An evaluation of grid based broad phase collision detection for real time interactive environments

Jonas Liljeby
June 2011

Bachelor Thesis, 15 credits, C
Computer Science

Dept of computer science, University of Gävle
Examinator: Peter Jenke
Supervisor: Stefan Seipel
An evaluation of grid based broad phase collision detection for real time interactive environments.

By
Jonas Liljeby
Dept of computer science
University of Gävle
S-801 76 Gävle, Sweden

Email:
pf01jly@student.hig.se

Abstract

Detailed and exact collision detection for large amounts of objects has for a long time been a non real-time affair because of the immense amount of computations necessary. This was however not only because of the complexity of the algorithms but also because discussed of the computations would not have had to be done in the first place. This paper has through literature research and empirical testing examined two different broad phase approaches to object culling in a three dimensional environment. The aim of such a broad phase algorithm is to decrease the amount of computation heavy narrow phase collision detection checks and thus enhancing application performance. Potential weaknesses of these approaches were addressed and possible solutions discussed. Performance comparisons were made to give a better overview of what kind of performance enhancements can be expected and to give a theoretical base for further research.

Keywords: Collision detection, object culling, broad phase, octree, hierarchical grid
# Contents

1 Introduction........................................................................................................................................1
   1.1 Background ..................................................................................................................................1
   1.2 Research focus ............................................................................................................................1
      1.2.1 Delimitations ......................................................................................................................2
   1.3 Research goal .............................................................................................................................2
      1.3.1 Research problems ..............................................................................................................2
   1.4 Related work ................................................................................................................................2

2 Methods .............................................................................................................................................3
   2.1 Descriptions of tested algorithms .................................................................................................3
      2.1.1 The naive base case ............................................................................................................3
      2.1.2 Fixed depth octtree .............................................................................................................4
      2.1.3 Fixed depth hierarchical grid ...............................................................................................6
   2.2 Tests ................................................................................................................................................8
      2.2.1 Test 1 .....................................................................................................................................10
      2.2.2 Test 2 .....................................................................................................................................10
      2.2.3 Test 3 .....................................................................................................................................10
      2.2.4 Test 4 .....................................................................................................................................11
      2.2.5 Test 5 .....................................................................................................................................11
      2.2.6 Test 6 .....................................................................................................................................11

3 Results ................................................................................................................................................12
   3.1 Literature research results ...........................................................................................................12
   3.2 Test results .....................................................................................................................................12
      3.2.1 Test 1 results .......................................................................................................................12
      3.2.2 Test 2 results .......................................................................................................................13
      3.2.3 Test 3 results .......................................................................................................................14
      3.2.4 Test 4 results .......................................................................................................................15
      3.2.5 Test 5 results .......................................................................................................................16
      3.2.6 Test 6 results .......................................................................................................................17

4 Discussion ........................................................................................................................................18
   4.1 General weaknesses ......................................................................................................................18
   4.2 Memory usage ..............................................................................................................................18
   4.3 Collision detection .......................................................................................................................19
   4.4 Updating objects .........................................................................................................................20

5 Conclusion .......................................................................................................................................21

6 Acknowledgements ..........................................................................................................................22

References .............................................................................................................................................23
1 Introduction

Collision detection (CD) is an intricate and vital component in many types of computer applications today, for example robotic engineering, virtual simulations and visual/immersive entertainment. Especially in virtual simulations focusing on an believable immersive experience, the use of collision detection can greatly reduce the need for otherwise scripted events. Objects are given the opportunity to realistically interact with each other or even the user in more ways than any programmer would even think about trying to predict.

1.1 Background

CD is a popular expression today, but it is often misinterpreted as being the whole process of interaction between objects in an environment. This is however not the case, CD is only one part of that process. "Collision detection aims at gathering information concerning the collisions that take place in the environment" [1]. The second part of the process is called Collision determination (CDE) [2] and determines the actual intersections between objects. The last part of the process is called Collision response [2] and deals with how the objects colliding will respond. Together CD, CDE and CR make up what is called Collision handling (CH).

CH becomes increasingly labor expensive to compute with increasing complexity of the geometry of the objects that are colliding. Collisions between simple geometrical shapes like spheres or cubes can be computed using relatively simple computations while more complicated shapes may require more complex and labor expensive computations.

One performance bottleneck of CD lies in that the base case pairwise tests between objects is often considered to be of O(n²) complexity [3]. What this means is that in a scene with 100 objects, object A must do a CD check against all the 99 other objects in the scene, and every other object in the scene will have to do the same thing to every other object but itself. With increasing numbers of objects present in the scene, this quickly becomes very labor expensive to compute. It is possible to lower the detail of CH to lessen impact on application performance but the result will also be less accurate and this may not be a viable option in applications that require both speed and accuracy.

To prevent the deterioration of application performance in environments handling a large amount of objects, some propose that CD is carried out in stages. "Hence, collision detection methods must ensure that a quadratic asymptotic behavior will not occur. To achieve this, most methods are carried out in two or three phases." [1] The two main phases that CD usually is divided into are often called broad phase and narrow phase [4]. Broad phase focusing on using fast algorithms to make a rough approximation to what objects can be considered unlikely to interact and what objects need a more detailed test. Objects that are not rejected as unlikely to intersect in the broad phase are passed on to the narrow phase that uses a more detailed approach to see if the objects intersect.

