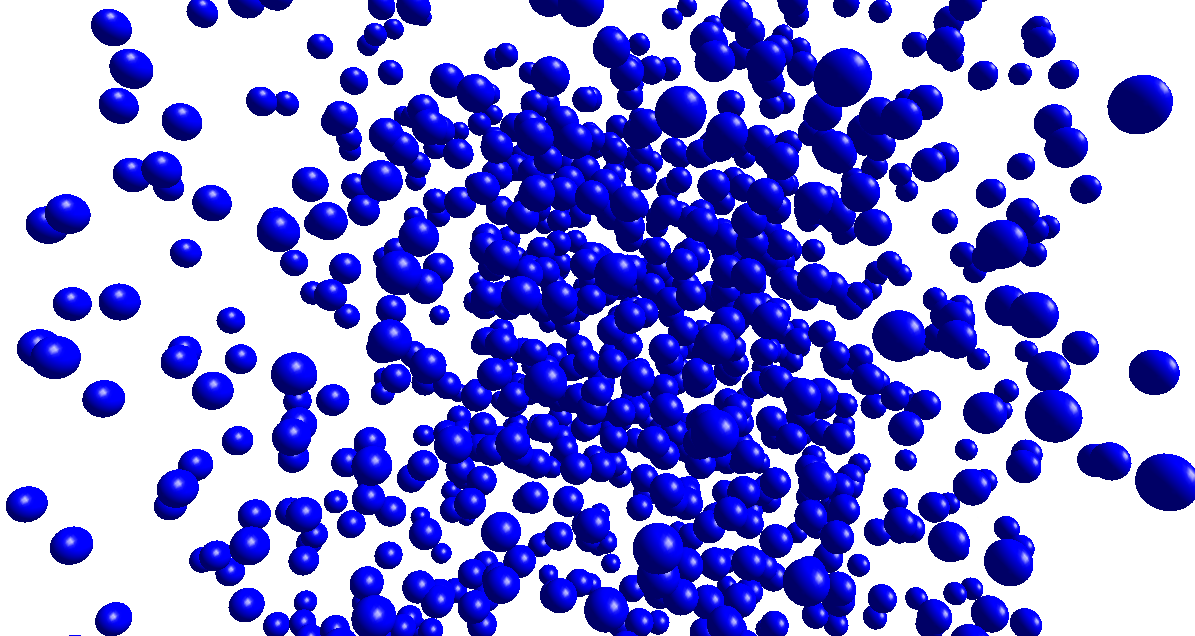
Discrete Collision Algorithms

Justin Bass James Gomez Utkarsh Jaiswal

University of California, Los Angeles



**Figure 1: Brute Force Collision** *for 1024 objects, initially directed towards the origin of the scene in order to maximize collisions. Although the algorithm used in inconsequential for the scene dynamics, using brute-force for this many objects runs at less than one frame per second - more efficient methods are needed for real-time simulations.*

# **Abstract**

In this paper, we explore various collision algorithms and their performance when used in scenes with varying numbers of objects. We explore, in particular, three discrete collision algorithms and their corresponding data structures: Octrees, KD trees, and Sweep and Prune. Each of these uses a Bounding Volume Hierarchy (BVH) data structure, and each BVH uses Axis-Aligned Bounding Boxes (AABBs).

# **1 Introduction**

Collision detection methods are widely used in computer graphics to enable the realistic simulation of virtual scenes in video games, in physical simulations such as particle physics, or even in robotics for automatic path planning. While elastic collisions are used in physical simulation, inelastic collisions often suffice for video game and graphics simulations, and we will mainly explore inelastic collision in this paper.

Continuous collision detection - which works by detecting an object’s predicted trajectory and ensuring that it nevers warps inside another objects through the allowed boundary - is more accurate for the appearance of a simulation, and without it objects may get stuck inside each other and be in a continual state of collision, which is undesirable. However, it makes only a small impact on the performance, whereas the underlying collision detection algorithm used makes the biggest impact on performance. Thus, we have chosen to use discrete collision for our performance calculations. Indeed, this causes only a small fraction of total objects within our scenes to become stuck inside each other, which has negligible impact on performance.

