Analysis of Broad-Phase Spatial Partitioning Optimisations in Collision Detection

Kevin Glass*
Department of Computer Science, Rhodes University

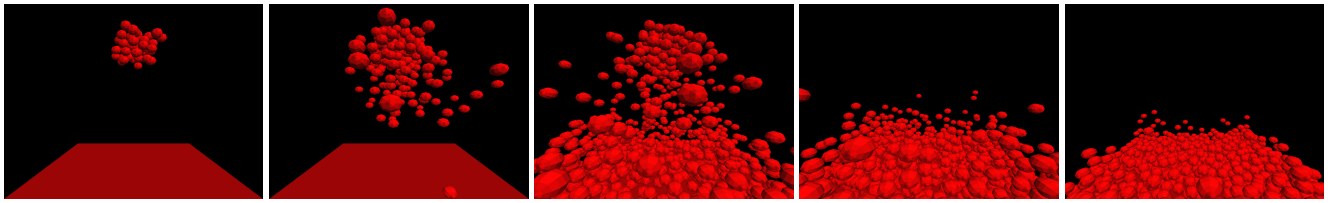

Figure 1: Dynamic simulation of bouncing spheres.

Abstract

This paper examines a number of broad-phase optimisations which can be used to improve the performance of collision detection in dynamic simulations. Specific focus is placed on hierarchical spatial partitioning techniques such as the octree, *k*-d tree and binary space partitioning (BSP) tree. A number of experiments are conducted using this subset of partitioning methods in order to evaluate their comparative performances in a controlled simulation. A generic structure is created and described which is able to implement each of the tree-based partitioning methods as well as a number of variations on some of the methods. This structure is generic in that the partitioning method implemented depends only on the initial hyperplanes passed to it as a parameter. The results of these tests are analysed in order to identify factors which may affect the extent to which each scheme optimises the collision detection process. Factors identified include allowable tree depth and branching factor, the choice of new partitioners in the recursion process, and the application domain of the partitioning scheme.

CR Categories: I.3.15 [Computer Graphics]: Computational Geometry and Object Modeling—Object Hierarchies

Keywords: Collision detection, optimisation, hierarchical partitioning, octree, *k*-d tree, BSP-tree

1 Introduction

1.1 Problem Statement

This paper investigates the problem of broad-phase collision detection in dynamic simulations. This includes the analysis of different spatial partitioning techniques, specifically quad/octrees, *k*-d trees and BSP-trees. The analysis includes a number of experiments which are designed to expose factors which affect the performance of each partitioning technique.

1.2 Background

Real time collision detection is an essential feature of many types of application including dynamic simulations, ray-tracing, robotics, engineering simulations, molecular modeling and electronic prototyping [Lin 1999]. Simplistically defined, *collision detection* is the process of discovering when two or more objects intersect. *Collision response* generally refers to the calculation of new linear and angular velocities for colliding objects [Moore and Wilhelms 1988]. A dynamic simulation of bouncing balls, for example employs collision detection between each ball in the simulation, and if collisions are detected, employs some collision response algorithm to calculate the subsequent bounce, based on physical phenomena such as momentum.

Realistic dynamic simulation, however, is complicated by a variety of factors. For instance, the more complex the shape of an object, the more difficult it becomes to calculate whether two such objects intersect. Additionally, the number of collision tests increase quadratically in relation to the number of dynamic objects in the simulation. Finally, the deformation of objects as they collide presents further challenges to a realistic dynamic simulation.

A number of optimisations have been presented in order to reduce the amount of computation required for collision detection, which can be roughly categorised into two groups [Luque et al. 2005; Mirtich 1998]:

- Broad-phase: a fast test which enumerates pairs of objects that potentially collide. This is also referred to as *rejection tests* [Lin 1999], where pairs of objects are rejected as potential collision candidates when they are too far apart.
- Narrow-phase: optimising the collision detection process between two complex objects.

Broad-phase algorithms typically aim to reduce the number objects to be checked for collision. They make use of *spatial partitioning* techniques to exclude objects which are too far apart. Narrow-phase partitioning aims to optimise the collision detection process between two complex objects which have passed through the broad-phase filter. Narrow-phase detection works by encapsulating the objects within a number of simpler constructs (for example, spheres) at different resolutions. If any of these constructs collide, then the underlying geometry within that construct is tested for collision, effectively reducing the number of collision tests required.

Broad-phase collision detection is not concerned with detailed geometry, and generally tests for intersections between the *bounding volumes* which encapsulate each object. Various types of bounding volume exist, including bounding spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB) and *k*-DOPS (discrete oriented polytopes) [Klosowski et al. 1998]. Many structures have been developed to spatially partition these bounding volumes in order to reduce the number of collision tests which must occur in the narrow-phase. These structures include quadtrees, octrees, *k-d* trees, binary-space-partition trees, BRep indices and regular grids.

This paper investigates and tests a number of spatial partitioning

*e-mail: kglass@rucus.net

methods. Each method is explained and its performance compared to that of a baseline method. The baseline method is the brute force solution to collision detection in which each object is checked for collision against every other object. This approach is often classified as having a worst case runtime of $O(n^2)$ [Lin 1999; Luque et al. 2005]. The spatial partitioning methods presented here are used to reduce this runtime.

1.3 Overview of paper

The rest of this paper is thus structured as follows: Section 2 gives an overview of a number of broad-phase spatial partitioning methods, a brief discussion of narrow phase optimisations, as well as an overview of similar performance studies on partitioning techniques. Section 3 presents the experiments done to evaluate the performance of each partitioning scheme, and discusses the results of these tests. Finally Section 4 gives a summary of the paper and briefly discusses further tests which may be done for a more thorough evaluation of broad-phase collision detection optimisations.

2 Related Work

A variety of broad-phase and narrow-phase optimisations exist for collision detection. This section provides a survey of a number of broad-phase partitioning methods used in collision detection, and briefly discusses some techniques used in the narrow-phase. Additionally previous comparative results between different spatial partitioning techniques are discussed.

2.1 Broad-phase (spatial) partitioning

The idea of spatial partitioning is to divide a space into distinct regions. Objects within the space are grouped according to these regions with the aim of reducing the number of candidates for collision testing. For instance, if object A is grouped according to region R_1 and object B is grouped according to region R_2, assuming R_1 and R_2 are disjoint, then no collision test is necessary between object A and B. A complication to this idea is that some objects may lie on the border of two regions. One solution is to subdivide the object into sub-objects, with each sub-object falling into a distinct region. A simpler solution is to include the object in both regions. For instance, if object A lies on the border of region R_1 and R_2 then it will be included in both regions, and hence a collision test will be necessary between object A and B.

The above discussion leads to the conclusion that there are some characteristics which differentiate the effectiveness of partitioning methods. These include the number of objects in each region; the distribution of objects over the regions; and the number of objects which need to be grouped into more than one region. These characteristics will be analysed in the context of each partitioning system.