1.2 Research focus

This paper is aimed at evaluating the use of two different broad phase approaches for real-time virtual simulation of a large dynamic three-dimensional environment. The approaches chosen for evaluation are of the types octree [5] and hierarchical grid [5].

Simplified it can be said that the chosen broad phase approaches are based on subdividing the environment into several smaller sections forming a grid [6] and through fast simple computations assigning objects in the environment to one or more of the grid sections. Objects will later only have to perform narrow phase CD against other objects present in the same sections of the grid in hope of lessening the influence of the typical O(n²) base case.

The environment that is meant to be simulated is one that is similar to the one we live in here on earth. It should therefor be capable of containing a massive amount of objects capable of CH, the objects can be of widely different sizes and many of them will not be statically fixed in one location. Large areas of the simulation should mostly contain very few interactive objects much like our skies while some areas like for example streets and cities could often be populated by large amounts of both moving and static objects. It is
however important to stress that these areas do not always stay the same, for example an explosion could throw large amounts of objects in the form of debris up into the air or an event can draw population and activity to an area that usually would see little activity.

1.2.1 Delimitations

This paper does not go into detail concerning narrow phase approaches, this is considered to be beyond the scope of this paper. Basic information provided on this subject is merely meant to aid in the understanding of the discussed broad phase approaches.

1.3 Research Goal

For the purpose of this paper the environment will be considered a hypothetical base for a three dimensional computer game. The gaming industry is not only a multi billion US$ business but also a large driving force behind development of technology and techniques aimed at massive real-time computations. "Interactive entertainment has long been one of the driving factors behind architectural innovation, pushing the boundaries of computing to achieve ever more realistic virtual experiences" [7]. Game engines have been implementing more and more complex physical simulations using large numbers of objects in recent years. With growing expectations on the realism of computer games, techniques for handling ever bigger and more interactive environments are in high demand.

With the evaluation of these broad-phase algorithms this paper hopes to shed some light on their weaknesses and potential usefulness in the ever evolving and more demanding computer game industry.

1.3.1 Research problems

• Do broad phase algorithms have any specific weaknesses that should be considered when using these approaches?
• Should memory usage be considered a vital factor in choosing between broad phase algorithms?
• From a runtime performance standpoint for different amounts of objects, how do the chosen broad phase approaches compare to each other and to not using any broad phase at all?

1.4 Related work

P.M Hubbard [8] early on wrote about the advantages of dividing collision detection into phases and using simple bounding volumes to speed up runtime performance. Today there are several different major types of broad phase approaches but the ones concerning this paper the most are the Octree and Hierarchical grid based types. Z. Chengming [9] describes an Octree algorithm with a treenode that has 8 branch nodes that in turn has branch nodes of their own. Every branch relates to a subdivision of the space that its parent node relates to. By dividing the environment into smaller pieces it is possible to perform intersection tests only between objects that are in the direct vicinity of each other and thus avoiding a lot of unnecessary calculations. An archetypical hierarchical grid is described by M. Eitz [6] where several grids in different resolutions are superimposed on the environment. Objects are assigned to the different grids depending on object size and later within the grids into a grid section depending on object coordinates.
2 Methods

For this paper tests have been performed for the relevant broad phase approaches in comparison to a naive base case.

2.1 Descriptions of tested algorithms

Three different cases were implemented for creating test results for the broad phase algorithms. First a naive base case was created, this was not created as a broad phase algorithm and instead was to represent collision detection not using a broad phase algorithm and to serve as mere comparison. The second algorithm was an octree, a popular and relatively easily implemented broad phase approach. The third and last algorithm was a hierarchical grid, a multi level grid algorithm using a hash function.

2.1.1 The naive case

The naive case algorithm stores one object pointer to the first object in a linked list. Every object in turn has a pointer to the next object in the list. It is important to understand that the linked list only stores pointers and not the actual objects, objects are stored separately.

Object position and size have no meaning for where in the linked list an object is inserted, this happens by chance and has no meaning for the execution of the naive base case algorithm. In figure 2.1 the basic workings of the base case can be observed, illustrating how the first object in the list performs collision detection against all other objects in the list and thus also the environment.

Object pointers are at insertion added to the front of the list, this can be done in constant time thanks to the simple linked list structure.

Removal of object pointers requires the list to be traversed until the relevant object is found, it can then be removed by simple pointer bypassing.

Collision detection is performed by completely traversing the linked list, making collision detection tests against all other objects but the object that is being tested as described in Figure 2.2.

![figure 2.1. An overview of the naive case.](image)

Figure 2.2 Pairwise naive collision detection pseudocode

Object X = First object in linked list
Repeat while X is not NULL
{

}
Object $Y = \text{First object in linked list}$

Repeat while $Y$ is not NULL
{
    if $X$ is not $Y$
    {
        $X$ tests for intersection with $Y$
    }
    $Y = \text{Next object in linked list}$
}

$X = \text{Next object in linked list}$

---

2.1.2 Fixed depth octree

An octree is a structure that in every octree node divides an environment down the middle in all three axis, this creates 8 new smaller octree nodes. The first level slices the environment into 8 smaller pieces, in the second level everyone of the 8 nodes created by the first level creates 8 more nodes of their own. Inspiration and basic theory behind this approach was derived from the work of Z. Chengming [9]. It is however important to note that the following description is of the authors own interpretation of an octree and may not be equivalent to earlier work.

