, that makes for n^2 checks. This scales very quickly. Checking 10 shapes makes for 100 checks, but 100 shapes is 10,000 checks! This is exactly where I propose improvements can be made.

B. *Detecting Collisions Generally*

Due to the simplicity of the algorithm, I was able to implement it myself easily. There were several parts to doing this. I prefer top-down design, so the first part was defining how to check whether any two shapes are colliding. I chose to store an array of shapes in my program, so the check looked like Fig. 2.

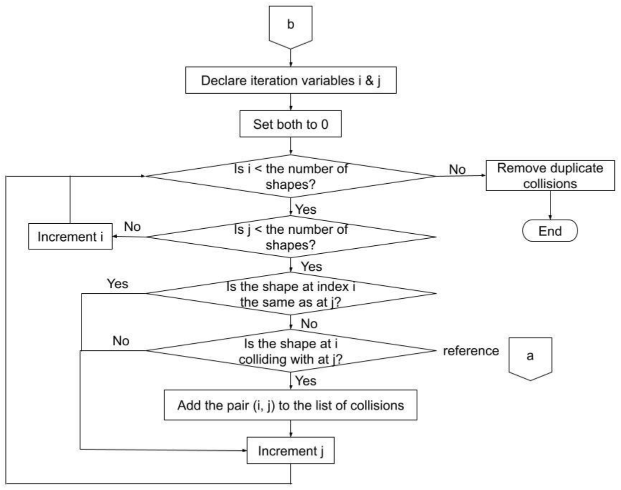
b
Declare iteration variables i & j
Set both to 0
Is i < the number of shapes?
No
Remove duplicate collisions
End
Yes
Increment i
No
Is j < the number of shapes?
Yes
Yes
Is the shape at index i the same as at j?
No
No
Is the shape at i colliding with at j?
reference
a
Yes
Add the pair (i, j) to the list of collisions
Increment j

Fig. 2. Flowchart for big-picture original algorithm

This is straightforward. For each shape in the array, for each shape in the array (again), if the shape from the outer loop is colliding with the shape from the inner loop, add the shapes to a list of colliding shapes. The one tricky part is that I do not want to check a shape against itself, because that would always show a collision.

This next part is where I implement the part of the algorithm given in the IEEE paper. I reference a function above that will tell us if two shapes are colliding, but I must define it.

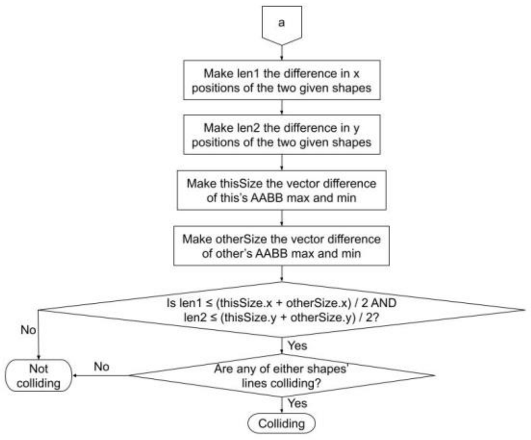
a
Make len1 the difference in x positions of the two given shapes
Make len2 the difference in y positions of the two given shapes
Make thisSize the vector difference of this's AABB max and min
Make otherSize the vector difference of other's AABB max and min
Is len1 ≤ (thisSize.x + otherSize.x) / 2 AND len2 ≤ (thisSize.y + otherSize.y) / 2?
No
Yes
Not colliding
No
Are any of either shapes' lines colliding?
Yes
Colliding

Fig. 3. Flowchart for AABB collision detection in the original algorithm

There are some obvious parallels between the algorithm in Fig. 3 and the algorithm outlined in [2]. I define Len1 to be the x distance between the shapes, and Len2 to be the y distance. Rather than using the notation found in (1) through (4), I use a 2D vector class I created to store the width and height of the AABBs. Then I use the exact Boolean expression laid out in the paper. If the expression is true, I run some code to check whether the shapes' lines are intersecting. If it is not true, I know the shapes are not colliding.

