**Broad-Phase Collision Detection for a Rigid Body Based Physics Simulation**

Sam Oates

**Abstract**

This paper aims to contrast and compare the benefits and draw backs of different broad-phase collision detection algorithms used within a real-time physics simulation.

**Introduction**

**The Problem**

Advanced physical simulation of many rigid bodies is a costly process to simulate in real-time. Close-Phase collision detection algorithms such as AABB (Axis aligned bounding boxes) or OBB (Object aligned bounding boxes) test against O(n2) bodies.

**Background**

Broad-phase collision detection algorithms are; for the majority based around some form of splitting the simulation world into smaller chunks. Splitting the simulation allows bodies within each chunk to only have to test against siblings that also reside in said chunk. This reduces the collision algorithm from O(n2) number of collisions to a much smaller easier to handle number of collisions. One type of broad-phase collision detection algorithms are based around a tree of data, such as; Binary space partition (BSP) trees, Quad trees, Octrees and K-d trees, rigid bodies are then attached to branches of said trees. A more popular type of broad-phase collision detection is the sweep and prune algorithm.

**Binary Space Partitioning**

Given certain parameters, binary space partitioning is the method of recursively dividing a simulation world in two until said parameters are met.

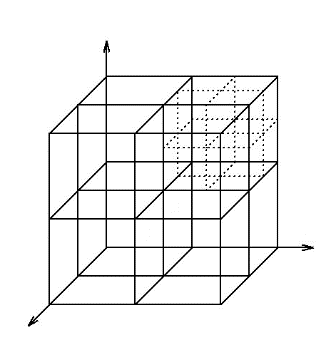
BSP trees can be advanced by adjusting the tree at runtime as proposed by Rodrigo G. Luque [1] in Broad-Phase Collision Detection Using Semi-Adjusting BSP-trees. Rodrigo suggests scheduling checks to re-evaluate the BSP tree. Should the BSP tree become unbalanced several strategies to re-balance the BSP tree. Rodrigo proposes altering cutting planes of the tree and re-structuring the tree based upon cost, potentially deferring the re-structure. Doing so can improve performance, however can also hinder the performance of the physics simulation due to the nature of re-evaluating the simulation world and moving data about the tree regularly.

Figure 1: The potential layout of how an octree divides 3D space.

Binary space partitioning is more commonly used with static data, such as graphical rendering and is still used within many modern graphical engines, however due to its performance limitations is rarely used within dynamic physics simulations.

**Quad and Oct Trees**

A Quad tree is a tree where each node within the tree is sub-divided into four sub-nodes, where each of the sub-nodes are sub-divided into four and so on and so forth until a given parameter is met [Samet 1984]. Reducing the number of collision detections required within the physic simulation as only bodies within the same sub-node are compared. Due to this, the number of rigid bodies within a node should never fall lower than two as a test between two rigid bodies is required.

Quad trees have the limitation of only splitting the simulation world into two dimensions. Thus why this technique is generally used within two dimensional simulations and terrain systems.

The three dimensional version of a Quad tree is an Octree [Moore and Wilhelms 1988]. The Octree similar to the Quad tree is split into sub-divisions, however rather than splitting into four sub-nodes the Oct tree is split into eight sub-nodes, covering three dimensions of virtual space rather than two.

**Sweep and Prune**

The Sweep and Prune technique is a pseudo-dynamic collision pruning algorithm which reduces three dimensional collision detection between AABBs into three separate one dimensional collision problems. Sweep and Prune is one of the most commonly used algorithms within broad-phase collision detection as it uses efficient pairs removal as well as it not depend on the rigid bodies’ complexity.