2.1.1 Regular Grid

The most intuitive method for spatial partitioning is to divide a region into evenly spaced partitions. A two-pass method is often used [Teschner et al. 2003]. In the first pass the space in $\mathbb{R}^3$ is implicitly subdivided into a regular grid of cells. This is done by discretising the positions of all object vertices with respect to a cell size l. Each component in the vertex position (x,y,z) is divided by l and rounded down to the nearest integer. The results are then hashed to a 1D index and the object information stored in a hash table at this index. During the second pass only the objects which fall in the same buckets are tested for collision.

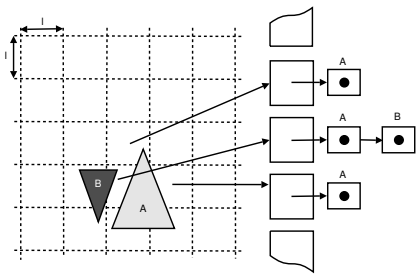

Figure 2: 2-dimensional representation of a regular grid and the mapping to a hash-table.

Figure 2 presents a two dimensional representation of the first pass. The number of objects which need to be tested for collision is dependent on the size of l. If l is too large then a large number of objects will be mapped to a small number of buckets, increasing the number of collision tests. If l is too small then objects overlap a number of regions, adding to the number of collision candidates once again. Additionally, the type of hash function used as well as the number of buckets used in the hash-table affects the number of objects which need to be tested.

Multi-resolution hash-tables have been used as a solution to the problem of finding an ideal value for l [Mirtich 1998]. The region is partitioned in increasing levels of resolution, each level having its own associated hash-table. The highest level of resolution is scanned first, and any region in a level of lower resolution which overlaps this region is included in the collision tests.

The major flaw with a regular grid partition is lack of adaptivity to the location of objects in the simulation. For instance, a large number of objects clustered together will result in a hash-table with a small number of buckets which have a large number of objects. Ideally a scheme which can focus on populated areas and attempt to partition only these areas is desired.

2.1.2 Quad/Octree

Quadtrees and Octrees are data structures which have the ability to focus in on regions of space which are populated. A hierarchical data structure is used to provide a multi-resolution representation of the location of objects in a simulation.

The quadtree [Samet 1984], initially used in image processing, works by partitioning a region into four equal quadrants. The root node of the tree represents the space with no partitioning, and has four children, each representing a partitioned quadrant. These children are parents to further subtrees, where each quadrant is repeatedly partitioned in a similar manner. Figure 3 presents a 2D example of a *region-based* quadtree, which divides the space into equal sized regions. The leaves at each level (shaded in gray) indicate empty quadrants. Non-empty quadrants are recursively partitioned until a predefined depth is reached, or until only one object per node is achieved. In this example a predefined depth of 3 is specified, and each leaf node on level 3 is used as a bucket for storing possible collision candidates.

The concept discussed above can be adapted to three dimensions. In the case of a quadtree, a space is divided into four equal sized parallelepipeds. Alternatively, as in the case of the octree, the space is divided into eight equal sized octants, resulting in each node having eight children [Samet 1984].

An alternative type of quadtree to the one presented in Figure 3 is a *point-based* quadtree. This method does not divide the space into equal sized regions but rather chooses some point from the region's objects as a center for the next level of partitioning. The optimal

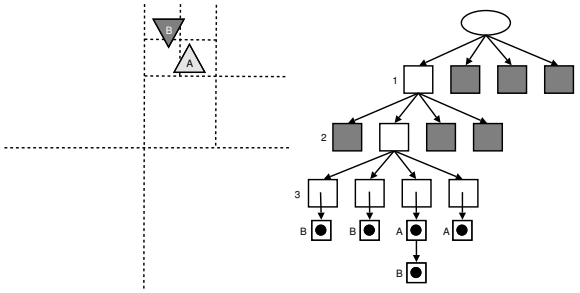

Figure 3: Quadtree partitioning of a 2-dimensional space.

choice of this point, however, often implies a search[Samet 1984]. This technique has also been extended to octrees [Moore and Wilhelms 1988].

2.1.3 *k*-d Tree

Extending a quadtree to an octree in the previous section raised the branching factor of the tree from four to eight. Generally extending this kind of partitioning into a *k*-dimensional space will result in a branching factor of 2^k branches per node. This kind of branching is undesirable and reduces the efficiency of searching through the tree. The *k*-d tree is proposed as a multidimensional space partitioning technique [Bentley 1975] which has a branching factor of 2, making it a variation of a regular binary tree. A set of *discriminators* exists which can be used to partition a space into two regions. At each level of partitioning only one discriminator is used from the set, and there is strict alternation between the choice of discriminator. In a dynamic simulation context these discriminators are partitioners or *hyperplanes.*

The root node in a *k*-d tree represents the entire space. Two decisions need to be made when choosing the next partitioner [Held et al. 1995]. Firstly, the axis ($x-, y-, z-$axis) against which the hyperplane should be aligned needs to be determined, and secondly the position on the chosen axis that the hyperplane should be placed. These decisions are at the discretion of the designer of the system, and could range from choosing the median along the axis, to searching for the planes which produce the most balanced partitioning. Once a suitable hyperplane is found, the space is partitioned in two, with all objects on one side falling into the corresponding child of the current node. This process is recursive, and each space is in turn divided until some threshold (for example minimum number of objects in a partition) is reached. Figure 4 illustrates the workings of a *k*-d tree in 2D which alternates between $x-$ and $y-$axis-aligned hyperplanes at each level of recursion. An arbitrary position along that axis is chosen. All objects which fall into the same partition are candidates for the narrow-phase collision detection.

k-d trees can be implemented as both region-based and point-based trees. For instance, a region-based tree will choose new partitioners based on some spatial division, whereas a point-based tree would choose some point within a region and use the point as the reference point for the new hyperplane [Samet 1984].

2.1.4 Binary Space Partition (BSP) Tree

Binary Space Partition (BSP) trees are a generalisation of the *k*-d tree, the main difference being that hyperplanes in any arbitrary direction can be chosen (as against axis-aligned hyperplanes). The construction of a BSP-tree takes as input the region to be partitioned, and a *binary partitioner* (hyperplane), and produces two disjoint subsets of the input region [Naylor 1992]. As with the *k*-d tree the process is recursive and each partition is in turn partitioned.

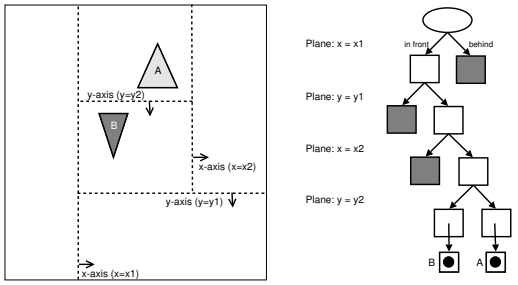

Figure 4: *k*-d tree in two dimensions.

