Adaptive Broad-Phase Collision Detection

Henry Owen
COS 497, University of Maine
4/19/18
Advisors: James Fastook, Phillip Dickens

Abstract

Choosing an appropriate broad-phase collision detection algorithm often depends on the environment in which the collisions are taking place. We conducted an in-depth experiment to understand whether object-speed and object-size variance have a role in deciding which algorithm is appropriate.

This experiment did not support that object-size variance is a significant factor in this decision. However, the experiment did suggest that object-speed is a factor given a fixed environment size. Sweep and Prune, Spatial Masking, and Oct-Tree performance was not affected by object size-variance, but was affected by object speed. The effect of speed on the algorithm's performance could be reversed by changing the environment size or the object-size. Finally, the average velocity of objects in a scene can be used to switch to the optimal broad-phase collision detection algorithm in real-time given a fixed environment size, object quantity and average object-size.

Introduction

It is important to many video games to run as fast as possible to give the user a high frame-rate. At lower frame rates the user begins to notice the lack of visual feedback. If a user is playing a game with fast moving objects, a higher frame rate will provide more detail as to what the objects are doing. For instance, in a Billiards game when a player strikes a ball, it might go off in a direction with considerable speed. With each new frame, the ball would appear in a new position along it's trajectory. A high frame rate is able to show smooth changes in the ball's position, while a slow frame rate would cause visual jumps from position to position. Of course, every frame, the game is checking whether the ball is striking another ball, a wall, or a pocket. A ball might collide with any number of entities present on the pool table.

This process of checking for collisions can be a major factor in the game's frame-rate, and reducing collision checks can improve performance. If there are *n* objects in a scene, the collision detection process must process n^2 possible collisions. A ball on the pool table must react to contact with each wall, each pocket, and possible a pool cue. In turn, if a cue-struck ball hits a stationary ball, that ball might roll off in a different direction and make collisions of it's own. Making this many checks can be taxing and unnecessary. Performance can be enhanced by reducing the number of collision checks.

Efficient collision detection is thus split into two phases. First reducing the number of collision checks, then measuring exact distances between the remaining pairs. During the first phase, if a ball is on one side of the pool table, and another ball is stationary on the other side, it is unlikely that they are colliding during that frame and so the collision detection process skips this pair. This is called the broad-phase because it makes broad calculations such as which side of the pool table the balls are on. The phase that measures the exact distance between two objects is called the narrow-phase.

Many broad-phase algorithms exist for collision detection with diverse methods for reducing collision pairs. Common ones include Spatial Masking, Oct-Trees, and Sweep and Prune algorithms.

Spatial masking uses a uniform grid to determine “which side of the table” the objects are on. If two objects inhabit the same cell in that grid, they are determined likely to collide. It does this by generating a bitmask whose bits represent a single cell. Each axis gets a bit mask as a cell's location in a 3D grid is indexed by three values. Now that the objects have bitmasks, whether or not the objects inhabit the same cell can be determined quickly with bitwise operations. This is O(n^2) in all cases.

Oct-trees use a non-uniform grid to partition the scene[5]. Again if two object inhabit the same cell, they are deemed likely to collide. Oct-Trees recursively partition the space into octants until either the cell only has one object or the cell reaches a minimum size. An Oct-Tree node is a bounding volume, and it is either a leaf node or has eight child Oct-Trees. If an object is entirely bounded by one of the Oct-Tree's children, then it is sent down the tree. If the object overlaps between two octants, the object stays in the parent node. The Oct-Tree creates collision pairs by first coupling its own objects, then sends those objects down to its children. If only leaf nodes contain objects, then finding collision pairs is O(nlogn).

Sweep and Prune algorithms keep a record of the order in which objects appear on each axis[4]. On the *x*-axis, an object appears first if it has the minimum *x* end point. Objects are sorted according to their end-points using insertion sort. These end-points determine which objects are entered into a table of collision pairs which is updated each frame. Sorting end-points can be *O(n)* if objects do not move drastically between frames, but must also traverse a 2D array to retrieve collision pairs, which runs in *O*(n^2).

The problem with these diverse broad-phase methods is that the nature of the scene can affect their performance in different ways. Oct-Trees store objects in a tree data structure according to their location. If the objects move too much, they have to be reinserted into the tree, a *O(nlogn)* procedure. The Sweep and Prune algorithm uses insertion sort to take advantage of nearly-sorted lists of end points, but insertion sort can be *O*(n^2) if too many drastic changes happen between sortings. Finally, the Spatial Masking algorithm creates bitmasks for each object every frame [3], which can be unnecessary overhead when there are few possible collisions in the scene.

