Thesis no: BCS-2016-01

View-Dependent Collision Detection and Response Using Octrees

Albin Hermansson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science of Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Albin Hermansson
E-mail: k.albin.hermansson@telia.com

University advisor:
Dr. Prashant Goswami
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Collision is a basic necessity in most simulated environments, especially video games, which demand user interaction. Octrees are a way to divide the simulated environments into smaller, more manageable parts, and is a hierarchical tree-structure, where each node has eight children. Octrees and similar tree-structural methods have been used frequently to optimize collision calculations and partition the objects in the 3D space.
Objectives. The aim of this thesis is to find a way to further improve upon the octree structure, by using a two-level octree structure, and simplify the collision of objects that do not demand much complexity, due to their size or the geometric simplicity of their 3D models, this is done by calculating how many pixels the objects occupy on the screen, and use that as a factor when deciding the depth of their individual octrees.
Methods. Each object in the 3D environment is divided using an octree. these octrees generated for the objects are then placed in a larger octree. This large octree use the smaller ones to check collision between the objects. The pixel area occupied on the screen by the objects' octrees is used to determine what depth of the octrees will be check for intersection. Two test scenes were set up to test our model.
Results. Our implementation could effectively reduce the depth of octrees belonging to objects occupying little space on the screen. The experiments also showed that the reduced depth could be used with only a slight loss in accuracy. The accuracy loss increased when more objects were used.
Conclusions. The results gained in the thesis show that the pixel area can be used effectively, and the simplified octrees can still represent the objects adequately, resulting in a cheaper but slightly less accurate collision.

Keywords: Collision, Octree, Pixel Area, Level of Detail (LoD)

Contents

Chapter 1

Introduction

In any virtual environment, interaction between different objects is one of the most fundamental requirements. Object interaction is what allows the user to affect the virtual environment, and allows for responses to the actions of the user. In the real world, objects are corporeal, meaning they can not pass through each other without first colliding with the surface. This property is not inherently true in simulated environments, but instead needs to be manually programmed. The environment needs to know when objects overlap, and properly move them to simulate solidity.[10, 11] In video games particularly, object collision is required to allow the player to move in the game world and fulfill the objectives set by the game.[6]

In a 3-Dimensional (3D) game environment, visual objects are shown using polygonal models. A detailed model can consist of thousands of polygons in modern games, and a scene can contain hundreds of models at a time. Calculating collision between these polygons can be a very expensive process, being potentially thousands of comparisons. In a game, these objects often have to be compared for collision dozens of times per second, to give the player an impression of realism. To alleviate this, simple geometric shapes are used to represent the objects, allowing intersection tests between several polygons at a time, as explained by Huang.[7]

Extending this, several objects can be represented using a single larger bounding box. Dividing the objects of the world into larger bounding boxes allows determining if it is impossible for the geometry of two objects to intersect, since two objects in different boxes can not possibly collide with each other. Such a structure can be implemented in several ways, but one approach is called an octree.[9] An octree is a hierarchical structure composed of several layers of boxes, all aligned to the same axis. Objects are placed in these boxes depending on their position in the 3D environment, and then collision is only checked against the objects in the same box. This and similar techniques have been used to reduce the computational cost of collision for several years.[4]

A two-level octree structure will be used for this experiment, the first level creates an octree around each object, to be used for collision calculation, and the second level creates an octree around all objects, and the objects in the same node of this tree will be checked against each other for collision.

To see these virtual environments full of objects, we use a 2-Dimensional (2D) screen. This means that, just like in the real world, objects further away will appear smaller and with less detail. This is doubly true for virtual environments, since the screen has a limited amount of pixels available to display the objects, and each pixel can only be one color at a time. This means that the same object will occupy a different amount of pixels on the screen depending on it's position in the simulated environment.

This thesis proposes that this can be used advantageously during collision computation. This thesis studies changes in performance and accuracy when using a rougher estimate of the bounding boxes belonging to objects occupying a smaller pixel area, to evaluate whether it can be a reasonable alternative in video games.

1.1 Problem Definition

Calculating collision between objects in a virtual environment is a costly process. As such we need a way to partition the objects into boxes represented by an octree. Further, since testing against complex models is a difficult and costly process, each object need to divide it's vertices in it's own octree.