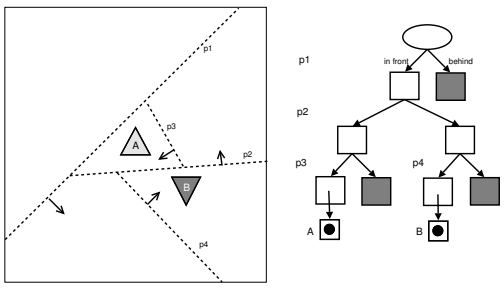

Figure 5: BSP-tree in two dimensions.

Figure 5 illustrates a BSP-tree in 2D. The diagram suggests that the correct choice of partitioner can provide more efficient division of regions than axis aligned methods. The choice of partitioner, however, is not a trivial one. Decisions when choosing a partitioner include the extent to which the angular range of a new partitioner is limited (for instance, within some angle from the normal of the parent hyperplane), as well as deciding where to place the hyperplane. Partitioners can be chosen based on goodness criteria [Luque et al. 2005]. The quality of a partitioner is evaluated using metrics such as:

- population: the number of objects tested against the partitioner
- balance: the measure of how evenly the partitioner splits objects in the region.
- redundancy: the number of objects which are intersected by the partitioner and will be placed in both partitions.

Evaluating the goodness of a partitioner, however, adds another level of time complexity to a collision detection system. For instance, evaluating the balance of a partitioner requires that each object in the partition be classified as *in-front-of* or *behind* every candidate partitioner. This process becomes feasible in cases such as [Luque et al. 2005] in which temporal coherence between time-steps is implemented in the simulation. In this case new partitioners are found for only those nodes where objects have moved significant distances over specified time intervals.

2.1.5 Other partitioning methods

A variety of other methods exist for spatial partitioning besides the ones mentioned thus far. Many of them, however, are extensions of one of the above techniques. For instance BRep indices [Vanecek 1991] extend the idea of BSP-trees to multiple dimensions in order to provide detailed information about the boundary representation of an object. This is done by recursively decomposing the d-dimensional hyperplanes into $(d-1)$-dimensional BSP trees. The space is thus partitioned into 3-, 2-, 1- and 0-dimensional regions.

An R-tree is a scheme which partitions the set of obstacles associated with a node, rather than the space that the node represents [Held et al.

1995]. For instance, the objects at a certain node are split by finding a median value between the centroids of the objects, and partitioning the objects based on their relationship with the centroid.

A technique named *sweep and prune* is also used [Lin 1999]. This approach projects the extreme edges of an object's axis-aligned bounding box onto the $x-$,$y-$ and $z-$axes. The result is a set of intervals along these axes. A collision occurs if an overlap in interval occurs in all three dimensions.

A dimension of collision detection which is not handled by these schemes is time. The major problem is that most dynamic simulation use a time-step based approach where object positions are updated at discrete intervals. If an object is moving at a very large velocity there is a good chance that it will appear to have gone through an obstacle, since the change in position between time-steps was large enough to move the object beyond the obstacle without experiencing any collisions. A *swept volume* approach has been proposed to solve this problem [Mirtich 1998], where a bounding box is created which covers an object's entire trajectory space over a time-step. This solution only works when the trajectory is predictable and simply calculated. Another technique is to maintain a priority queue of object pairs that might collide, sorted with estimated time-to-collision, in order to provide real-time performance [Lin 1999]. Other solutions include using four dimensional representations, where the fourth dimension represents time [Klosowski et al. 1998], to bound the position of objects within the near future.

Often in dynamic simulations the geometric relationship between objects only differs slightly between successive timesteps. In addition to this, a simulation may contain a number of static objects which do not move at all. These properties allow the the implementation of *temporal coherence* [Lin and Gottschalk 1998] between successive frames. This means that a new broad-phase partitioning structure does not need to be applied at each new frame, but rather the same structure used repeatedly over a number of frames and updated with the new object positions. As mentioned previously, temporal coherence can be implemented where the spatial hierarchy (BSP-tree) is updated at regular intervals [Luque et al. 2005]. Updating only occurs when certain conditions of the BSP-tree are met, since in a system of many dynamic objects, dynamically updating the tree can consume as much time as building the tree from scratch at each frame. As object positions are changed at each frame, the leaves of the BSP-tree which contain ordered lists of objects are also updated. Using ordered lists at the leaves allows the exploitation of temporal coherence to optimise the update process.

2.1.6 Bounding Volumes

Broad-phase collision detection aims to reduce the number of collision candidates to be tested for collision at a later stage. Therefore, the partitioning tests to be done during the broad-phase need to be efficient, but do not need to be completely accurate. It is for this reason that complex objects are rather encapsulated within a convex bounding volume for which collision tests are more efficient. During the broad-phase, it is these bounding volumes which are compared for partitioning purposes.

A number of bounding volumes can be used to encapsulate complex objects. Although the precision when using such volumes is more relaxed, it still holds that the more closely a bounding volume approximates the shape of an object, the more efficient the collision detection process becomes.

The simplest type of bounding volume is the bounding sphere, where the entire object is encapsulated by a single sphere. This option is attractive since it is simple to calculate intersections between spheres. However, since not all objects are spherical in shape, a bounding sphere often does not produce a tightly fitting bounding volume.

Axis-aligned bounding boxes (AABBs) are parallelepipeds whose faces are aligned with the basis vectors which define the coordinate system [Eberly and Schneider 2003]. Intersection testing between such volumes entails projecting the dimensions of the bounding box onto all the basis vectors, and determining whether an overlap occurs on at least one axis. Once again, this type of bounding volume does not always closely bound an object, and hence the development of the Oriented Bounding Box (OBB). This bounding volume is a parallelepiped whose faces are not aligned with the basis vectors which define the coordinate system. A similar method for detecting collision between OBBs is used as for AABBs, except the OBBs are projected onto an arbitrary axis to determine if overlap occurs, rather than a basis vector.

Finally, a *k*-DOP (discrete oriented polytope) is presented [Klosowski et al. 1998], where the bounding volume can be formulated using *k* facets whose outward normals come from a small set of fixed orientations. This structure allows a tighter fit with the object than AABBs and OBBs, since a larger number of facets is used.

2.2 Narrow-phase partitioning

Once the number of collision candidates has been filtered down by the broad phase collision detection process, more accurate (but more time-consuming) algorithms are used in the narrow-phase to determine whether objects penetrate. Such methods include techniques such as the Lin-Canny algorithm [Mirtich 1998] for finding the closest points between two objects. It works by dividing the surrounding space of a convex-polyhedron into Voronoi regions. If points on both objects fall within a Voronoi region of the other, then the two points are the closest points between the objects. This process is iterative in nature, but eventually terminates after settling on the closest points. Pure algebraic intersection testing can also be done. In ray-tracing, for example, an intersection between a ray and a triangle can be found by parameterising the triangle with respect to barycentric coordinates [Buss 2003].