In this particular algorithm the octree has a fixed depth of 4 levels.

This version of an octree can store object pointers not only in the leafs but also in the nodes themselves, this makes it possible for object pointers to be inserted anywhere in the tree. Linked lists are used for storage in the octree nodes, every node stores a pointer to an object pointer and that object stores a pointer to the next object in the list and so forth. By using such a simple linked list memory usage is kept lower than if using a complex structure for all pointers. Object pointers can also be inserted very quickly at the front of the linked list at constant time.

When an object pointer is inserted it is first tested at the top level of the octree, if the object bounding sphere can fully be inserted into a lower level it will be passed on to that level. This continues throughout the tree until the bounding sphere can either not be fully fitted in a lower level or the object reaches a leaf where it in that case is inserted. This means that even if the insertion into the linked list can be done in constant time there will still be some costly traversal of the octree nodes that prevents the full tree insertion from being carried out in constant time.

If the objects bounding sphere cannot be fully inserted into a single lower level but also infringes on the space of another octree node then the object will not be passed down to that level and its pointer will be inserted on the current level instead.

Figure 2.3 presents a visual example of how a quadtree divides the environment like explained above. The reason a quadtree was chosen for illustrating this is purely because illustrating the problem is simpler in two dimensions. A quadtree works much in the same way as an octree with the difference being that a quadtree has 4 childnodes and works in two dimensions while an octree has 8 child nodes and works in three dimensions. In the figures it can be seen that objects that can't be fully inserted into any of the first level divided surfaces are inserted in the top level quadtree node. In the next level only the first node has divided its surface into 4 new surfaces to make the figure simpler. Much in the same way as the level before it, objects that can be fully fitted into the new smaller surfaces are passed down one level while objects that cannot are inserted at the current level node.
Removal of an object is handled much the same way as insertion, tracing the object the same way down the tree before, using its bounding sphere coordinates and size. Here the linked list that stores object pointers however needs to be traversed to find the object that is to be removed before removal from the link list can be performed. Removal is done by a simple bypass of the affected object.

When using the octree for collision detection an objects bounding sphere will first be compared with all the objects on the top level of the tree. If the object is not found at the current node it will just as with insertion and removal pass the object down by testing for if its bounding sphere can fit entirely into a single lower nodes space without infringing on other nodes on that level. If it fits in one lower node it will be passed on down that side of the tree only, testing against all objects present in the nodes it passes. If it however is found to infringe on the space of one or more additional nodes it will be passed down to be tested against all objects in all children nodes on the next level as described in figure 2.4.

Figure 2.4 Pairwise octree collision detection pseudocode

Object X = Object that is to be tested against octtree structure
Object Y = First object in linked list
Repeat while Y is not NULL
{ 
    if X is not Y
    {
        X tests for intersection with Y
    } 
    else
    { 
        X was found on this level
    }
2.1.3 Fixed level hierarchical grid

A hierarchical grid structure works by having two or more grids in different resolutions covering the scene. Inspiration and basic theory behind this approach was derived from the work of M. Eitz et al [6]. It is however important to note that the following text describes the authors own take on a hierarchical grid and may not be equivalent to earlier implementations.

This particular version starts out with a coarse grid dividing the scene into 8 sections, the next level grid has a resolution of 64 sections and so forth. This is not to be confused with an octree, here every grid covers the entire simulated environment and not just smaller parts like lower levels of an octree.

There is at any single time only one hierarchical grid node and it holds the pointers to all linked lists for all the sections of all grids. This is achieved by using an array of object pointers, every pointer pointing at the first object in a linked list. Every position in the array represents a grid section in one of the grids. In this case the first 8 positions belong to the first grid, the next 64 positions belong to the next grid and so forth. Insertion into the hierarchical grid is achieved by catching objects in different grids depending on the objects bounding spheres diameter and the grids sections size. This algorithm has defined that a bounding sphere may not have a diameter larger than one quarter of a sections side, if the bounding sphere is larger it is passed to the next grid with larger sections. It is possible to set different tolerances for when a bounding sphere will be caught but the bounding spheres radius may not be larger than half the section size, this is important for the collision testing and is described in the collision detection part further down. To decide in what section and array number an object pointer is to be stored a spatial hash function [10] that uses bounding sphere coordinates and the level at which it is to be inserted is called.

Figure 2.5 illustrates how this is achieved in two dimensions. In two dimensions the four first positions in the array are responsible for storing the object pointers related to the grid with the lowest resolution and the 16 next positions are related to the next level grid with a higher resolution instead of 8 and 64 respectively for the three dimensional version.
The particular hash function used for this implementation uses a somewhat complicated and computationally heavy approach but yields very accurate results. This makes it possible to link every grid section to a particular hash number, minimizing unnecessary collision tests. The hash function takes input regarding the objects position, what level grid it should be inserted into and the size of the grid sections in that grid. By dividing the X, Y and Z position values by the size of the grid sections an indication of what grid section an object belongs in can be produced. The level number is used to offset the resulting position number so that the numbers produced by objects in the most coarse grid will occupy position 0 to 7 and objects in the next, finer grid will occupy numbers 8 to 63 and so forth. A less computationally heavy but less accurate alternative to this type of hash function could be the XOR hash function used by M. Teschner et al [11].