In order to use the object's relative size on the screen for as a variable in collision detail, a way need to be devised to calculate the area occupied on the screen for each object, measured in pixels.

Finally, two intersection tests need to be built. One only using the octrees of the objects, and another one which uses the octrees, but also uses the area occupied by the pixels as a guideline to traversing the octree.

1.2 Aims and Objectives

The aim of this project is to determine if a reduction in collision checks can still have a level of precision adequate to not miss collision or give false positives. The aim is to use the relative size of the objects on the screen to reduce the accuracy of their collision boxes, in order to give us fewer checks. Since octrees are a hierarchical structure, where each level has 8^n boxes, where n is the current level

of the tree. If an object's collision check can be effectively reduced by one or two levels, there is a significant reduction in the number of boxes to check collision between.

In order to do this we first need to construct the two-level hierarchy, to partition the virtual space, and divide the polygons of the individual objects. Then we need to calculate the screen space occupied by each object, to be able to determine the depth of the individual collision.

1.3 Research Questions

We are going to use octrees to partition our objects in the world, in an attempt to optimize the collision. The performance and accuracy of the collision depend on the object properties. Complexity and size of individual objects both affect performance and accuracy. The severity of these factors vary depending on the number of objects. Therefore, we need to compare performance with differences in these factors.

Since we will be using the screen space occupied as a factor in the depth of the octrees, we need a fast way to determine how much screen space each object occupies. Viewing this as a separate part of the problem, the final research questions are as follows:

RQ 1 Can collision detection and response be optimized based on distance from the camera, using varying levels of octrees, depending on the object's occupied area on the screen to reduce the number of collision checks?

RQ 1.1 Is the reduction of collision checks dependent on the size of the object?

RQ 1.2 Is the reduction of collision checks dependent on the complexity of the object?

RQ 1.3 Do the number of objects affect the reduction of collision checks?

RQ 2 Is it possible to calculate the depth of an octree based on pixels occupied on the screen?

1.4 Approach

Two cases will be compared. The first case is without using the pixel calculation, and the depth of the octrees are always the same. The second case will use

the pixel area of an object to determine the depth of the octrees for collision calculation. The maximum depth of the octrees for the objects will be three in all tests. As the number of nodes on the lowest level in this case becomes $512(8^3)$, which is a sufficient number of boxes. Increasing the depth would give a shape more similar to the models used, but a depth of three is detailed enough, for the purposes of this test. The computer that runs the test is also not powerful enough to use a depth of four or higher and have a framerate that will give a clear result. Given the same virtual environment in both these cases, with the same movement of objects, the following will be measured:

- Number of Collisions
- Position of each collision
- Which objects collided
- Time of collision

For the case where we use pixel area for depth, measurements will also be taken for which level of the tree is used for the collision.

The test will be implemented using Unity3D Version 5.3.4f1 (personal edition). The octrees are generated upon program start and are static during the execution time, to avoid unecessary calculations in runtime. The objects can and will exist in several nodes at once, and only check collision against objects in the same node. When pixel area is used, calculation of the area occupied is done at program start and whenever an object enters a new node. This because it is unnecessary to do it every frame, since an object will occupy approximately the same size on the screen if it does not move outside it's starting node. The implementation is run and tested on a Hewlett-Packard® Pavilion 17 Notebook PC, with the following specifications:

Operating System: Windows™ 10 Home Edition 64-bit
Processor: Intel® Core™ i5-4210U CPU @ 1.70GHz
Graphics Card: Nvidia® GeForce™ 840M, 6GB
RAM: 8GB

Chapter 2

Related Work

The subject of sorting objects in virtual space has been widely researched in the past. Tree structures such as the octree and similar models, such as quadtrees and boxtrees have been in place for several years to organize objects.[2, 4] Brunet et al mainly talks about using octrees to represent individual models for rendering purposes, however the same principle can be applied for collision detection, as the model data still needs to be divided into the tree. Brunet et al investigates a model they named an Extended Octree, which is synonymous with a regular octree today, in the world of computing. The extended octree model does not allow for box subdivisions if a box is empty, and it allows storage of several data types. Brunet et al use them to store model data, so information about the faces, edges and vertices are stored in the extended trees.