The above collision detection processes provide examples of computationally expensive methods. Therefore, the fewer of these tests that are the done, the better the performance of the system. Broad-phase partitioning is responsible for filtering out entire objects which need not be compared for collision. However, this can be extended to the narrow phase, partitioning the object space.

Octrees, for example, are not limited to broad-phase optimisation, but also apply in narrow phase collision detection [Bandi and Thalmann 1995]. In the broad-phase the bounding volumes of two complex objects are used in conjunction with the octree to decide if the objects are potential collision candidates. In the narrow phase, the object space of the two objects are further partitioned using octrees to speed up collision detection between the individual polygons making up the objects. Therefore, for complex objects constructed from a large number of primitives, such partitioning can reduce the number of intersection tests required between two objects.

Simple constructs such as spheres can also be used to optimise narrow-phase collision detection [James and Pai 2004]. An object is encapsulated with a number of close fitting spheres which adjust automatically to any deformation that the object may experience. These spheres are then used in the narrow-phase to reduce the number of collision-tests. This method can be applied as a multi-resolution sphere tree, which begins with one bounding sphere at the root which covers the entire objects. At each level of the tree more and more spheres are used to cover the object with higher precision [Hubbard 1996].

Another example of narrow phase partitioning involves partitioning an object into a hierarchical structure of oriented bounding boxes called an OBBTree [Gottschalk et al. 1996]. This technique involves tightly fitting OBBs around groups of polygons, and then nesting

these OBBs into a tree hierarchy. Such partitioning allows for the accurate detection of collisions between objects at interactive rates.

2.3 Evaluations of spatial partitioning techniques

Many evaluations have been done which compare various optimisation schemes in terms of performance. Such tests occur most frequently when a new technique is developed, as is the case in Luque *et al.* [2005], where the new technique for updating BSP trees is compared against quadtrees, spatial hashing and a sweep-and-prune approach. Metrics such as frames per second (FPS), number of collision pairs, broad-phase collision time and update time are collected. In this particular case, the quadtree and BSP-tree produce the best performance improvements.

Various hierarchical methods have uses which extend to many other domains including ray-tracing. A number of performance tests in a ray-tracing environment have been done using spatial partitioning techniques for optimisation [Havran et al. 2000]. Such tests include BSP-trees, *k*-d trees, uniform grids and octrees. A number of test simulations are generated, some with more than 10^6 objects, which are used as test environments for the ray-shooting scheme. The *k*-d tree is reported as having the best overall performance.

Another comparison between partitioning techniques is presented in [Held et al. 1995] where a grid-based technique, *k*-d tree, a custom defined mesh-based scheme and R-tree are compared. Each scheme is evaluated over a number of randomly generated "obstacle" environments. This experiment reveals that the R-tree is the best performing collision detection scheme.

Finally Levey *et al.* [2000] present metrics for evaluating collision detection techniques based on performance, scalability, robustness and ease of implementation.

3 Experimentation

The aim of this experimentation is to evaluate the performance of a subset of the discussed broad-phase collision detection schemes. Tree-based optimisations in particular are examined, including octrees, *k*-d trees and BSP-trees. A generic structure for implementing these optimisation schemes is created, and as a result a number of modified versions of these schemes are also tested. In particular, the following tests are conducted:

- Determine which algorithm improves performance by the largest margin.
- Determine the impact of increasing the depth of the associated tree.
- Determine the characteristics of each algorithm in terms of tree construction as well as the number of collision tests which need to be performed.
- Determine which algorithm performs the best consistently as more objects are added to the simulation, and thereby identify the point at which the various schemes fail.

3.1 Design

In order to evaluate the performance of each partitioning technique, a baseline model is required against which each technique can be compared. This baseline model is the worst case situation, where each object in the simulation is tested against every other object. Therefore the number of collisions in this model have a quadratic relationship with the number of objects in the simulation.

Two types of octree are implemented in this experiment. The first type of octree is a *region-based* octree, and takes the centre point of each octant as the centre point for three orthogonal axis-aligned hyperplanes. The second type is *point-based* where the centre point of each octant is the median of all the objects in that octant. The octree is simplified to form a quadtree, where the region is divided into four equal sized parallelepipeds, and different combinations of axis aligned planes are tested. Finally a binary version of the tree is tested where only one axis-aligned plane partitions the space (bin-tree). Once again, performance is measured for different planes.

Two types of *k*-d tree are also implemented, one regional and the other point-based. The *k*-d tree begins with an axis aligned plane which partitions the space, and at each level in the recursion chooses the next axis-aligned plane from a set. In this 3-dimensional case, level i in the tree will correspond to plane $(i \, modulo \, 3)$ in the set of available planes. For instance, on level one, the $x-z$ plane is the partitioner, and on level two the $x-y$ plane is the partitioner. The region-based *k*-d tree chooses the centre point in the region to determine the location of the new plane, while the point-based *k*-d tree uses the median of the objects within the region.

A BSP-tree is implemented in the following manner. At each level of recursion a normal is created by taking the direction from the median of objects in the region to the previous reference point on a plane. This normal is rotated by ninety degrees around the $x-$, $y-$ and $z-$axes. A new hyperplane is formed using this normal, with the median as the new reference point.

Each of these trees are tested using different depths in order to evaluate the effect of tree depth on the performance of the optimisation.

Statistics gathered include the time taken for the entire collision process to complete (including broad-phase collision detection and collision response), and the number of collision-tests which need to be performed at each frame.

Generic Structure In order to compare the optimisation schemes in a fair manner, a generic structure is implemented that can represent octrees, *k*-d trees and BSP-trees. The structure requires that a set of planes be added which will be used as partitioners. A division function iterates through this set of planes, recursively partitioning a region with regards to the specified planes resulting in 2^p partitions if p planes are added. Once a region is partitioned according to this set of planes, each resulting partition is recursively partitioned in turn. Choosing the new centre for the set of planes (or modifying the planes' orientations) at each level o recursion differentiates the partitioning schemes from each other. For instance, at this stage a region-based octree would calculate the centre of the new partition and create a new set of axis aligned hyperplanes at this point. A BSP-tree however, creates only one new plane with a randomised orientation.

This structure is designed as a class containing a single publicly accessible method, CONSTRUCT, which takes as input: a set of planes which will partition the space; the node in the tree representing the current region to be partitioned by these planes; the set of objects in this region; the current tree depth; and the type of partitioning to occur. Two private methods are also used in this class, SINGLESPLIT, and SPLIT. CONSTRUCT is responsible for choosing a new set of hyperplanes for each partitioned space, while SINGLESPLIT is responsible for the partitioning of the space accordingly.