Spatial Masking algorithms use a uniform grid which is optimal when the cells in the grid are about the same size as the objects in the scene. Grids with cells that are too small means that *j* bits have to be set in each object's bitmask where *j* is the number of cells that an object is in. Cell sizes that are too big could cause the extreme case in which all objects inhabit the same cell, causing *O*(n^2) complexity. Thus cell sizes are determined by the average size of the objects in the scene. Too much variation in object size and it could cause degradation in performance.

It is well known that broad-phase algorithms have strengths and weaknesses. For instance, Nvidia's documentation of PhysX's broad-phase algorithms states that their Sweep and Prune algorithm's performance degrades when objects are moving, or when many objects are being inserted and reinserted[7]. However, a single algorithm is usually selected to best suite the application.

It is not well-known whether the conditions in a scene can be used to determine which of these algorithms to use in real-time. We conducted an experiment which tested whether average object-velocity and object size-variance could be used to determine which algorithm to use in real time.

We hypothesized that object size-variance and object-velocity could be used to determine the optimal broad-phase algorithm, and that calculating this information and switching between algorithms would yield faster broad-phase performance.

The results suggest that object size-variance is not a significant factor in broad-phase algorithm performance, but speed was a factor given fixed environment size, object-size, and object quantity. This information was used to determine the speed at which each algorithm performed best given these fixed conditions. Calculating the average velocity in real time and switching between algorithms resulted in faster broad-phase performance.

Related Work

Wolfe and Manzke created a framework for benchmarking broad-phase collision detection algorithms and tested some algorithms [1]. They implemented a testing environment using OpenGL, enumerated testing conditions, and tested Sweep and Prune and AABB Tree algorithms [1]. There were a number of conditions tested for including the geometric aspect ratio, spatial distribution and quantity of objects. This series of experiments found that the Bullet Physics DBVT broad-phase algorithm performed best best regardless of object quantity or object aspect ratio. This suggested that these parameters did not significantly affect the performance of the algorithms [1]. This series of experiments is similar to those in this paper. This experiment seeks to test the algorithms in a number of different conditions. However, their goal was not to attempt to combine algorithms based on this information. The information that they collected from their experiment was used to demonstrate the value of a well designed benchmarking environment. Their experiment suggested that the Bullet Physics Sweep and Prune algorithm is not as efficient as the DBVT algorithm, a result that contradicted conventional wisdom [1]. They also did not test a Spatial Mapping or an Oct-Tree algorithm.

Rene Weller et al. created a suite for benchmarking algorithms in a similar way to Wolfe and Manzke but concentrated on both broad and narrow phase collision detection[2]. They were interested in comparing the performance of both broad and narrow-phase based on various object geometries, a factor that becomes important during narrow-phase. In the experiment presented below, the algorithms use the same narrow phase collision detection algorithm which serves to better compare the broad-phase algorithms alone.

Materials and Methods

We used Unity to implement a virtual scene for testing the algorithms. Unity has support for creating simple objects like spheres and planes, and provides a 3D vector API, which we used for implementing particle physics.

Although Unity has a native physics engine, we opted to use our own particle physics engine for simpler integration with custom collision detection algorithms. This particle physics engine allows collisions between spheres and planes. For our experiment we only need spheres bounded by a cube.

Particle physics serves as a comprehensive subject for broad-phase algorithm testing as the broad-phase is identical to any other scheme that uses spherical object approximations, also known as spherical bounding volumes, a common scheme in collision detection applications[6].

For our experiments we distributed the spheres randomly inside the bounding cube. This is because it is possible for the Oct-Tree to perform badly when uniformly aligned objects align with octant boundaries.

We also ensured that random object clustering was eliminated. Objects were given velocity in random directions, there was no fluid drag on the objects, and the collisions in the tests were perfectly elastic to keep the objects bouncing off each other indefinitely. This creates a scene in which objects diffuse evenly in the scene. This elimination of object clustering isolates object velocity and object-size variance as factors to a greater degree.

The average random velocity was calculated using a simple rolling average. At each time step of the simulation, every time an object's velocity was updated, the object's old velocity was factored out of the average and the new velocity was factored in. Given that there is no loss of energy in the simulation thanks to there being no fluid drag and perfectly elastic collisions, the average velocity is constant throughout the test, but the calculation was needed in order to determine whether the extra calculations needed to calculate the average nullified the benefits of using them to determine the optimal algorithm.