In [2], Barequet et al suggests a similar model called the Boxtree, which operates much the same as an octree, with the difference that each box only contain two children. Unlike the octree, the orientation and size of these boxes can be arbitrary, allowing for a more flexible tree, containing the same amount of data but stored more effectively. We have chosen not to use the boxtree in this study as it was unnecessary for our purposes, and the axis-aligned boxes of the octree gives us sufficient precision.

Octrees are further used for general space partitioning, and not only as representations of single models. Zou et al[5] use them combined with a 2D plane collision algorithm to check for intersection between objects globally in a 3D environment. Zou et al are not the only ones to do this, as octrees are frequently used, either by themselves (Lucchesi et al[9]) or in conjunction with another technique (Jung et al[8]). Overall, space and model partitioning techniques are nothing new, and have existed for a long time, but they are being continually expanded upon and improved.[7, 12, 13]

Chapter 3

Method

In order to collect the desired data, environments need to be created. The environments have a number of objects of varying shapes and sizes placed in a 3D space. Some of the objects are then set to move in different directions and speeds, in order to get a controlled environment where meaningful data could be collected.

3.1 Defining the Two-Level Octree Structure

Two different implementations are being tested, both using the same test environments, to ensure the only differences in the collected data are from the different implementations. We are using a two-level octree structure to organize the objects in the 3D space. The first level of the structure is a global octree holding all the separate objects placed in the world. The second level are local octrees, one created for each object, containing the vertex data of the object's 3D model.

3.1.1 Basic Octree Design

An octree is a method of subdividing a 3D space into smaller, more manageable parts. It starts out by identifying the extreme points of all containing data. Extremes here meaning the maximum and minimum value possible by the data, since we are partitioning a 3D space, the data is our object coordinates in the virtual environment. From these maximum and minimum coordinates, we identify two separate points in the 3D space. These points are used to create an axis-aligned bounding box.

Since the boundaries for the box are the maximum and minimum coordinates of our data, it is guaranteed to encapsulate all objects in the 3D space. Because we require eight points to create a box, and we have identified two points, the maximum and minimum points of objects in 3D space, we can create the six other points by using the x, y and z coordinates in different combinations. Using all the possible combinations of these, we have a total of eight different points in 3D

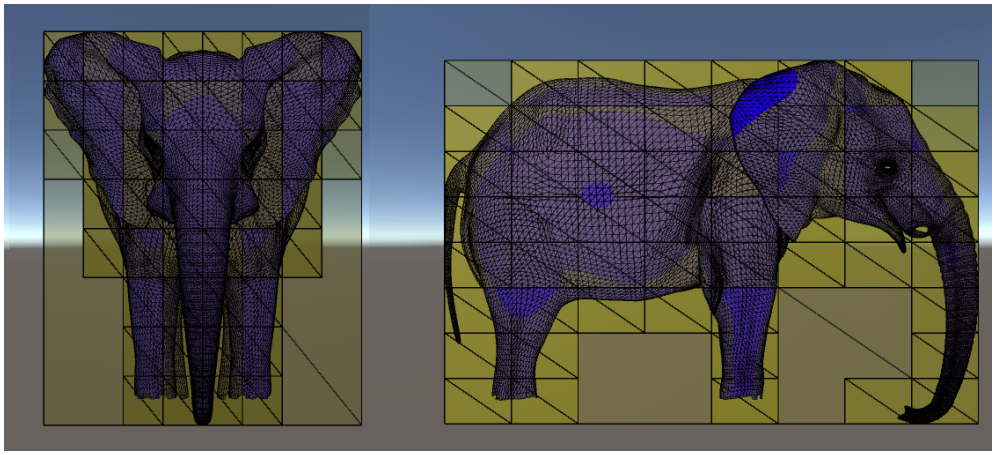

Figure 3.1: Front and side view of the octree on an Elephant model

space.

The bounding box created by these eight points is then divided into eight smaller boxes. These eight boxes are placed as children nodes to the bigger parent-box, creating the tree-structure of the octree. The boxes are $^1/_8$ the size of the original box, and they are each placed in a different corner of the parent box, so they do not overlap and are all contained within the parent box.