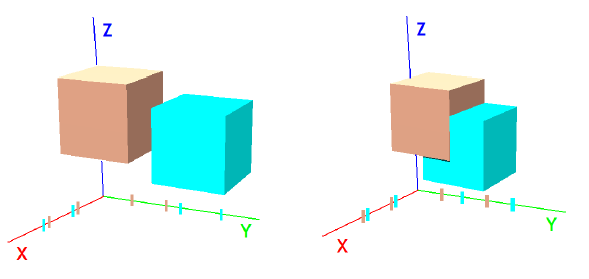


Figure : (Left) Non-overlapping sweep and prune (Right) Overlapping sweep and prune

Sweep and prune was developed by Baraff [3] and implemented by Cohen et al. [4]. The technique improves upon an existing technique called sweeping plane, it does this by exploiting the temporal coherence existent within physics simulations. Knowing that rigid body positions often change minutely each update, the projected sort order on all three world axis and the resultant output remain similar between updates. Therefor by storing the sorted interval lists and output results, then performing incremental changes for movement, simulation performance is substantially improved. A projection of higher and upper bounds of each rigid body (AABB) is made on the three different world axis. Three lists containing overlapping pairs on each axis are then obtained.

**Paralysation of Collision Detection**

With limitations of hardware speed becoming a factor in modern computing, paralysation of algorithms is an ever present solution to performance issues.

**Making use of the GPU**

Using the GPU for paralysation relies on image-based algorithms. The use of this has been suggested to exploit the growing computational speed of graphical hardware due to its very efficient rasterization of polygons.

Image-space visibility queries have been proposed to perform the broad-phase collision detection process [8]. Cinder [15] is an algorithm using and somewhat exploiting the GPU to implement a ray-casting method which is used to detect collisions. It is also worth mentioning GPU based algorithms for use with self-collision and cloth have also been introduced [7].

**Paralysation on the CPU**

Few papers have addressed parallel collision detection algorithms using multi-cores. However a task splitting approach for implicit time integration and collision detection upon multi-core architecture has been proposed [24].

Tang et al. [22] suggests using a hierarchical representation to help accelerate collision detection queries as well as using an incremental algorithm exploiting temporal coherence, this is then distributed among multiple cores. Using this technique a 4 - 6 times speed-up on an 8-core based machine was achieved.

Kim et al [12] proposed the use of a feature-based bounding volume hierarchy (BVH) to help improve the performance of continuous collision detection. Along with this it also proposed the novel task of decomposition methods for their BVH-based collision detection as well as dynamic task assignment methods. Using said techniques a 7 – 8 times speed-up using 8-cores compared to a single-core.

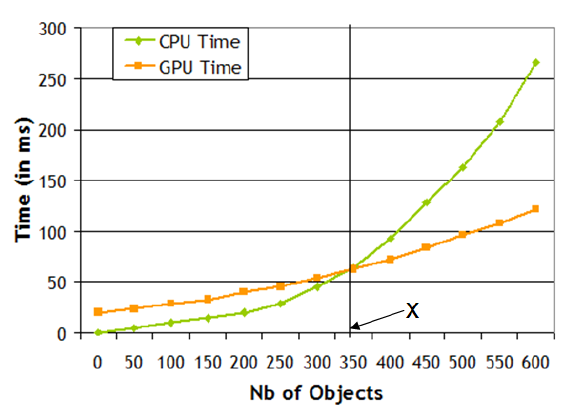
A first adaptive parallelization of the pipeline stages has also been proposed. [2] The narrow and broad phases are executed at the same time using a buffered structure this enables the body to dynamically adapt thread repartition during the simulation.

Figure 3: Comparing performance of paralysation upon the CPU and GPU

**The Proposed Implementation of Broad-Phase Physics Detection**

**Overview**

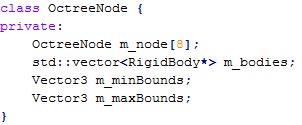
Although the performance improvements of paralysation on both the GPU and CPU show great promise, some limitations and complications are in place which make generic paralysation is somewhat more difficult and limiting for a generic broad-phase physics solution.

For said reasons I propose sticking to the more widely used Sweep and Prune technique, however with a few alterations. Potentially combining the Sweep and Prune technique with a dynamic octree could produce the resultant performance gain required. Rather than splitting the simulation world down to a size of say four bodies per octree node, it could be split to a higher level then use the Sweep and Prune algorithm to filter down more before using a narrow-phase procedure to check for collisions between rigid bodies’.