The first experiment was a baseline benchmark which tested the algorithms against several object quantities. The objects were placed in inside a 1024^3 m^3 bounding cube, given a radius of 0.5 m and a velocity of 0.173 m/s. The object quantities ranged from one to three thousand.

To measured the effect of object-velocity on algorithm performance, we ran a test in which the bounding volume was 64^3 m^3, with 339 objects with 0.5 m radii. Then the experiment measured speeds from 0 to 415.9 m/s, while calculating the execution times of the algorithms.

We ran an experiment which measured the effect of environment-size on algorithm performance. The quantity, size, and speed of the objects were fixed at 1000, 0.5 m, and 43.3 m/s respectively. The environment sizes measured were from 32^3 m^3 to 2048^3 m^3.

To test size variance, we designed a test to determine whether object size-variance alone is a factor in performance. The algorithms are tested against three different size distributions, normal, inverse-normal, and uniform, as well as control runs with no size-variance. See the appendix for diagrams of the distributions used in this test.

To eliminate the effect of speed and environment size seen in the previous tests, and to isolate the size-variance factor, environment-size and object-speed were adjusted in proportion to the average object-size. Environment size was adjusted so that the objects took up 0.01% of the bounded volume, and the speed was adjusted so that a particles would take 25.6 seconds to traverse the length of the bounding area. The object quantity was fixed at 500. In each distribution the minimum radius size was 0.001 m and the maximum was 80 m.

Since every distribution had different environment size, average object-size and object-velocity, we ran control tests for each distribution. The control tests use the same average radius size (but with no size-variation) as their respective distribution in order to duplicate the environment size and speed of the last test.

The final test incorporated extreme conditions in which each algorithm would be able to perform the best at different speeds. The bounding area was 32768^3 m^3, and contained 2000 objects with 10 m radii. Speeds ranged from 0.1 to 119110.8 m/s.

Results

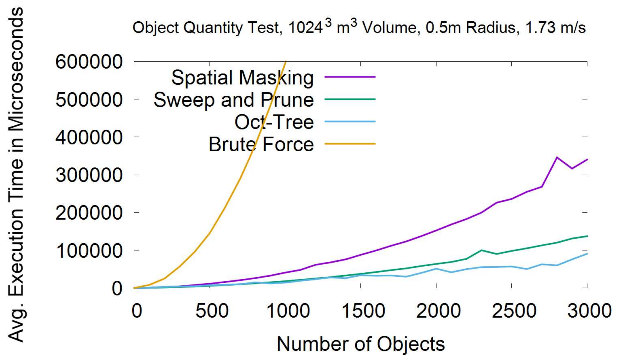

Fig 1: Baseline test measures execution time against object quantity. Y-axis is execution time in microseconds, the X-axis is the number of objects in the scene.

In Fig. 1, the algorithm marked "Brute Force" is a control *$O(n^2)$* algorithm. The brute force algorithm took 600,000 microseconds to execute at one-thousand objects, while the others had an upper bound of 350,000 microseconds at three-thousand objects. The Oct-Tree performed the best as quantities became larger. This was because with three-thousand objects the result was a sparsely populated scene with small objects compared to environment size. This particular set of conditions allowed the Oct-Tree to perform well.

As described in the above section, the objects were moving at 0.173 m/s with a radius of 0.5 m. This is relatively slow speed and small radius for the size of the bounding volume. This creates a scene in which the objects are sparse. Since an Oct-Tree recursively partitions the scene until only one object inhabits a partition or it reaches a minimum partition size, the Oct-Tree create large partitions containing a single object. Since the partitions are large, the object can move further without having to be reinserted into the tree. Since the objects are evenly distributed in the scene, the tree is likely to have one object per partition. This creates a near-optimal case for the Oct-Tree in which upkeep of the tree is inexpensive, and collecting collision pairs is *$O(nlogn)$*.

The Sweep and Prune algorithm's execution time grows faster than the Oct-Tree due to its traversal of a 2D array to retrieve collision pairs. However, it still outperforms the Spatial Masking Algorithm.

The Spatial Masking execution time grows significantly faster than the other two algorithms. This is a result of the extra overhead involved with creating bitmasks. This is *O(n^2)*, but is significantly faster than the brute-force method at these object-quantities.