The line intersection detection code is outside the scope of this algorithm, but [3] was a great help for this.

The last part of the implementation of this algorithm is calculating the AABBs! I use them in the above flowchart, but they must be calculated. This is done in Polygon's update method. It is standard min/max finding code.

I simply iterate through the list of this Polygon's lines and keep track of the largest and smallest x and y values that have been seen so far. I then use those points to construct the AABB for the shape. This is done every frame, so the AABB is always a snug fit around the shape.

III. Improvement to the Initial Algorithm

A. Approach

The general goal of the improvement was to reduce the time complexity of the function from O(n^2) to O(n log n). When n = 100, calculating collisions with an O(n^2) algorithm made for 10,000 calculations. With an O(n log n) algorithm, this is reduced to just 200 calculations (assuming log base 10). This is just for illustrative purposes; these are not concrete values. As is apparent, O(n log n) is a massive improvement. More concretely, my hope was to be able to simulate collisions for 100 random shapes without a significant frame rate reduction (from 60 fps). This part of the evaluation includes drawing, moving, and handling collisions.

As mentioned above, the biggest slowdown in the original algorithm is the fact that every shape needs to be checked against every other shape. I can eliminate many of these checks using a common problem reduction in computer science: sorting.

First, I must change and organize the approach a little bit. I proposed that I think about detecting collisions between AABBs as an "interval" problem. The edges of shapes act as opening and closing intervals, and the goal is to check if any two intervals are open at the same time. Fig. 4 has an illustration of what I mean by this.

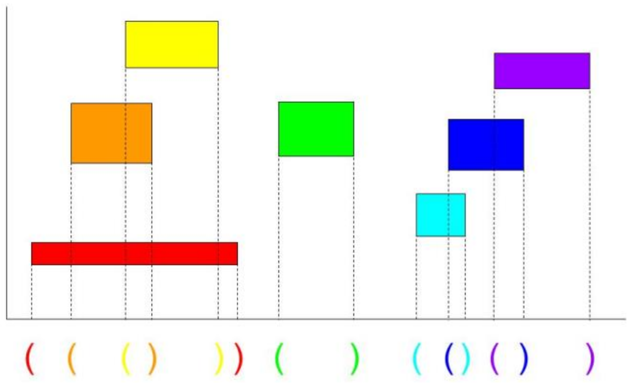

Fig. 4. An illustration of some shapes' x-axis intervals, represented as labelled parentheses

I am now presented with a different problem. The goal now is to determine which intervals overlap. In other words, which intervals become active while another interval is active. Before this is possible as in the illustration above, I need to do two things: get the intervals from the shapes and sort the intervals by their positions. To do this, I created an Interval struct that contains a position, a Boolean representing whether it is opening or closing an interval, and the index of the shape it refers to in the main array of shapes. It also overrides C++'s less than operator so that I can sort the intervals by their position.

This Interval keeps track of all the things that are kept track of in the figure above. Intervals are sorted by their position using std::sort, which is a highly optimized sorting algorithm with time complexity O(n log n). Intervals are easy to create if you have a polygon. Here is a code example:

```
Interval(&shapes[i].aabbMin.x, true, i)
```

This creates an interval with the minimum AABB x-value, which makes it the start of an interval, and it has index i, which is where the shape came from.

B. Implementation

The actual algorithm follows. The core of it happens in getPossibleCollided(), which takes a vector of collision intervals and the main array of shapes. It returns a set of CollisionPairs (2 Polygons that are colliding) filled with all the Polygons that are colliding with each other. The flowchart for the algorithm is in Fig. 5.