Removing an object from this hierarchical grid is fast and easy thanks to an internal section number that all objects are awarded at insertion. An object thus already knows in what part of the array it is located and only the objects present in the same section will need to be traversed. It is possible to also search for an object using the objects internal section number, in that case the spatial hash function is used.

Collision detection in a hierarchical grid is achieved through comparing the objects bounding sphere with the other objects bounding spheres in the different sections of the grid. Unlike the octtree, objects in a hierarchical grid are inserted in the section where their bounding spheres center point is, they may very well overlap into other sections. This is the reason why an object sphere radius may not be larger than half of the section size of the grid it is caught in. Objects are only tested against the objects in the same section and the objects in directly adjacent sections. There is a chance that if bounding spheres have a radius larger than half the section size they may come into contact with bounding spheres that are further away than the directly adjacent sections and then collisions that should occur will not. To decide what sections an object should be compared to a spatial probing by hash function is done in all directions depending on the bounding sphere radius and the section size. This same procedure is done for all grids, the probing depth varying with the resolution of the affected grid to assert that all relevant sections are tested against as described in figure 2.6.

Object $X = \text{Object that is to be tested against hierarchical grid structure}$
For all grid levels do
{
    \( Q = \text{hashfunction}(X, \text{gridlevel}, \text{grid section size}) \)
    \( X \) does naive collision detection against grid section \( Q \)'s linked list
    \( \text{Probedistance} = \text{radius of bounding sphere} + \text{grid section size} \times \text{object to grid ratio} \)
    for all directions
    {
        \( Y = \text{hashfunction}(X + \text{probedistance}, \text{gridlevel}, \text{grid section size}) \)
        \( X \) does naive collision detection against grid section \( Y \)'s linked list
    }
}

### 2.2 Tests

Even if it is possible to make a theoretical assessment of O-notation for different operations on the discussed broad phase algorithms it is still very hard to predict actual real time performance. For this reason synthetic benchmarks that measure performance have been created.

The benchmarks have been written in C/C++ and were during the tests running on a PC equipped with 2 Gigabytes of RAM and a Intel Q9450 CPU running at 2.67 Gigahertz.

All benchmarks started by creating a simulated scene that is contained in a cube with every side representing a simulated distance of one kilometer. Inside the cube various numbers of objects with bounding spheres [5] with a diameter varying between 1/10 and 1/1000 the length of the cube's side are present. Bounding sphere sizes were randomly chosen as to simulate different size objects by an even dispersion random function.

In these tests objects would interact solely based on their bounding spheres, shapes, and sizes. In computer games it is common to rely on this type of simple collision detection depending on bounding spheres or axis aligned bounding boxes (AABB) [5]. For more demanding applications a simple collision detection between the bounding spheres and/or AABB can be made to trigger more complicated collision detection between the objects contained within (see figure 2.7).

![Figure 2.7. Bounding sphere collision detection triggering more detailed collision between object geometry.](image)
From this point on this paper mentions both objects and bounding spheres, to avoid confusion every object should be considered to have a bounding sphere and every bounding sphere should be considered to belong to an object.

![Figure 2.8. An uneven distribution of objects throughout the environment.](image1)

Initial object positioning was random but contained in boundaries created within the environmental cube. In this way more objects can be assigned to parts of the cube representing areas with a high density of objects. This makes it possible to easily represent a larger density of objects at ground level than far up in the air (see figure 2.8) or for example a more even distribution throughout the scene (see figure 2.9).

![Figure 2.9. An even distribution of objects throughout the environment.](image2)

Among the objects, 20% were not statically fixed in place and had a random movement direction. Only environmental objects that were not statically fixed performed bounding sphere collision detection against other moving and non-moving environmental objects.

To represent a slight narrow-phase burden for more realistic results in the tests a simple mimic of a real narrow-phase has been implemented. This allowed for the objects to interact with each other, changing their movement directions as they collided. In a full CH application the objects inside the bounding sphere’s
could perform narrow phase collision detection against each other in a more complex manner. For this evaluation however simple narrow phase calculations like for Spheres or Cubes is more appropriate since this is what is usually used in most computer games.

Objects that were initially statically fixed in the scene would not start moving if they were hit by a moving object, this was meant to prevent an increase in moving objects during runtime that could have given uneven test results.

Collision tests between bounding spheres were made and object movement was updated for every frame, not depending on a timer. In a real life application it may be desirable to not update with every frame, either for performance or application runspeed reasons. For these tests this was however not a problem since it’s just a benchmark without demands on user interaction.

Multiple tests were ran involving varying amounts of objects present in the environment. This was done to create an overview of broad phase behavior and performance at increasing amounts of objects. Tests were done using between 50 and 6400 objects in two different types of environments. The first environment represented objects being evenly distributed around the environment and second environment represented objects being much more predominant in the lower 10% of the environment.

All tests were ran for algorithms of the naive base case, a fixed depth octree and a fixed level hierarchical grid. The naive base case uses no broad phase algorithm and naively makes narrow phase comparisons between all other objects in the scene.

Because the execution times for single bounding sphere testing in test 1 and 2 often was very short, the software had trouble accurately measuring it and tests were done in the following fashion. All moving bounding spheres in the environment were tested for bounding sphere collision detection, sometimes more than once if the amount of moving bounding spheres were low. The overall measured time was then divided by the amount of bounding spheres tests made to calculate an average collision detection time per bounding sphere.