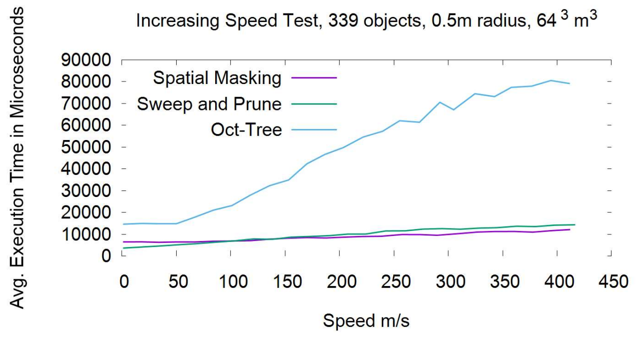

Fig 2: The effect of object speed on algorithm performance. The Y-axis is execution time in microseconds, the X-axis is the speed in m/s.

The next experiment tests the effect of increasing object speeds. As can be seen from Fig. 2, the Oct-Tree is affected the most by speed, with Sweep and Prune being more resilient and Spatial Masking the most resilient.

The Oct-Tree approaches worst case when object speeds are high. Although the Sweep and Prune algorithm is affected by object speed it handles it better than the Oct-Tree algorithm. The Spatial Masking Algorithm is not significantly affected by object speed.

The Oct-Tree algorithm is significantly affected by object speed because a moving object is removed from and placed back into the tree structure when the object's location changes significantly.

The effect of fast moving objects on Sweep and Prune ability to sort end-points is visible. Under these conditions, Sweep and Prune is still practical.

The Spatial Masking algorithm is not affected by object speed because bitmasks are remade every frame regardless of whether or not an object's location has changed. This process is O(n^2) no matter what.

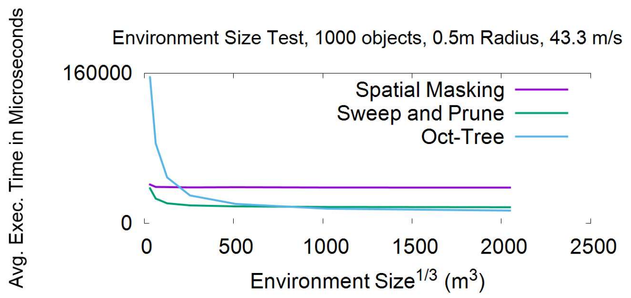

Fig 3: Measured the effect of increasing the bounding volume while keeping speed, object size and quantity constant. The Y-Axis is the average execution time in Microseconds, the X-Axis is the cube root of the volume of the bounding cube in m^3.

Figure 7 shows that increasing the size has the opposite effect of increasing speed. As the environment gets larger, the better the Oct-Tree and Sweep and Prune algorithms perform, while Spatial Masking again remains unaffected. This refutes that hypothesis that object velocity alone would be able to predict which algorithm to use.

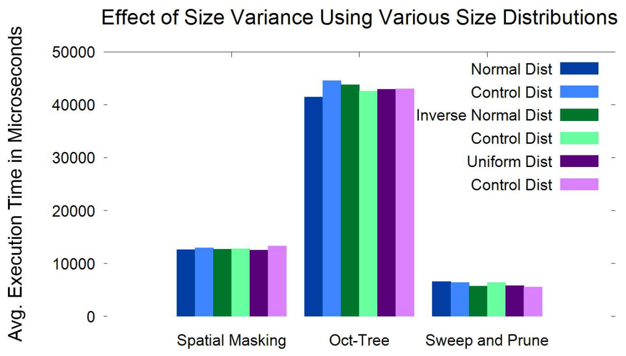

Fig 4: Measures the effect of object size-variance on the algorithms with three different size distributions. The Y-axis is execution time in microseconds.

Figure 4 shows that there is little difference in performance between normal, inverse-normal, and uniform distributions of object-size for each algorithm. Despite some differences in performance across distributions, size-variation does not appear to be a critical factor for algorithm performance. This refutes the hypothesis that object size-variance would be able to help determine the optimal algorithm. In this range of variation, the Spatial Masking algorithm did not appear to suffer from variation as was expected.