SINGLESPLIT is a recursive function which iterates through the set of planes, partitioning the current space accordingly. Determining whether an object lies in-front-of or behind a plane is implemented in SPLIT. Once the region is fully partitioned by all the planes in the set, a new node is created in the tree for each partition. This node contains a list of pointers to the objects it contains. The algorithm implemented in SINGLESPLIT is presented in Algorithm 1.

Algorithm 1 SINGLESPLIT function

```
void singleSplit(
  Node parent,
  Collection planes,
  Collection spheres,
  int i, //which plane this split uses
  )
{
  if (i < size of planes){
    nf = Empty Collection of spheres (front);
    nb = Empty Collection of spheres (behind);
    //Split the collection of spheres
    //according to plane i from planes and
    //puts all spheres in front of the plane
    //in nf, and those behind the plane in nb
    split(ith plane, spheres, nf, nb);

    //Split collection in front of plane
    singleSplit(parent, planes, nf, i+1, fm);
    //Split collection behind plane
    singleSplit(parent, planes, nb, i+ , bm);
  } else {
    n = new Node;
    n.spheres = spheres;
    add n to the parent's children;
  }
}
```

Algorithm 2 CONSTRUCT function.

```
void construct(
  Vector point, //centre point of region
  Collection planes, //hyperplanes
  Node current, //current node of tree
  float width,
  Collection spheres,
  int depth, //Max depth of tree
  int type   //Type of tree
  )
{
  //Partition the region and add the
  //relevant children to the current node
  singleSplit(current, planes, spheres, 0);
  iterate through current's children{
    child = current.child[i];
    if (spheres in child > 1 && depth > 0){
       switch (type){
       //Each type defines its own unique
       //way of creating a new set of planes
       //for the next level of recursion
       case OCTREE: ...
       case POINTOCTREE: ...
       ...
       }
       //Recurse: newCenter is the reference
       //point of the new region; newplanes
       //is the set of new planes which will
       //partition the regions
       construct(newCenter, newplanes,
                 child, newWidth,
                 child's spheres, depth-1,
                 type);
    } else {
       //make spheres aware of surrounding
       //collision candidates
       iterate through all spheres in child{
         sphere[i].neighbours = node.spheres
       }
    }
}
```

	Original hyperplanes	New hyperplanes at each level of recursion
Octree	$x-y$, $y-z$, $x-z$ planes	$x-y$, $y-z$, $x-z$ aligned planes centered at midpoint of new partition
Point-based octree	$x-y$, $y-z$, $x-z$ planes	$x-y$, $y-z$, $x-z$ aligned planes centered at median of objects in new partition
k-d tree	$y-z$ plane	plane $(i \bmod 3)$ from the set of axis-aligned planes, where i is the current depth of the tree. The plane passes through the midpoint of new partition
Point-based k-d tree	$y-z$ plane	plane $(i \bmod 3)$ from the set of axis-aligned planes, where i is the current depth of the tree. The plane passes through the median of objects in new partition
BSP-tree	$y-z$ plane	New plane defined by median of objects in partition and reference point on previous plane

Table 1: Parameters for implementing different schemes with the generic structure.

The algorithm which CONSTRUCT implements is presented in Algorithm 2. Note that this structure makes the partitioning process similar over all differing types of techniques. The algorithm is parameterised only by the new set of planes which are chosen at each level of recursion. When the depth reaches zero, each object is made aware of the other objects which fall into the same partition. These surrounding objects are the collision candidates which will be passed to the narrow-phase.

3.2 Implementation

The simulation environment for this experiment consists of a virtual axis-aligned box into which 1000 spheres are dropped (see Figure 1). Spheres are used since many collision detection schemes use bounding spheres for complex objects. Since this experiment is aimed at testing broad-phase optimisations, more complex structures are not necessary. A simple test is done for collisions, where a collision occurs if the distance between the centres of two spheres is less than the sum of the radii of the spheres. If a collision occurs, the locations of these spheres are modified so that they no longer intersect. In addition, a simple momentum based calculation is done in order to determine the new velocities of the involved spheres. Each sphere is also tested against the boundaries of the box, and the correct component of the velocity is reversed to simulate a bounce off the boundary.

The generic structure is parameterised with the planes defined in Table 1 in order to implement the different types of trees. The "Original Hyperplanes" column indicates the initial set with which the entire region is partitioned. The "New hyperplanes" column indicates how new planes are created for the recursive partitioning. Note that the major difference between k-d trees and the other types is that the k-d tree chooses only one plane from the set of available planes, while the other methods use the entire set of planes for the next level of recursion. In addition, this implementation of a BSP-tree always begins with the $y-z$ plane, after which non-axis-aligned planes are created for the recursive partitioning.

During the creation of the tree structure, the SPLIT function is used to determine whether the objects lie in front of or behind the cur-

rent plane. Two points on each sphere are tested against the current plane to determine in which set the sphere falls. The two points are derived by adding the radius to the centre of the sphere in the direction and reverse direction of the plane normal. A point is in front of the plane $(x,y,z)\cdot(n_x,n_y,n_z)+d=0$ if a positive d is obtained when substituting the point coordinates into (x,y,z), where (n_x,n_y,n_z) is the plane normal. If the two points for the sphere fall both in front of and behind the plane then the sphere is added to both regions.

Once the specified depth of the tree has been reached each object in the set of spheres at each node is updated with the node reference in memory, which results in a pseudo-hashtable of collision candidates. Thus, after the creation of the tree each sphere will be aware of which subset of spheres to test for collision.

The simulation is implemented in C++ using OpenGL for rendering and is run on a 3GHz Pentium 4 HT processor with 1GB of physical memory. A new sphere is added to the simulation at each frame, until a total of 1000 spheres is reached. The simulation is allowed to continue until 2000 frames have been rendered. Each new sphere appears at the same location but with seeded randomised velocities. Gravity is simulated in the system, and thus the spheres eventually settle in the bottom of the box.

The simulation is updated as follows. At each frame the new position for each sphere is calculated, after which the spheres are tested against each other for collision. Collision against the boundaries are then handled. Since this experiment is aimed at testing the construction and usage time of the various types of trees, no spatial coherence is implemented, and thus a new tree is created for each frame.

Measurements are taken at each frame but averaged over 10 frames in order to smooth out any external processing activity which may take place. Each measurement contains the number of spheres in the system, and average timings for the collision process to complete over the past 10 frames. In addition to this each test is repeated 10 times over with the results of each frame-reading averaged in order to eliminate any spurious operating system processing which may have occurred.

3.3 Results

3.3.1 Overall Performance

Figure 6 presents the time taken for all collisions in the system to be resolved. The x-axis denotes the frame number (which corresponds the number of spheres until 1000 spheres have been added), while the y-axis denotes the total time taken to detect and respond to collisions. Note that the timings presented here do not include rendering time, and that each tree structure is limited to a depth of 3.