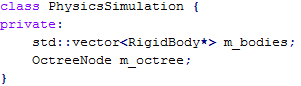
**Octree Implementation**

The basics behind an octree is to split the three dimensional world into eight segments (nodes) then each node into eight segments and so on traversing the world space until a given parameter is met.

The data required to represent this within code is very minimal;

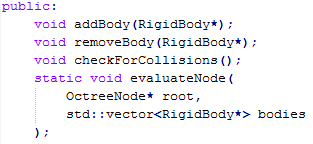


Simply storing the eight child nodes and a list of rigid bodies will suffice, however for ease of use storing the known bounds of the octrees node can also be stored. The above code could also store a parent node, allowing for easy bi-directional traversal of the octree.



Simply creating an instance of an octree node within the simulation is all that is required.

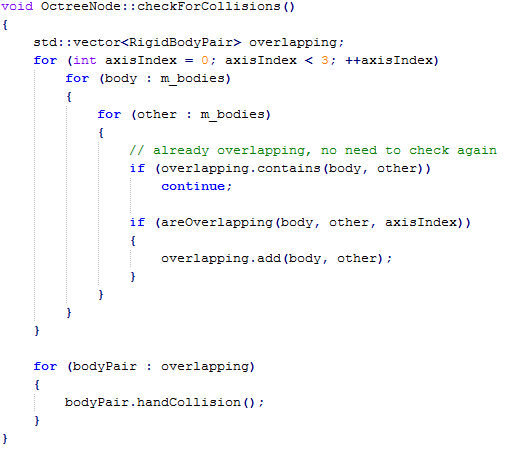
The public interface for the octree node is also quite simple as the majority of the code lies within the narrow-phase collision detection within the rigid bodies’ themselves. The *checkForCollisions()* method can simply be called on the root of the tree, in doing so all rigid bodies within that node will be checked, then traversing down through the trees nodes, checking only for collisions between bodies’ within that node.



There is also a static method used to evaluate the nodes within the octree. Given an octree node and a list of rigid bodies, said bodies will be added to the relevant octree node based upon the position of the rigid body and the bounds of the octree node. Re-evaluating the entire tree could be an expensive procedure, however checking for collisions against bodies which should no longer be within a given node is also a hindrance to performance. It is a matter of evaluating the tree at a given interval, also rather than evaluating the entire tree, you could also start half way down the tree and split the evaluation into multiple passes.

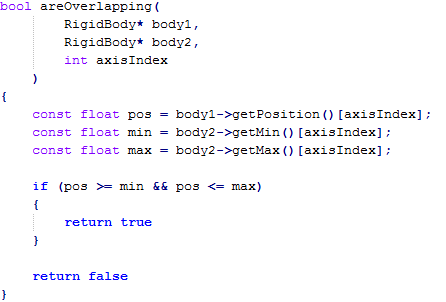
**The Sweep and Prune Implementation**

The sweep and prune phase will be implemented within the *checkForCollisions()* method of the octree. The basic principle is to only perform narrow-phase collision detection upon bodies whom have at least one axis which overlaps.



The above algorithm can be improved somewhat as this is performing the check n2 times, but the basic principle of the sweep and prune algorithm is addressed. To improve the algorithm the inner loop should start from the next body in the list from the outer loop, meaning the test is only performed once per rigid body.

The check for overlap method simply returns true (success) should any one of the three world spaces of the rigid bodies overlap.



**References**

<http://www.brandonpelfrey.com/blog/wp-content/uploads/2013/01/simple-octree.png>

<http://www.cs.berkeley.edu/~demmel/cs267/lecture26/Quadtree2.gif>

[www.inf.ufrgs.br/~comba/papers/2005/sabsp-i3d05.pdf](http://www.inf.ufrgs.br/~comba/papers/2005/sabsp-i3d05.pdf)