https://github.com/cs540-25/Methods-in-Collision-Detection.git

Methods in 2D Collision Detection

Benjamin Belandres Alexander Merkle

Department of Electrical Engineering and Computer Science

University of Tennessee, Knoxville

bbelandr@vols.utk.edu amerkle@utk.edu

Abstract—

INTRODUCTION: While a wide variety of collision detection methods have been devised and implemented, there is a need and demand for understanding the scenarios in which each of these methods can be used to their fullest extent.

OBJECTIVES: Our goal is to quantitatively measure performance metrics with a number of collision detection methods in order to determine strengths and weaknesses for each. By highlighting this information, we hope to provide the reader with the knowledge necessary to make informed decisions regarding which of our tested methods is best suited for their purposes.

METHODS: We implemented Brute Force Center-Radius Axis-Aligned Bounding Boxes (AABB), Brute Force Bounding Circles, Sweep and Prune AABB, Variance Based Sweep and Prune AABB, and Uniform Grid AABB as Arrays of Linked Lists.

RESULTS: We found that our implementation of Variance Based Sweep and Prune performed the best out of all of the methods that we chose, but perhaps if our implementation of Uniform Grids changed we would have seen a more competitive performance from it.

CONCLUSION: Based on our results, Variance Based AABB is the most efficient form of collision detection out of the 5 in our study.

I. Introduction

Many different kinds of engines use collision detection in order to help simulate physics. Such a varied range of usage leads to a wide quantity of versions that can be used to implement collision detection. This project was born out of interest for the construction of game engines and the mechanics that come together to make them work. Our primary objectives were to create a control environment for these versions to be tested and appropriately compared. Ultimately, these comparisons and measurements can be used to identify scenarios in which each method can be justifiably used. This should allow for you, the reader, to gain an increased understanding of each version so that you may identify which collision detection method would be best used given your needs and resources.

This work built upon a great deal of research conducted by Christoph Ericson in *Real Time Collision Detection* [1]. Different implementations are listed and explained with added commentary on their performance. There is a gap in existing research concerning the identification and contextualization of these widely used versions. This paper will expand upon Ericson's by contrasting these detection methods.

II. Objectives

This research is primarily intended for a practitioner in the field to use. It will be particularly useful when trying to decide which form of collision detection is the right kind to use for a given scenario. Hopefully, through its publication, the research will help improve the efficiency of individualized services created by people who employ collision by allowing them to make informed decisions.

With this paper, we sought to examine the functionality and efficacy of a number of widely used collision detection methods before identifying their strengths and weaknesses compared to one another. With this objective in mind, we intended to provide informed answers to the following questions:

- *What effects do the implementations have upon the performance of running the physics simulation?* Each implementation's efficiency can be measured in terms of performance indicators such as framerate, memory allocation, and CPU usage.
- *How do these metrics compare to one another, and what conclusions can be drawn regarding each implementation's feasibility in usage?* We intend to identify the strengths, weaknesses, and limitations of each method tested.
- *Under what circumstances can each of the methods be used in a manner beneficial to the user?* Different scenarios will allow each method to gain or lose utility.

III. Methods

In order to collect the information needed, the team built a physics engine capable of switching between and analyzing multiple collision detection methods. Those methods are as follows:

- Brute Force Center-Radius Axis-Aligned Bounding Boxes (AABB)
- Brute Force Bounding Circles
- Sweep and Prune AABB
- Variance Based Sweep and Prune AABB
- Uniform Grid AABB as Arrays of Linked Lists

The engine had to be able to render multiple physics-enabled objects at the same time. For that reason, the engine was written in C++ along with the Simple DirectMedia Layer library (SDL) to allow the use of a window for rendering. Everything else was written completely from the ground up over the course of a semester.

A. Measurements and Performance

To produce metrics, the process's performance was measured through a ratio of its total frames and the amount of objects being rendered. Each collision method was run multiple times for a total of 10 seconds. Metrics were made of each method's average performance so that they can be easily compared. We tested each collision detection method with 1000 objects.

B. Reference Points

The collision methods will be designed following the guidelines according to Christer Ericson's *Real Time Collision Detection* [1]. Once an implementation is complete, a code review will be conducted with Ericson's examples to ensure that everything was written correctly.

1) Brute Force Center-Radius Axis-Aligned Bounding Boxes (AABB): This method of collision detection uses an AABB system to determine when two objects are colliding. The center-radius implementation uses a centerpoint of the bounding area along with the radius, or half of the length, of each side. This implementation uses a radius to make the required data size to store the AABB smaller; a normal length and width would take a larger data structure, like a uint_32, to hold the information while a radius only needs half of the size, making it fit inside of a uint_16.

This method is titled "Brute Force" because we are exhaustively checking every object against each other to find if they are colliding or not; there is no sorting or clever behavior to minimize computational costs.

2) Brute Force Bounding Circles: Bounding circles are the most efficient way to store a collider, needing only a center-point and radius (which actually means the radius of a circle, unlike center-radius AABB). The structure of a bounding circle makes its memory footprint the least taxing out of all of the collision methods in this project.