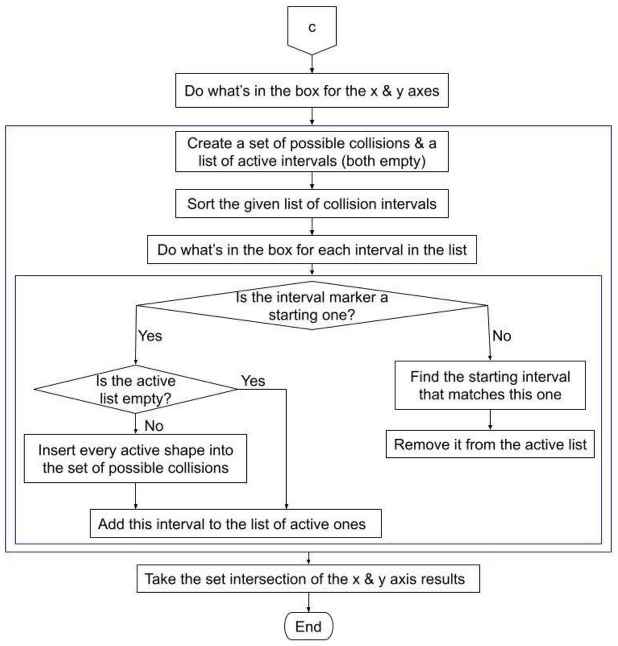
c
Do what's in the box for the x & y axes
Create a set of possible collisions & a list of active intervals (both empty)
Sort the given list of collision intervals
Do what's in the box for each interval in the list
Is the interval marker a starting one?
Yes
No
Is the active list empty?
Yes
No
Find the starting interval that matches this one
Remove it from the active list
Insert every active shape into the set of possible collisions
Add this interval to the list of active ones
Take the set intersection of the x & y axis results
End

Fig. 5. Flowchart for the improved algorithm

To rephrase briefly, I kept track of all the shapes I was currently "inside of" as I traversed the vector of intervals. If a new interval opened when there were active intervals, I knew there was a collision, so I added it to the set of CollisionPairs. When I encountered a closing interval marker, I deleted its opening pair from the list of active intervals. This whole process finds collisions on one axis.

However, I need to find collisions on both axes! If I get a set of the CollisionPairs on each axis, finding the set intersection of those two sets will give us the shapes colliding on both axes, which are exactly the ones I want. Success.

Aside: this part was tricky because I had to define a way to compare CollisionPairs. In the end I went with lexicographical comparison: comparing the first two, and if they were not the same, comparing the second two. This is like how alphabetical strings are compared. The cpCmp struct in the code defines a function to perform this operation.

I will go into more detail concerning time complexity in the results section below, as it shows clear success in this area. Inserting an item into a set, which could happen *n* times, happens in O(log *n*) time, so that makes O(*n* log *n*) time. Inserting an item onto my linked list implementation is always a constant time operation, so I do not have to worry about it. The real parts that seem problematic are finding and erasing an item from the linked list, which is O(*n*) time, and could theoretically happen *n* times. Although, I know that the thing I am trying to find will usually be close to the front of the list because that is where things are inserted, so it is not a big concern.

While sorting and interval problems are not new to computer science, I devised and implemented this solution entirely myself.

IV. Results

A. Methods of Analysis

I decided to use CPU time to evaluate the efficiency of the original algorithm and my improved algorithm. Time complexity of algorithms is usually considered the most important attribute of an algorithm in computer science, but particularly in graphics and game development. I wanted to use a respected form of evaluation. I decided against wall time simply because it seemed less consistent and accurate for measurement. I did not want the measurement to rely on other aspects of the DE1-SoC like any other processes it may be running. There are only a couple of ways to measure CPU time in C++ in Linux. I chose <ctime> for its simplicity.

I only measured the time it took the algorithms to generate all the collision pairs. I did not include drawing, handling, or updating time. This is partially because it was likely to be the same for both algorithms, and partially because I was only evaluating the collision detection algorithms themselves, not the other algorithms.