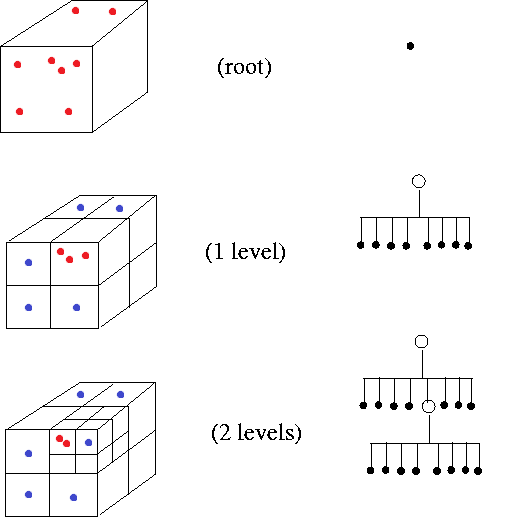
# **2 Related Work**

Much work has been done on improvements to these basic three algorithms, such as choosing the best sweep direction and reducing false overlaps for Sweep and Prune [Liu et al. 2010], or displaying the structure of an octree used within a scene in order to visualize the AABB locations [Chandran et. al. 2000]. Since many of these algorithms - especially in more recent papers - focus on complicated improvements to the BVH, or use recent hardware/GPU enhancements, they are often not used except in cutting-edge applications. For more common applications, more basic algorithms are applied, and so we have focused on these basics.

# **3 Octrees**

Octrees are one of the most commonly used data structures for efficiently managing a BVH. The algorithm recursively splits a scene of objects up into a number of octants, and checks collisions only within each octant. The advantage of this method is that far fewer collisions must actually be handled compared to brute-force. Within each sub-octant, collisions are only handled if there is more than one object within the sub-octant bounds. Otherwise, the octree is not generated any further, and collisions are not handled within. The classic algorithm does not actually check if the objects within a sub-octant are colliding; this, however, can lead to two objects which appear far away and are on opposite sides of the sub-octant, but are detected as colliding. For this reason, I have chosen to actually check collisions within each sub-octant so that the simulation is as accurate as possible.

As an example, in Figure 2 there are 8 objects, where colliding objects are colored red and non-colliding objects are colored blue. For the brute-force collision detection algorithm - equivalently, a 0-level octree - each object must be compared to one another for a total of 28 collisions. The 1-level octree has only three objects which are colliding, and thus only 3 collisions must be checked. In the 2-level octree, only 1 collision must be checked - for the single colliding pair of red balls. Thus for a 2-level octree, 27 of 28 collision pairs are culled in this example.



**Figure 2: An octree with 2 levels for a scene of 8 objects**

The more levels created within the octree, the more efficient the culling is, but since the octree grows exponentially in size with the number of levels, a deep octree may require more time to generate than it saves. Further, since objects in more than one sub-octant are included in both, a very deep octree may be less efficient than a brute force algorithm, since collision pairs may be duplicated across boundaries smaller than the size of the objects themselves. Scenes with a small number of objects may create very shallow octrees, but for scenes with many objects, a deeper octree will be generated. Having a deeper octree will cause a sparse scene to simulate quicker since many branches will be culled, but a dense scene will simulate slower in comparison to a shallow octree due to the overlap alone. Thus the maximum number of levels allowed is critically important to the performance of the octree algorithm, and is dependent on the number of objects in the scene and how dense the scene is.