To find an average number of tests between bounding spheres in test 3 and 4, all moving bounding sphere tests were counted. The resulting number of tests were divided by the amount of moving bounding spheres to find an average number of narrow phase tests per bounding sphere.

One problem resulting from the nature of inserting and removing objects in tests 5 and 6 was that if the objects were actually inserted or removed, the amount of objects in the environment would also change. This could potentially affect the results because of the tests. To prevent this special pseudo-insertion/removal functions were created for the different broad phase algorithms and the naive case. These functions worked much like the normal insertion and removal functions, calculating positions and traversing the structures but with the exception that it did not actually make the actual final insertion/removal and thus did not increase the amount of objects.

### 2.2.1 Test 1

The first test examined the average time taken for one bounding sphere to perform bounding sphere collision detection against the other bounding spheres in the environment.

The dispersion of bounding spheres throughout the scene was 80% of the bounding spheres inhabiting the lower 10% of the y-axis and the residual 20% of bounding spheres were randomly placed throughout the entire scene. A visual demonstration of this level of dispersion can be seen in figure 2.8.

### 2.2.2 Test 2

The second test, much in the same manner as test 1, examined the average time taken for one bounding sphere to perform bounding sphere collision detection against the other bounding spheres in the environment.

The dispersion of bounding spheres throughout the scene was in contrast to test 1 100% of all bounding spheres randomly placed throughout the entire scene as demonstrated in figure 2.9.

### 2.2.3 Test 3

In the third test the objective was to find the average amount of narrow phase collision detection tests a
bounding sphere needs to make when performing bounding sphere collision detection against other bounding spheres in the environment.

The dispersion of bounding spheres throughout the scene was 80% of the bounding spheres inhabiting the lower 10% of the y-axis and the residual 20% of bounding spheres were randomly placed throughout the entire scene.

2.2.4 Test 4

In the fourth test the objective was to find the average amount of narrow phase collision detection tests a bounding sphere needs to make when performing bounding sphere collision detection against other bounding spheres in the environment.

The dispersion of objects throughout the scene was 100% of bounding spheres randomly placed throughout the entire scene.

2.2.5 Test 5

The aim of this test was to measure the average object insertion time in environments with certain amounts of objects already present.

The dispersion of objects throughout the scene was 100% of objects randomly placed throughout the entire scene.

Much like in earlier tests there was a problem with accurately measuring the time one object at the time. For this reason this test also measured a multitude of different insertions and divided the resulting time by the amount of objects inserted.

2.2.6 Test 6

Finally test number six tested the average object removal time in environments with certain amounts of objects present.

The dispersion of objects throughout the scene was 100% of objects randomly placed throughout the entire scene.

Much like in earlier tests there was a problem with accurately measuring the time one object at the time. For this reason this test also measured a multitude of different removals and divided the resulting time by the amount of objects removed.
3 Results

In the results part of this paper both the results from literature research as well as the actual running tests of the broad phase algorithms are summed up and analyzed.

3.1 Literature research results

It was found that F. Liu [12] holds that the use of these types of grid type spatial subdivision is highly effective as long as the objects are of roughly the same size. If not, a more sophisticated algorithm must be created to make the most of the subdivided spaces. If objects are too big they can be present in too many subdivision spaces at the same time and if they are too small they may be too numerous in one space for the algorithm to have a positive effect on application performance.

Luque [4] holds that this type of spatial subdivision also can cause problems in highly dynamic scenes because of the typically rigid structure used in many such techniques. The efficiency of the algorithm is partially dependent on the dispersion of objects between the different subdivisions of the space. In the case dispersion becomes increasingly uneven, performance may lessen.

According to R. Lau [13] the use of an AABB [14] in combination with these types of spatial subdivisions can result in the need to constantly recompute the AABB even if the contained object only is performing rigid transformations like rotation. An AABB or Axis Aligned Bounding Box is a bounding volume much like the bounding sphere described earlier in this paper. An AABB however has a boxlike shape that is always aligned with the axes of the environment and cannot be rotated. Much like the bounding spheres the AABB should be as small as possible while still containing the related object. If an object is formed in such a way that it needs different size boxes depending on its rotational degree the AABB is forced to comply. Not only does this mean recomputing the AABB but in the case of an octree it can mean that an object may have to be moved back and forth between different parts of the tree depending on what regions the AABB intrudes on which may prove a larger time waste than recomputing the AABB itself.

C. Ericson brings up the issue of collision tunneling [5] where in a discrete collision handling system [15] an object may pass straight through another object without registering a collision. This may happen in the case that the interval between collision tests is so great or the speeds of the objects so high that one or more objects move far enough to pass through each other before a collision test is issued between the objects.

Even though broad phase spatial subdivision can decrease the amount of objects to be compared by a significant amount, the typical approach within the subdivisions still seemed to be the naive O(n^2) solution of comparing every single object to all others. Little actual research was found discussing alternatives to this.

3.2 Test results

3.2.1 Test 1

First the average collision detection time for an object in an environment with a an uneven distribution of objects was addressed. Here it was discovered that for very low amounts of objects in the environment, both the naive case and octree are significantly faster than the hierarchical grid algorithms (see figure 3.1).

The octree algorithm showed high performance at low amounts of objects and outperformed both the naive case and the hierarchical grid in the 50 to 400 objects interval. The hierarchical grid algorithm appeared not to be as greatly influenced by an increase in environmental objects and equaled the performance of the naive and octree algorithms at 100 and just above 400 objects respectively.
In a larger scope (see figure 3.2) it could be seen that the hierarchical grid executes significantly faster than the naive and octree implementations with very large amounts of objects present in the environment. The octree however still showed a large performance improvement over the naive base case.