This method also uses the Brute Force method, meaning that we are simply checking every object against every other object for collisions.

3) Sweep and Prune AABB: Sweep and prune AABB uses the AABB data structure and finds a clever way to avoid having to Brute Force through every object in the engine to find collisions.

In a nutshell, sweep and prune AABB casts a projection of every object onto either the x or y axis. If the projections of two objects overlap, then the program checks if there is a collision between the two. As a result, this method cuts down quite a lot of processing.

4) Variance Based Sweep and Prune AABB: This method takes a slight modification to the standard sweep and prune method; it changes the axis that it chooses to project the objects onto. The algorithm determines which axis to project onto based on the variance, or the maximum value minus the minimum value, of the objects in each axis on the previous frame. The axis with the largest variance gets chosen for the next frame's projections.

5) Uniform Grid AABB as Arrays of Linked Lists: We chose the simplest method conceptually for implementing uniform grids. Christer Ericson called this method "Grids as Arrays of Linked Lists" [1]. There are many other implementations of uniform grids that improve upon the efficiency of the implementation, but due to time constraints, we chose to create the easiest algorithm.

This method models a uniform grid quite directly through the use of a 2D array that comprises of doubly-linked lists. The linked-lists point to objects that happen to exist in the same cell of the uniform grid. Those objects are then checked for collisions against one another.

This implementation comes with an important question: *How large should a cell of the uniform grid be?* Having a grid that is too large makes uniform grids function very similarly to the Brute Force computations, and making the cell size too small increases the memory footprint dramatically. Following Ericson's guidelines, we chose to make the average cell the size of an object [1]. This helps avoid making the grid too large or too small.

IV. RESULTS

Variance Based Sweep and Prune seems to have been the best implementation out of all of the methods in this paper. This likely has happened not only because Variance Based Sweep and Prune effectively weeds out extraneous computations, but also because our implementation for uniform grids was made less complex and therefore less efficient.

While we were running the simulations, it was worth noting that when objects clustered into a certain area of the screen, the methods that tried to weed out extraneous calculations suffered performance wise while the Brute Force methods stayed consistent. This can explain why the variability of the Brute Force methods are the lowest of the five methods. The Brute Force method consistently operates under the worst-case scenario of the more efficient methods and therefore is not subject to more dramatic swings in framerate.

Additionally, our program originally had an issue where some objects could get pushed outside of the bounds of the screen. This was likely because the program didn't have continuous collision detection for the edges of the screen, allowing for a phenomenon known as tunneling. It was interesting to note that as the simulation ran, the uniform grid method performed better over time because the objects that fell out of the screen no longer had to be considered for collisions. While we could not record this information directly, the memory performance of each method can be intuited. The Brute Force

RESULTS					
Collision Method	Average framerate	Max framerate	Min framerate	Framerate variability	Frames per Object
AABB Brute Force	41.91	44.06	35.32	8.75	0.39
Circle Brute Force	34.19	37.92	29.17	8.75	0.34
Sweep and Prune AABB	72.79	77.78	66.80	10.98	0.73
Variance Based Sweep and Prune AABB	90.48	94.55	82.71	11.84	0.90
Uniform Grid AABB as Arrays of Linked Lists	77.03	81.89	67.75	14.14	0.77

methods took up the least amount of memory since they had no additional data structures associated with them except for the bounding volumes themselves. The Sweep and Prune methods used slightly more memory since little to no new data structures were made for their implementations, and Uniform Grids were the most expensive on memory because that method had a whole 2D array, multiple members, and multiple functions dedicated to it.

A. Future Work

Everything in this project was single-threaded. It would be interesting to see if the amount of threads actually hinders or helps the performance of a certain method of collision detection.

There are many other scenarios in which one could have to detect collision. It would be interesting to see more scenarios tested.

There are a plethora of other collision detection methods that are not mentioned in this project. Seeing more analysis of the other collision detection methods would be very interesting.

V. CONCLUSION

The Brute Force method of implementation for each of our tested methods required the least amount of memory usage, but skyrocketed in total CPU usage as compared to other methods. Given its simple implementation, its variations would naturally serve as good preliminary methods when focusing one's trials on ensuring the functionality of other systems within a project. It is also ideal for projects where one's central goal might be to minimize the amount of allocated memory. However, its usefulness rapidly wears out as the scope of any project expands. Even when conducting experiments that purely focus upon these methods' viability and do not incorporate processes that are not directly contributing to collision detection, the maximum number of objects that could be tested without significant shortcomings in latency was comparably limited.

Variance Based Sweep and Prune AABB tended to be the most efficient in terms of CPU usage and framerate. This method would likely serve as an excellent solution for projects built with less restrictions on resources. Normal Sweep and Prune AABB, meanwhile, tended to be easier on memory, making them ideal for middle ground scenarios where resources are apparent but not abundant.

VI. REFERENCES

[1] C. Ericson, *Real Time Collision Detection*. San Francisco: Elsevier Inc., 2005. Accessed March 1st, 2025. [Online]. Available: https://www.r-5.org/files/books/computers/algo-list/realtime-3d/Christer_Ericson-Real-Time_Collision_Detection-EN.pdf