For the measurements, I decided to vary the number of inserted shapes (at 1, 10, 100, 200), as well as which shape was being drawn. I compared the square shape to the 'X' shape for both algorithms. The square shape had 4 lines and the 'X' shape had 12. Note that I varied the number of inserted shapes. In my implementation, the edges of the screen were also Polygons that had to be processed, so inserting 1 shape makes 5 shapes total.

B. Data

As shown in Table 1 and Figs. 5 through 8, my improved algorithm is faster than the original in every case! Additionally, with 100 shapes, the collision calculation time, which is the most expensive step in the broader program, takes less than 3 milliseconds, which is a small fraction of 16.67 milliseconds (the time one frame takes at 60 frames per second). Therefore, there is no significant slowdown at 60 fps with 100 shapes, and the goal has been met.

TABLE I. Timing Results

Shape	**Inserted Shapes**	**Average Calculation Time Per Tick (ms)**		**Speed Improvement Factor**
		Original	***Improved***	
Square	1	0.133668	0.0506073	2.6
	10	1.57945	0.110429	14.3
	100	90.6606	2.97351	30.5
	200	195.187	9.15208	21.3
X	1	0.241546	0.0365497	6.6
	10	12.4563	0.202086	61.7
	100	577.627	2.44561	236.2
	200	1336.33	11.396	117.3