As expected the baseline approximates a quadratic function, which becomes constant when all spheres have been added to the simulation. All graphs show a levelling at approximately frame 1200, which indicates the point at which the spheres settle in the base of the box.

Both the region-based and the point-based octree initially perform more slowly than the baseline algorithm, but a crossover occurs in both cases. This poor initial performance is attributed to the complexity in constructing the initial octree, since each region must be divided into eight partitions. This is a large amount of complexity compared to the k-d tree and BSP-tree which are binary trees and result in only two partitions per region. Both BSP- and k-d trees perform equally well or better than the baseline algorithm. Once all the spheres are added to the system and the simulation stabilises, the octree provides the best performance, which indicates that the high cost of constructing the tree pays off in terms of performance.

This figure also presents the performance of region-based and point-based octrees and k-d trees. In the case of the octree, using the object median as the new centre point for the next level of partitioning (point-based) results in noticeably poorer performance than a region based octree, which merely uses the centre point of each partition. However, in the case of the k-d tree the point-based method is far more efficient. An explanation for this is that there is a trade off between the balanced partitioning of a region, and the number of planes used to partition a region. The octree uses a larger number of partitioning planes, and therefore the choice of the median of objects as a center point results in far more plane/object intersections. This results in reduced performance. The k-d tree however, uses a single partitioner which provides the benefits of a more balanced tree, without the overhead of too many plane/object intersections.

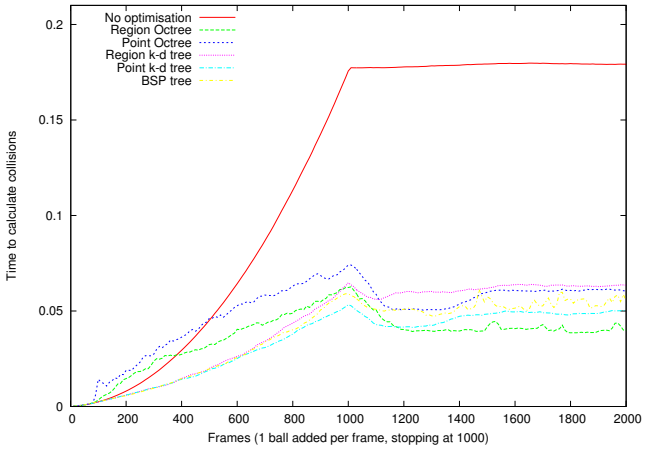

Figure 6: Total time of collision calculation.

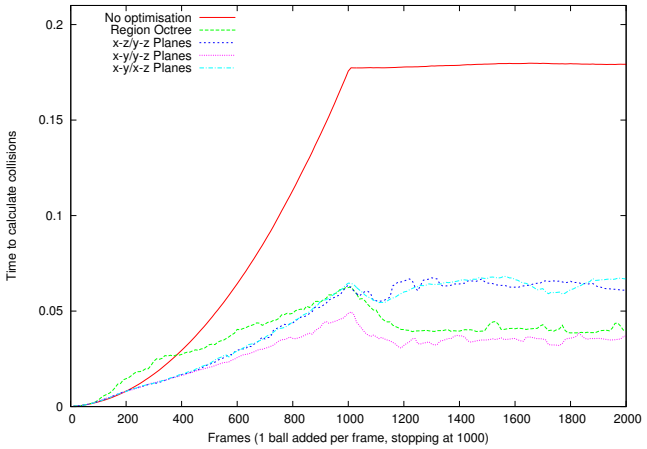

Figure 7: Total time of Quadtree with varying hyperplanes.

Figure 7 presents performance results of three quadtree variations. The baseline model is included to highlight that in all cases performance is improved. Notice that both quadtrees using a partitioner aligned with the $x-z$ plane perform the worst once the spheres have settled. This can be attributed to the fact that when the spheres have settled the use of the $x-z$ plane as a partitioner will result in a large number of plane/object intersections. Such intersections increases the number of collision candidates in each partition.

Figure 8 presents performance results of various axis-aligned binary trees. In no case did these binary trees outperform the region-based octree. The most noticeable feature of this graph is the poor performance of the binary tree using partitioners aligned with the $x-z$ plane, while the use of $x-y$ and $y-z$ planes produces optimised performance.

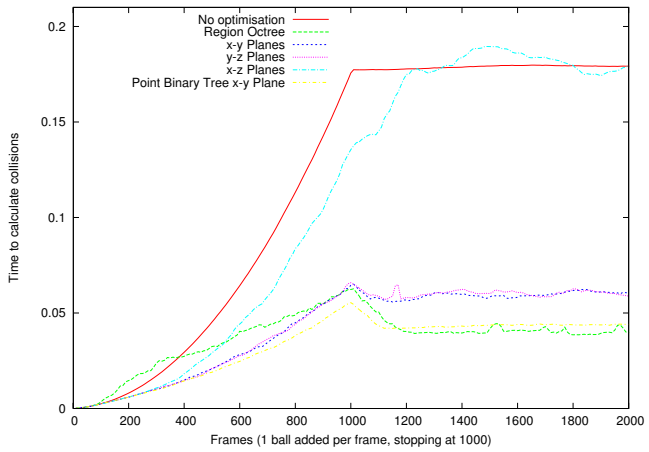

Figure 8: Total time of binary tree (bintree) with varying hyperplanes.

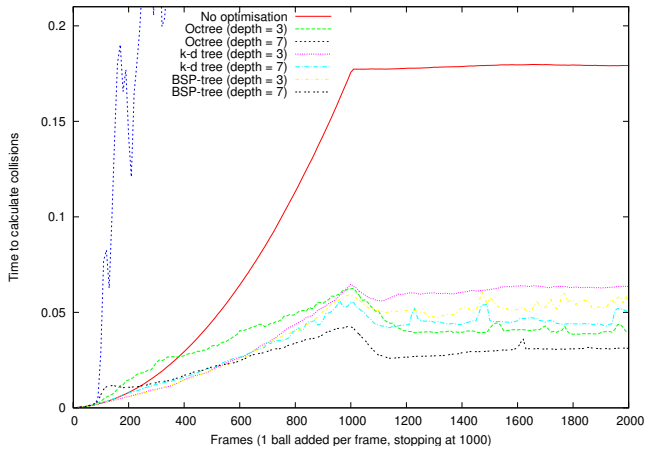

Figure 9: Comparison of schemes with increased depth.

3.3.2 Variation in Depth

The octree, *k*-d tree and BSP-tree are tested once again, but this time with a depth restriction of 7 instead of 3. Figure 9 presents the comparative results of each scheme with both depths. Strikingly noticeable is the extremely poor performance of the octree. This can be attributed to the high branching factor of the structure, which results in $\Sigma_{i=0}^{h} 8^i$ partitions, where h is the height of the tree, and in this case results in the order of 2 million partitions being generated per frame. Both the *k*-d tree and the BSP-tree, however, benefit from an increase in depth.