3.2.2 Test 2

In the case where objects are evenly distributed over the environment the naive algorithm showed no significant difference in performance (see figure 3.3) compared to the results from Test 1. The octree however started to show a very slight decrease in performance when the amount of environmental objects closed in on 400 while the hierarchical grid showed a slight increase in performance over the test results from Test 1.
Viewed in a larger scope (see figure 3.3) it became apparent that as the number of environmental objects rise to large numbers the performance of the octree dropped slightly compared to test results from Test 1 while the performance of the hierarchical grid increased by a significant degree.

The performance gap between the hierarchical grid and octree at very large amount of environmental objects became more pronounced than in the results from Test 1 while the gap between the base case and octree lessened by a small degree.

### 3.2.3 Test 3

When it came to the average amount of narrow phase tests being made per object that performs collision detection against other objects, the naive case showed an overall high amount of tests compared to both the octree and hierarchical grid algorithms, even at low numbers of environmental objects (see figure 3.5).
The octree presented a big improvement over the naive base case but the hierarchical grid performed even better testing close to half of the objects the octree did over the entire test range.

This relationship held true also for higher numbers of objects in the environment with the hgrid testing just above half the amount of objects the octree tested and roughly 1/6th of the amount of tests the naive case performed (see figure 3.6).

3.2.4 Test 4

When objects were distributed evenly over the scene the naive case performed identically to the results in test 3. The octree however showed a slight increase in tests performed and the hierarchical grid showed a slight decrease in tests performed (see figure 3.7) compared to the results of test 3.
For larger amounts of environmental objects (see figure 3.8) it became more apparent that the octree algorithm indeed was showing a tendency towards more object tests than in the results produced by test 3.

The amount of naive object tests performed by the naive case however again presented identical numbers and the hierachical grid showed an even greater reduction in objects tested at the higher environmental object range compared to test 3.

3.2.5 Test 5

The test for evaluating the insertion time on an object into a structure displayed little influence from the amount of objects already present in the environment. The naive case displayed the fastest insertion time followed by the octree algorithm. The hierarchical grid was substantially slower at inserting objects compared to the other two (see figure 3.9).

Take notice that insertion and removal times are measured in microseconds because of their fast execution times while other tests in this paper are using milliseconds as measurement.
3.2.6 Test 6

In the removal of objects test it was found that the amount of environmental objects had a large impact on performance on the naive and octree algorithms (see figure 3.10). The hierarchical grid algorithm however performed exceptionally well in this test and even though its performance degrades somewhat at high amounts of environmental objects it is still many times faster than the other tested algorithms.

Figure 3.9. Average insertion time for one object

Figure 3.10. Average removal time for one object
4 Discussion

In the discussion part of this paper literature research, results from the runtime test and personal experiences are summed up and discussed. The main goal being to shed light on the broad phase algorithms most common weaknesses, their memory usage and runtime performance.

4.1 General weaknesses

There are indeed some problems with using these types of spatial subdivision algorithms as found in the literature research. Most of these problems can however be overcome by implementing support algorithms that solve or lessen these problems significantly.

It is possible to solve the dynamic environments problem Luque [4] brought to light by further subdividing areas as they are increasingly populated by objects. This is however a costly operation that theoretically could make a noticeable hit on application performance as it is performed. Even if average performance may be higher using a well made dynamic implementations the extra calculations during reconstruction make performance less dependable. Stuttering performance in a real-time interactive application is highly undesirable and may in a computer game greatly sacrifice the players sense of presence [16]. Peak performance may in such cases be traded for a slower but more stable execution. If the environment and object sizes are already know it can be argued that using a fixed depth structure will see less overhead computations at the expense of extra memory usage. The problem Fuchang brought up considering different size objects [2] is also related to this but would also mostly be solved by using a dynamic or cleverly precalculated fixed depth algorithm.

The use of the naive $O(n^2)$ algorithm for comparisons between objects within subdivided spaces may be a minor problem as long as the number or objects are low. However, it takes little pessimism to claim the efficiency of the subdivision algorithm would only in the best of worlds be so high that the amount of objects at all times stay low. It is considerably difficult to make a purely theoretical assessment of the performance gains or losses of making a non naive algorithm, further research in this field may be on interest.

In the algorithms used for this paper bounding spheres have exclusively been used but there may be cases when axis aligned bounding boxes are to be used instead. As R.Lau [13] notes there is something of a problem that this can present since an AABB may in contrast to a bounding sphere intrude on different parts of the scene depending on rotational degrees. This does not present a problem for a hierarchical grid since an object is not inserted into a section depending on the overlap of the bounding volume but the center position. In the case of an octree this may however become a problem since the AABB may at different rotational orientations put the object in different parts of the tree. What this leaves is that a simple stationary but rotary object may force constant costly position updates in the case that an octree is used with rotating AABB. This should be considered in applications where such situations may occur.