This process is repeated for each of the eight nodes, each creating eight children nodes until the octree has reached a desired depth. Once the subdivision is done, each of the smallest boxes contain an amount of objects. If a box does not contain any objects, it will not divide itself into eight children.

3.1.2 The First Level of the Octree Structure

The first level of our Octree Structure is the World level. This level is one large octree containing all the objects placed in our 3D space. This tree places the objects in it's nodes, and then checks for collision between the objects in the same node. These objects only need to check collision against other objects in the same node. This is because the spatial subdivision tells us right away that objects in the same node are the only ones with possible intersection. This tree structure holding the objects in the 3D environment will be referenced in this thesis as the WorldTree from here.

3.1.3 The Second Level of the Octree Structure

Each object also contains it's own octree. But instead of position of objects in 3D space, these are calculated using the positions of the vertices forming the model

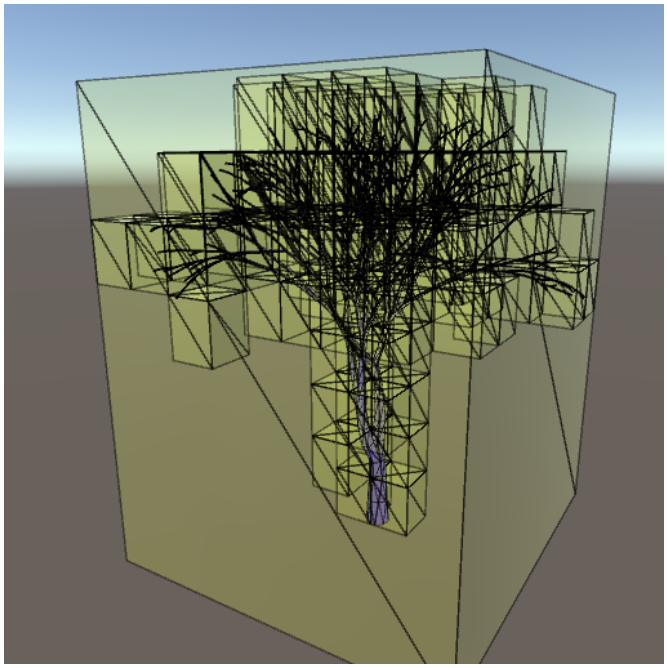

Figure 3.2: Graphical representation of octree on a 3D model

in 3D space. It follows the same hierarchical structure as the WorldTree does, but it's total size is relative to the model used, and each box contains information about the vertices instead of the objects. Boxes that hold no information are flagged as empty, and will not be used. These octrees for the objects, called ObjectTrees, are what will be used to measure collision between the objects. The final result of an ObjectTree is illustrated in 3.2. Only the boxes containing data are shown.

3.2 Intersection and Overlap Calculation

All objects in our scene are represented by ObjectTrees, and these are the ones to be used for calculating intersection. Since they are all aligned in the same direction, we can use basic Axis-Aligned Bounding Box (AABB) collision[11]. The collision works as follows: For each axis, in a collision between two objects A and B, there is a collision if, and only if the follow conditions are met.

- The maximum value of A on the axis is equal to or more than the minimum value of B on the axis.
- The maximum value of B on the axis is equal to or more than the minimum value of A on the axis.

If both of these are true for each axis, there is an intersection. We use this intersection test both to determine which node in the WorldTree and object belongs to, as well as to determine intersection between two different ObjectTrees.

When determining which node in the WorldTree an object belongs to, we only check collision between the object and the lowest level of the Worldtree, since only the nodes of the WorldTree handle information about objects. Objects in the same node of the WorldTree are then checked against each other for collision.

The objects check collision between each other using their respective octrees. This is a process done in multiple steps. First, the largest bounding boxes of the two octrees are checked against each other. If these intersect, both octrees check against the eight children of the large boxes. If any of these intersect, their children are checked for intersection. This process is repeated until the lowest level has been checked for intersection. If the lowest level detects a collision, we know it is true.

In our case, where the object's octrees have a deepest level of three, we can calculate the maximum number of checks for each collision. The maximum level of three is chosen because it gives a shape around our model that does not have much empty space. Using a higher depth gives a more accurate representation, but it is not necessary for our test.