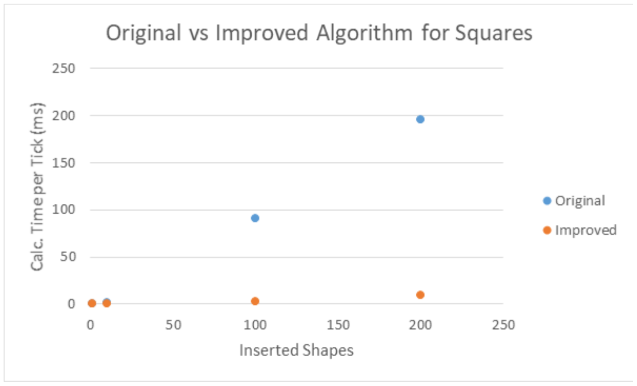

Fig. 5 Plot of original vs improved algorithm for squares

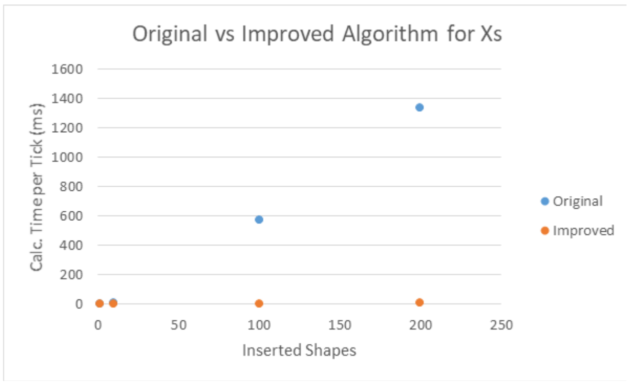

Fig. 6 Plot of original vs improved algorithms for ‘X’ shapes

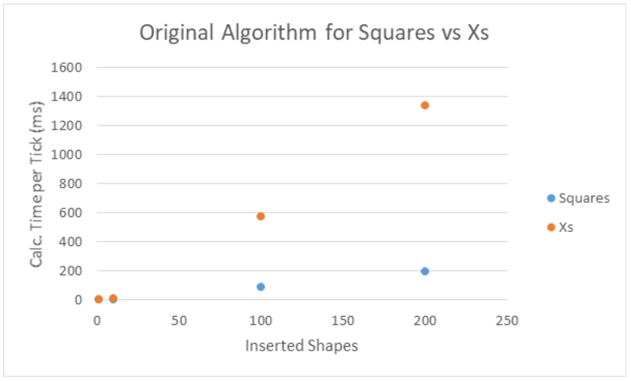

Fig. 7 Plot of original algorithm for squares vs ‘X’ shapes

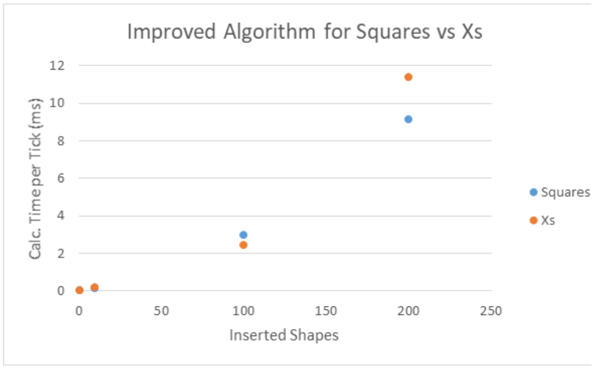

Fig. 8 Plot of improved algorithm for squares vs ‘X’ shapes

As you can see, there is a very large difference between my improved algorithm and the original algorithm. It is very noticeable for higher values of inserted shapes. I am particularly proud of the fact that my algorithm is over 200 times faster than the original in the case of 100 ‘X’ shapes. The plot also clearly shows that the original algorithm’s time complexity is growing faster than my improved algorithm, which is really the key.

One positive thing that I was not expecting was that my algorithm reduced the difference between computation times for different shapes, as showing in Figs. 7 and 8. The original algorithm has a very large difference between the computation times of the ‘X’ shape and the square shape – a difference that the improved algorithm does not have.

V. Conclusions

I have successfully and drastically improved the algorithm found in [2]. As far as the overall project goes, this algorithm is sufficient to detect collisions between any reasonable number of polygons that would fit on the screen at the same time without any lag, so it is declared successful. However, there is still room for improvement.

One large inefficiency that would be easy to cut out is calculating the AABBs every frame. This has a time complexity of O($n * m$), where n is the number of shapes, and m is the number of lines in each shape (because all of the lines have to be iterated over to find the min and max for each shape). An alternative would be to assign an AABB to a shape upon creation, large enough that no rotation could make the shape extend outside of the AABB. Then, the AABB could just be updated by the velocity of the Polygon every tick, rather than being recalculated with all the lines.

Another thing that might be a slowdown is the use of a linked list in the improved algorithm. Constant time insertion is guaranteed, which is very useful, but the linear time it takes to get an element in the list is not great. An alternative solution would be to declare a vector of some estimated size, and insert and get from that, and both operations would probably be constant time, even though the vector might take more time to initialize. This would be a good thing to experiment with.

Lastly, note that the improved algorithm could be easily extended to work in n dimensions. All that is needed is an AABB of n dimensions. Each of the n axes could be checked in the same way, and the set intersection would just involve more sets.

Acknowledgment

I would like to acknowledge Dr. J Marpaung for his assistance with my understanding of memory mapping in C++.

References

[1] B. V. Mirtich, “Impulse-Based Dynamic Simulation of Rigid Body System,” dissertation, University of California at Berkeley, Berkeley, CA, 1996.

[2] Y. Tan, C. Liu and Y. Yu, "The Design of Collision Detection Algorithm in 2D Grapple Games," 2009 International Conference on Information Engineering and Computer Science, Wuhan, China, 2009, pp. 1-4, doi: 10.1109/ICIECS.2009.5363537.

[3] D. Gouveia, “How to detect 2D line on line collision?,” *Game Development Stack Exchange*, 2012. [Online]. Available: https://gamedev.stackexchange.com/questions/26004/how-to-detect-2d-line-on-line-collision. [Accessed: 15-Apr-2021].

[4] “Bresenham's Line Generation Algorithm,” *GeeksforGeeks*, 22-Mar-2021. [Online]. Available: https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/. [Accessed: 15-Apr-2021].