3.3.3 Reduction in Collision Tests

Figure 10 presents the average number of collisions tests per frame by each scheme, using a depth of 3. Notice that both variations of octree produce the least number of collision tests. Of particular interest is the behaviour of the region-based octree, which returns approximately 10 000 collision candidates once the spheres have settled. This indicates that for such a shallow depth, the octree produces very good results.

Figure 10 also highlights another phenomenon which has previously been ignored. There are a number of spikes which occur after the leveling-out of the spheres in the system, at approximately frame 1500, 1700 and 1950. These bumps are also noticeable in Figures 6 and 9. A number of observations can be made about these anomalies. Firstly, only some schemes seem to be affected by this anomaly. For example, the region-based octree and the BSP-tree are clearly affected, while the point-based octree is not. Secondly, in affected schemes, the bumps tend to occur at approximately the same frame. The fact that these bumps occur in approximately the same positions in Figure 6 (which is time-based) as well as in Figure 10 (which is not time based) rules out the possibility that such bumps can be attributed to operating system activity. Additionally, these anomalies cannot be entirely attributed to the coincidental configuration of spheres in the simulation, since all schemes would have been affected. The only remaining explanation is that the affected schemes are partially influenced by the sphere configuration, in the sense that at particular configuration, the partitioning schemes fail dramatically, resulting in many more collision tests than usual.

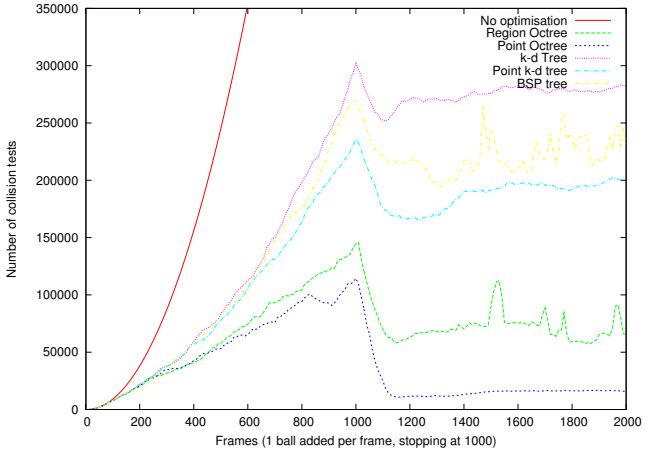

Figure 10: Total number of collision tests with $depth = 3$.

Figure 11 presents the number of collision tests resulting from the region- and point-based *k*-d tree and the BSP-tree with a maximum depth of 7. Notice how all of these schemes benefit substantially from an increased depth.

The major difference in Figure 11 from Figure 10 is the dramatic smoothing of the BSP-tree's performance after the settling point. This serves to highlight that the configuration of spheres which so radically affected the BSP-tree with depth 3 does not affect the same tree on depth 7. In this figure it is the *k*-d tree which suffers the most from arbitrary bumps after the settling mark. However, the average number of collisions is noticeably reduced from Figure 10. It is conjectured that further increasing the maximum depth of the *k*-d tree will smooth out these bumps in the same manner as with the BSP-tree.

3.3.4 Crossover in Performance

Figure 12 presents the running time of each of the most optimal schemes derived so far, but with an increased limit of 2000 spheres. The purpose of this test is determine which scheme works consistently better than the others, and in doing so identify under which circumstances different types of partitioning schemes are optimal. As noted previously the point-based and region-based octrees perform worse than the baseline model. However, the rate of increase in running time is not as great as the baseline, and thus there is soon a crossover, where these two structures perform better. This indicates that there is a minimum threshold concerning the number of objects, below which these optimisation schemes are ineffective.

The region-based *k*-d tree performs better than the baseline model and produces a consistent reduction in running time. However, once again there is a threshold after which this scheme fails. This can bee

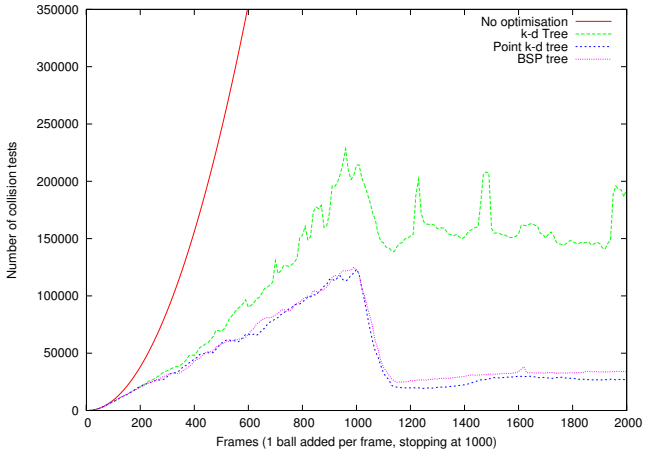

Figure 11: Total number of collision tests with $depth = 7$.

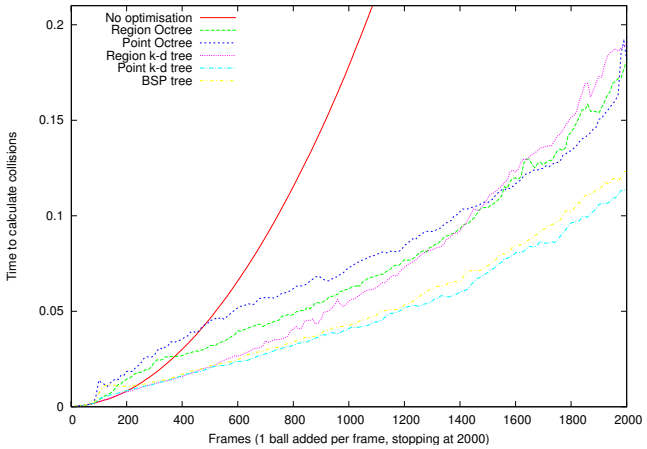

Figure 12: Crossover in performance

seen between frame 1400 and 1600, where the performance crosses over both octrees.

The best overall performing schemes are the BSP-tree and point-based *k*-d tree. Both these schemes remain consistently below the other schemes, and have smaller gradients in comparison. Whereas the gradient of both octrees and the region-based *k*-d trees appear to increase towards the last 300 frames, the gradients of the BSP-tree and *k*-d tree remain relatively linear. This indicates that it is unlikely that the runtime of these two schemes will ever cross over those of the other schemes.

3.4 Conclusions

A number of conclusions are inferred from the results presented in the previous section. The overriding conclusion of all the above tests is that all the optimisation schemes improve the performance of collision detection dramatically compared to the baseline model.