Collision tunneling [5] may be a serious problem in an environment with fast moving objects if collision detection is not performed at close enough intervals. This may be a problem in environments containing large amounts of objects since this makes collision detection slow down considerably. If collision detection is time wise expensive it will become troublesome to carry it out at close intervals in a real time application and with increased intervals between collision detection the risk of collision tunneling increases. One solution may be to update the broad phase structures and carry out collision detection at closer intervals for objects that move very quickly in relation to their bounding sphere size and using longer intervals for objects that move slowly in relation to their bounding sphere size. This may aleviate the problems of collision tunneling while not requiring all objects to perform collision detection at unrealistically close intervals.

4.2 Memory usage

Memory usage may not be the most heavily discussed aspect of broad phase algorithms but it should not be completely forgotten. In some portable devices fast memory can be sparse and even in workstations with a lot of memory it may be a important factor if the number of objects simulated is very high.
In the case of a naive base case node it only uses a total of 8 bytes of memory since it only carries an object pointer and an object used for counting test results. Without the non essential counting object it effectively only is 4 bytes in working size and there is ever only one naive node no matter how many objects it relates to. In addition to the 4 bytes of the naive node are all the object pointers contained within the objects for upholding the linked list, each of these are 4 bytes in size. If \( n \) is the number of objects this makes \( 4 + (n \times 4) \) bytes of memory usage.

The octtree nodes use a total of 80 bytes each including the 4 byte non necessary diagnostic counter object making 76 bytes the effective size of a single octtree node. In contrast to the naive case the octtree however does not have one node, it has one node at the first level that branches out to 8 additional nodes on the next level and so forth. In a 4 level tree this makes for \( 1 + 8 + 64 + 512 = 585 \) nodes. The result of this is \( 585 \times 76 = 44460 \) bytes of memory and since the octtree also uses linked lists like the naive case this makes for \( n \) being number of object \( 44460 + (n \times 4) \) bytes of memory usage. It should however be mentioned that because the leafs of the octtree all have 8 unused octtree node pointers \( 512 \times 8 \times 4 = 16384 \) bytes are not even used and something of a waste of memory. This waste of memory grows quickly with added levels, at level 5 it rises to 131072 bytes and at level 6 a substantial 1048576 bytes. This should be taken into consideration when considering the usage of high level octtrees in low memory environments.

Having a look at the hierarchical grid algorithm it can initially seem a bit bloated with its 37456 byte node, this also includes the non essential counter object so in effect it is really 37452 bytes in size. The hierarchical grid however only uses one node and it contains the pointers for all the sections in all its grids. It also uses the same type of linked lists as the naive and octtree algorithms but every object also carries section and level numbers at 4 byte extra each. This makes the overall memory usage if \( n \) is the number of objects \( 37452 + (n \times (4 + 4 + 4)) \) bytes in a 32 bit memory system.

### 4.2 Collision detection

It is important to understand that the use of a broad phase algorithm does not come without a cost. There is a considerable amount of computations and memory usage going into upholding and updating the structure behind it. It can be argued that there should be a lower threshold in the number of objects in the scene where using a broad phase approach no longer will be beneficial from a performance standpoint. Finding such a threshold purely theoretically is not a easy task, there are just too many variables in hardware, software and the actual simulated environment to take into account, test results are discussed below.

The results from test 1 and test 2 show performance thresholds for at what number of environmental objects the octtree and hierarchial overcome the naive base case and each other. It should however be considered that the test algorithm uses a very lightweight narrow phase and collision response. For some light weight simple game implementations this may be fine but for more advanced games may use a more advanced and computationally heavy narrow phase and collision response.

The test results from both test 1 and test 2 imply that for small numbers of environmental objects the octree and naive case are faster than the hierarchical grid algorithm. The hierarchical grid however catches up to the naive case near 100 environmental objects and later on also the octree algorithm at just above 400 objects in test 1 and near 400 objects in test 2.

In the case that a more advanced narrow phase and collision response is used the thresholds for the octree and especially the hierarchical grid can be argued to be found at slightly lower numbers of environmental objects. This in light of "test 3" and "test 4" that show that the octree and hierarchical grid clearly cull a lot of unnecessary narrow phase tests but do not necessarily execute faster because of it. It is perhaps most clear in the case of the hierarchical grid that has the lowest number of narrow phase tests but also the worst performance for very low numbers of environmental objects (test 1 and test 2). This can be seen as an indication that the hierarchical grid algorithm makes heavier overhead computations that are overshadowed at higher numbers of environmental objects because of the amount of time it takes to execute all of the extra narrow phase tests. If the narrow phase tests were heavier to execute they would likely overshadow the overhead computations at a lower number and thus lower the performance thresholds for the broad phase algorithm, especially so for the hierarchical grid.

When comparing test 1 to test 2 there is little if any difference in runtime performance for the naive case, this is to be expected since the naive case does not take into account the positions of objects in the environment, it simply tests all objects no matter their positioning.

In the case of the octree however there is a noticeable drop in performance in test 2 compared to test
1, test 1 being the one that used the uneven dispersion of objects throughout the environment and test 2 having placed all objects at random. This may seem to be an unexpected result, a more evenly dispersed body of objects should mean that objects should be more evenly dispersed among the tree nodes and thus the amount at each node should be less. This should theoretically mean better performance but as it appears here, so is not the case. There is however one thing about an octtree that is easily overlooked, it divides the environment along the X, Y and Z axes and any object that comes into contact with these axes are stuck at a higher level than if it had not. The problem here seems to be that when a large amount of the objects are focused at the lower 10% of the y-axis fewer objects are fitted into the first two levels of the octtree. Because of their position they will only be able to collide with two of the three axis for the first levels of the octtree, but if the objects are evenly dispersed more objects are eligible to collide with all axis at the top levels of the octree. As mentioned in the methods section an object that is placed at the top of an octree will be testing against all children nodes which is a heavy load performance wise, so more heavily populated top levels means lower performance. Test 3 and Test 4 also show telltale signs of this, showing that more narrow phase tests are performed for an octree in the evenly dispersed environment of test 4.