# **4 KD Trees**

KD Trees

# **5 Sweep and Prune**

Sweep and Prune

# **6 Results and Analysis**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Simulation Name | Frames Simulated | Slowest Framerate | Quickest  Framerate | Least  Compared | Most Compared |
| No Collisions | 199 | 13.8414 | 20.286447 | 0 | 0 |
| Brute Force | 32 | 3.000265 | 3.159149 | 130816 | 130816 |
| Octree 1 Levels | 37 | 3.339111 | 3.75212 | 572 | 628 |
| Octree 2 Levels | 36 | 3.159687 | 3.531472 | 663 | 909 |
| Octree 3 Levels | 39 | 3.244116 | 4.287352 | 707 | 1586 |
| Octree 4 Levels | 57 | 2.858051 | 10.005186 | 282 | 2246 |
| **Octree 5 Levels** | **75** | **2.858024** | **12.017064** | **121** | **3117** |
| Octree 6 Levels | 63 | 2.401134 | 12.007982 | 124 | 5395 |
| Octree 7 Levels | 31 | 1.714951 | 6.668846 | 2178 | 15617 |
| Octree 8 Levels | 10 | 0.750217 | 0.93783 | 47653 | 65311 |
| Octree 9 Levels | 4 | 0.207679 | 0.207679 | 262144 | 344280 |

Table 1: **Octree performance after 10 seconds for 512 objects in the scene,**

Table 1 was generated using an AMD Phenom II X4 processor. Each scene is simulated with a number of spheres that have cubic bounding volumes, and are oriented inward towards the origin initially such that the scene is densely populated at first, but then sparsely populated as time passes, as the spheres move further apart from the origin of the scene. The average (as total/time), maximum and minimum framerates are calculated, and the approximate number of pair-collisions evaluated are listed in the last two columns. When the scene is sparse near the end of the simulation, very few pairs are collision-checked, but when the scene is dense, many pairs are checked. The framerate is more important than the number of collision checks, since the latter does not take into account how much time is needed to generate the octree on each frame.

For reference speeds, the 512 object scene was computed first with no collisions, at an average rate of 19.9fps. Then, the scene was calculated with brute-force collision detection, at an average rate of 3.2fps. The optimal number of octree levels is 5 for 512 objects in the scene - this achieves approximately double the average performance of the brute-force algorithm. However, the bottleneck framerate is 2.85fps, which is slower than the brute-force algorithm. This occurs because the scene is dense. Reducing the octree to 1-3 levels for the dense scene can remedy this bottleneck, at the expense of highly reduced performance in the sparse scene. Using 8 or 9 levels actually reduces performance in any case, since the octree is so large.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Simulation Name | Objects | Last Frame | Min FPS | Max FPS | Min Compared | Max Compared |
| Octree 3 Levels | 128 | 291 | 12.968157 | 59.294344 | 25 | 338 |
| Octree 4 Levels | 256 | 198 | 6.660184 | 30.116998 | 37 | 934 |
| Octree 5 Levels | 512 | 75 | 2.858024 | 12.017064 | 121 | 3117 |
| Octree 5 Levels | 1024 | 10 | 0.78997 | 1.132408 | 5913 | 9546 |

Table 2: **Octree performance after 10 seconds for the optimal number of levels**

Table 2 was generated in the same manner as before, but the optimal number of levels for the octree is shown for each number of objects in the scene. It is immediately apparent that a larger number of objects requires a larger number of levels within the octree, which is intuitive.

From the two tables above, a rough formula can be generated as a rule of thumb for the number of octree levels L needed in dense scenes where n objects collide, each with a 6-faced AABB:

L = min(6, log2(n)-4)

And for sparse scenes:

L = min(6, log2(n)-4) - 2

# **7 Conclusion**

Conclusion

# References

Chandran S., Gupta A.K., Patgawkar A., A fast algorithm to display octrees. Indian Conference in Computer Vision, Graphics and Image Processing (ICVGIP) 2000, organized by CAIR, DRDO.

Liu, F., Harada, T., Lee, Y., Kim, Y. 2010. Real-time Collision Culling of a Million Bodies on Graphics Processing Units. ACM Trans. Graph. 29, 6, Article 154 (December 2010), 8 pages.

DOI = 10.1145/1866158.1866180 http://doi.acm.org/10.1145/1866158.1866180.