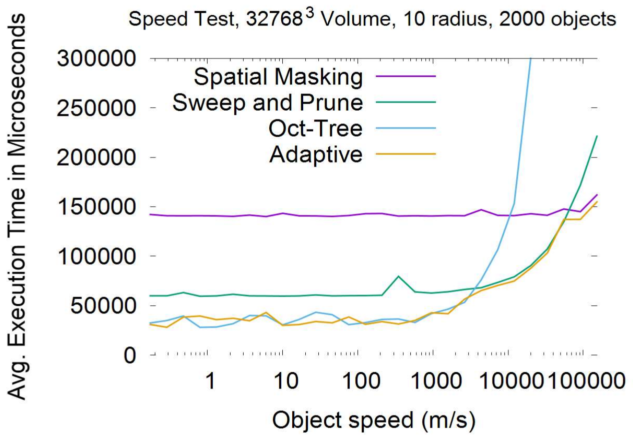

Fig 5: Measured the affect of speed on algorithm performance. The Y-axis is execution time in microseconds, the X-axis is object-speed in m/s.

The experiment in figure 5 was designed to include conditions in which each algorithm would be able to outperform the others. At ~3100 m/s, the Sweep and Prune began outperforming the Oct-tree. At ~74000 m/s the Spatial Masking algorithm began outperforming the Sweep and Prune. The Adaptive algorithm uses these values to switch to the optimal algorithm at these speed thresholds. The use of an adaptive algorithm based on object-velocity supports the hypothesis, but only given a fixed object quantity, environment size, object size.

The scene is very large compared to the size of the objects, this allows the Oct-Tree to perform well, but means that the experiment had to increase to very high speeds before the other algorithms outperformed the Oct-Tree. Combined with Fig. 2, where the environment was smaller and speeds slower, this suggests that the relationship between environment size, object size, and object speed has a much more powerful bearing on algorithm performance than object-velocity alone.

Although switching to the Oct-Tree and Sweep and Prune algorithms requires extra overhead during initialization, the cost is not prohibitive. Oct-Trees have to build the tree from scratch on the first frame, and Sweep and Prune algorithms have to sort a totally unsorted list of end-points. For the Oct-Tree, building a tree is not unlike reinserting all the objects. However, when reinserting, the object travels up the tree until it reaches the first appropriate octant, then it travels down to find the minimum bounding octant. When building, an object only has to travel down the tree. Thus the average case for updating an existing tree is $log_8 n$ operations, while the average case for inserting an object into a new tree is $log_8 n * (½)$ operations. For Sweep and Prune, the first frame uses C#'s List<T>.Sort method.

Conclusion

The tests support the hypothesis that object speed can be used to get better performance out of the algorithms. However, this in only when the speed reaches a certain threshold given a fixed volume and object quantity. For this reason, it appears other factors such as environment size and average object size would have to be considered to determine which algorithm is optimal.

The tests do not support that object size variance can be used to get better performance out of the algorithms. Object size-variance does not appear to be a critical factor in algorithm performance and therefore other factors should be considered first.

Future Work

To reform the hypothesis, future testing would involve testing whether the percentage of space taken up by objects can determine the speed at which the algorithms have equal performance.

For addressing the limitations of the algorithms when scaled to large environments, we would like to implement Multi-Sweep and Prune and Multi-Spatial Masking. These are schemes for splitting up an environment into multiple areas each using their own localized Sweep and Prune or Spatial Masking scheme. This addresses issues such as when an environment is too wide for a 64-bit integer to encode and objects location specifically enough.

Works Cited

[1] Wolfe, Muiris, and Michael Manzke. "A Framework for Benchmarking Interactive Collision Detection." *ACM Digital Library*, ACM, 25 Apr. 2009, dl-acm-org.prxy4.ursus.maine.edu/citation.cfm?id=1980501.

[2] Weller, Rene, et al. "A Benchmarking Suite for Static Collision Detection Algorithms."

[3] Bruce, J. "Real-Time Motion Planning and Safe Navigation in Dynamic Multi-Robot Environments Ch. 3: Collision Detection." *Http://www.cs.cmu.edu/*, Carnegie Mellon University, 15 Dec. 2006, www.cs.cmu.edu/~jbruce/thesis/chapters/thesis-ch03.pdf.

[4] Terdiman , Pierre – September 11, 20. "Sweep-and-Prune." Sweep-andPrune, www.codercorner.com/SAP.pdf.

[5] Nevala, Eric. "Introduction to Octrees." GameDev.net, 19 July 2017, www.gamedev.net/articles/programming/general-and-gameplay-programming/introduction-to-octrees-r3529/.

[6] Larsson, Thomas, and Linus Källberg. "Fast Computation of Tight-Fitting Oriented Bounding Boxes." Game Engine Gems 2, 2011, pp. 3–19., doi:10.1201/b11333-3

[7] "Rigid Body Collision." *NVIDIA Developer Documentation*, docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/RigidBodyCollision.html.