However, each scheme requires a different configuration in order to perform optimally. The configuration of each scheme depends on a number of factors. As identified in the experiments, these factors include allowable tree depth, the branching factor of the scheme, the choice of new partitioners for each level of recursion, and the application of the optimisation scheme. Incorrect choices for these factors can lead to results worse than when no optimisation is applied. Table 2 presents a summary of the observations made in the experiments with regards to these factors. Note that in the column indicating whether the scheme is affected by the application domain the specific factor which is affected is indicated in brackets. The conclusions made are as follows:

Allowable tree depth and branching factor The most noticeable difference in optimisation performance is a result of changing the allowable tree depth. For example, allowing a depth of 7 results in extremely poor performance in the case of the octree, but leads to the best performance in the case of the BSP-tree. A general rule can be applied as a result of this observation which states that an inverse relationship should be maintained between branching factor and depth. That is, the higher the branching factor, the lower the allowable tree depth should be. Different partitioning schemes also experience periods of uncharacteristic inefficiency, which can only be explained by temporary failures in the partitioning scheme to handle certain configurations of objects in the simulation. Methods exist, however, which can eliminate these periods such as increasing the allowable tree depth.

Choice of new partitioners These experiments test two different methods of selecting new partitioners, region-based and point based. Even though the choice of a point based partitioner in these experiments is very simplistic, the performance of a number of algorithms including the *k*-d tree are greatly improved. In other situations, very little difference is exhibited. Therefore the choice of partitioner is dependent on the application.

Application of the partitioning scheme The amount of optimisation capable of being achieved is highly dependent on the application to which it is being applied. A clear example of this is in Figure 8, where dramatically poor performance is achieved after choosing an optimisation scheme which is not suited to the general motion of the objects in the simulation.

In terms of the initial goals of the experiments the following conclusions are drawn:

- The scheme which provides the greatest increase in speed, and the greatest reduction in the number of collision candidates is the point-based *k*-d tree, limited to a depth of 7. Closely following is the BSP at level 7. However, one cannot exclude the octree which performs remarkably well at a very shallow depth.
- Different schemes respond differently to change in depth, with the performance exponentially proportional to the branching factor of the structure.
- Each scheme reduces the number of collision tests dramatically from the baseline method, but large differences occur between different optimisations.
- The BSP-tree and point-based *k*-d tree perform consistently well as objects are added to the simulation, compared to the other schemes, and it is unlikely that these schemes will ever perform worse than the any of other schemes.

4 Conclusion and Future Work

Collision detection is a well studied area in dynamic simulation, robotics and many other fields. This paper introduces collision detection in terms of dynamic simulations, and describes a number of broad-phase techniques which can be used to improve the performance such a system. In addition a number of experiments are done in order to evaluate the comparative efficiency of a subset of the described schemes, specifically tree-based partitioning schemes. A generic structure for implementing these tree-based partitioning

	Improved by Increasing Depth	Improved by Point-based Partitioners	Affected by Application Domain
Octree	No	No - but number of collision-tests very low and stable where point-based scheme used.	No
QuadTree	-	-	Yes (choice of partitioner)
BinTree	-	Yes	Yes (choice of partitioner)
k-d tree	Yes	Yes	Yes (anomalies in number of collision-tests)
BSP-tree	Yes	N/A	Yes (anomalies in number of collision-tests)

Table 2: Summary of observations.

schemes, and testing them in a neutral manner, is created. These tests highlight a number of important relationships which are important when choosing a partitioning scheme for collision detection. Overall, every partitioning scheme tested provides some performance improvements over the baseline model.

This paper contributes to the relatively small body of work of which the focus is the analysis of well known partitioning techniques for optimising collision detection. Well known techniques such as the octree, *k*-d tree and BSP-tree are evaluated and implemented using a generic structure, and a common testing simulation. This generic structure allows for the testing of derivatives of each structure, such as point-based trees, as well as trees which are reduced to lower dimensions (for example, the quadtree).

There is great potential for future work in this area. Each of the above tests can be run in a range of different simulations (where the object dynamics are different) in order to establish a relationship between the performance of an optimisation and the application domain. Additionally, the generic structure can be enhanced to support temporal coherence between frames. This will establish whether there is a performance gain in implementing temporal coherence, or whether the overhead in implementing such a scheme outweighs the benefits. Finally, similar tests can be run on narrow-phase optimisations in order to establish which works the best for different types of objects.

References

BANDI, S., AND THALMANN, D. 1995. An adaptive spatial subdivision of the object space for fast collision detection of animating rigid bodies. In *Proceedings of Eurographics '95*, 259–270.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9, 509–517.

BUSS, S. R. 2003. *3-D Computer Graphics*. Cambridge University Press, Cambridge, United Kingdom.

EBERLY, D. H., AND SCHNEIDER, P. J. 2003. *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers, San Fransisco, USA.

GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1996. OBBTree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 171–180.

HAVRAN, V., PRIKRYL, J., AND PURGATHOFER, W. 2000. Statistical comparison of ray-shooting efficiency schemes. Tech. rep., Czech Technical University, Czech Republic.

HELD, M., KLOSOWSKI, J. T., AND MITCHELL, J. S. B. 1995. Evaluation of collision detection methods for virtual reality flythroughs. In *Proceedings of the Seventh Canadian Conference on Computational Geometry*, 205–210.

HUBBARD, P. M. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 3, 179–210.

JAMES, D. L., AND PAI, D. K. 2004. BD-tree: output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics 23*, 393–398.

KLOSOWSKI, J. T., HELD, M., MITCHELL, J. S. B., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (January), 21–36.

LEVEY, E., PETERS, C., AND O'SULLIVAN, C. 2000. New metrics for evaluation of collision detection techniques. In *International Conference in Central Europe on Computer Graphics and Visualization*.

LIN, M. C., AND GOTTSCHALK, S. 1998. Collision detection between geometric models: a survey. In *IMA Conference on Mathematics of Surfaces*, 37–56.

LIN, M. C. 1999. Fast and accurate collision detection for virtual environments. In *Dagstuhl '97, Scientific Visualization*, IEEE Computer Society, Washington, DC, USA, 171–180.

LUQUE, R. G., COMBA, J. L. D., AND FREITAS, C. M. D. S. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, 179–186.

MIRTICH, B. 1998. Efficient algorithms for two-phase collision detection. *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, 203–223.

MOORE, M., AND WILHELMS, J. 1988. Collision detection and response for computer animation. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 289–298.

NAYLOR, B. F. 1992. Interactive solid geometry via partitioning trees. In *Proceedings of the conference on Graphics interface '92*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 11–18.

SAMET, H. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys 16*, 2 (June), 187–260.

TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'03*, 47–54.

VANECEK, G. 1991. BRep-index: a multidimensional space partitioning tree. In *SMA '91: Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*, ACM Press, New York, NY, USA, 35–44.