The hierarchical grid especially for high numbers of objects show considerable gains in performance in test 2 compared to test 1, this was expected for much the same reason considered for the octtree above. With objects being more evenly distributed in the environment grid sections should theoretically on average be less populated and thus fewer $O(n^2)$ comparison be made. Test 3 and test 4 also show much the same result, the amount of comparisons for the hierarchical grid go down largely proportionate to execution times in test 1 and test 2.

It should be mentioned that Q. Avril et al [17] claim that broad phase algorithms taking advantage of multi-core architectures [18] may significantly improve broad phase performance in environments with large amounts of objects present. It is however also claimed that for small amounts of objects such an algorithm may run slower than an algorithm running on a single core.

4.3 Updating Objects

Insertion and removal times for objects are perhaps more important to consider than what at first can be thought. It’s easy to think an object can just be left where it is after having been inserted into a broad phase structure but that is not entirely true. For stationary objects it is often fine to leave objects where they are when using bounding spheres but moving objects may need to be removed and reinserted with every collision detection update frame.

The naive case does not need to make these updates since it has no real structure to take into account, it does not matter in what way the object pointers in its linked list are arranged. When it comes to the octree and hierarchical grid algorithms, updating moving objects may however be a necessity. If an object moves or transforms in some other way it may fall into another part of the structure and collision detection may fail or the object may not be found when removal is called.

In the case of the hierarchical grid it does not need to update an objects pointers position if the updated object has not changed its size and creates the same hash number it had before the update. The octtree algorithm however updates every moving or transforming object every time, this is done because calculating if it will still reside in the same place within the structure is more complicated than actually removing it and reinserting it from the top. This may be somewhat of a problem since removal of objects in an octtree did not show very high performance in test 6, something that is discussed below. In animations with few moving objects updating objects may not be a big issue but if there are a large amount of moving objects in a scene with an overall large amount of environmental objects it may need consideration.

Removal of objects from an octree was in test 6 shown to be somewhat of a weak point, even if it is significantly faster than the naive case it lags far behind the hierarchical grid algorithm. This is most likely because an octree typically traverses the levels to find the affected object while a hierarchical grid algorithm can either call its hashfunction to find the correct grid section that needs to be searched or use the objects own internal section number to find the right section. It may be an idea to implement an octree that works on objects that carry a pointer to the part of the tree they are located. This way traversal of several levels of the tree will be unnecessary for removal of objects and performance may see a boost. The memory cost of adding pointers to every object in the scene is in modern memory handling terms not substantial. In a 32 bit memory system a pointer is 4 bytes in size so even for the greatest numbers of environmental objects (6400) tested in this paper that would only mean 25.6Kbytes or for a 64 bit system, 51.2Kbytes of extra memory.
5 Conclusion

The broad phase approaches discussed in this paper do not come completely without flaws. Using them in environments where the upper and lower bounds of objects sizes are not known may have an negative impact on performance. Alternatively dynamic versions of the broad phase algorithms may be used to avoid having to plan ahead but performance may be erratic and less dependable.

Even if the broad phase algorithms discussed in this paper greatly reduce the $O(n^2)$ complexity of the naive base the narrow phase tests still use the same basic technique. To avoid performance degradation at high numbers of objects the size of the broad phase structures should be adjusted accordingly.

It has been found that using certain types of bounding volumes may cause complications with some broad phase approaches. It is recommended to thoroughly consider the choice of bounding volume types to suit the broad phase algorithms used.

Collision tunneling was found to not only be a problem for general collision detection, it also demands for the broad phase structures to be updated accordingly. Something that may easily be overlooked since it is rarely brought up in this context. A simple broad phase updating scheme was proposed to reduce the risk of tunneling without affecting performance significantly.

When it comes to memory usage it was found that using the discussed broad phase algorithms require substantially larger amounts of memory than the naive base case. In a modern stationary computer the amounts of memory used can however be considered small enough to largely be ignored. In mobile systems like mobile phones where memory is not as abundant some consideration may however be taken in the matter.

In the end, the decision to use a broad phase algorithm is often dependant on the actual performance gains that it is likely to produce. In this paper it was found that at low amounts of objects, using an octree or even a naive approach are preferable over using a hierarchical grid. However as objects present in the environment become more numerable the hierarchical grid eventually shows the greatest performance. In cases where more advanced and computationally heavy narrow phase algorithms are used, this is likely to tip the performance scale even further in the direction of algorithms that cull large numbers of narrow phase tests. In this paper the hierarchical grid was found to cull the largest percentage of narrow phase collision tests at all tested numbers of environmental objects, followed by the octree and naive case algorithms in that order.

The typical octree algorithm has been found to utilize a less efficient structural updating scheme than the hierarchical grid. A partial solution to the problem can be found in a way to speed up removal of objects at the expense of extra memory usage.
6 Acknowledgements

Thanks goes out to Stefan Seipel for his technical advice and critical eye, his involvement has played an important role in the completion of this paper.
References