Our collision is calculated in hierarchical steps, thanks to the hierarchical structure of our octrees. First, one check is made between the two largest boxes. Then up to eight checks are made, one for each child to the large box. Since we are checking between two trees, the eight boxes of one tree are each checked against the eight boxes of the other tree. This process is repeated until we reach the lowest level. Giving us a maximum number of checks of $1+(8*8)+(8*8)+(8*8) = 193$. In the test where we use our pixel area algorithm to calculate the depth to use, collision stops at the depth the algorithm tells us to use, reducing the number of checks by $8 * 8 = 64$ for each level of the octree not checked.

However, in the common case, this number is reduced. Since, as soon as we find that two children intersect, we stop checking between the other boxes at the same level. Meaning if the first two children we check intersection between on one level intersect, we do not check the other seven children of either octree on that level. Meaning we only have 193 checks in the worst possible case, but in actuality, the number of checks varies between the worst case, 193 checks, and the best case, which is $1 + 1 + 1 + 1 = 4$ checks.

Whenever a collision is detected, it is documented to a file, the following

attributes are stored about the collision:

- Which objects collided
- How much time has passed since program start, in seconds
- What position the objects were in at time of collision
- What level of the ObjectTrees were measured for the objects

3.3 Resolving a Collision

Once it has been determined that two objects overlap, something needs to happen. We want to avoid having the same collision between two objects registered more than once, to avoid obstructing our data. When we find that a collision has happened, we then reverse the direction of the objects who collided, if they were moving. If they were stationary, we do not do anything. This puts a requirement on our scene to never have two stationary objects collide, as it would obstruct the data with the same collision over and over.

After we have reversed the direction of the moving objects, we then move them in their new direction by a set length l. Usually, it is desired to calculate l as the amount of the two objects that are overlapping, and moving each object by half of l would place them so they are not intersecting, but close enough to not leave any empty space between them. For the purpose of our test, l is given as follows:

$$l = (Velocity_{Object1} + Velocity_{Object2}) * 1.5$$

This will result in empty space between the two objects, but it is also a cheap way to guarantee they no longer overlap. This is favorable for the purpose of our test, as we do not require realism to get the desired data.

3.4 Pixel Area from Cuboid Shapes

In order to calculate the pixel area occupied by our objects, we use their individual octrees. We know the nodes of these octrees are cuboid in shape, since they're derived from three points in 3D-space, the position of the object and the maximum and minimum coordinates of their bounding box. We can use this information to calculate the pixel area occupied by the octrees of the objects.

We use the lower nodes of the ObjectTree to decide the total area it occupies on the screen at any given time. The naive approach for this is to find the two corners of the cuboid furthest away from each other, use their respective x and y

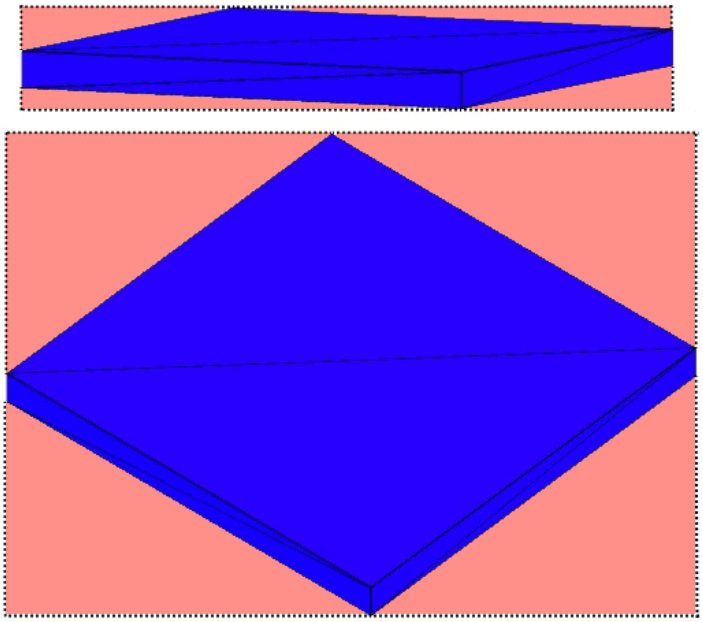

Figure 3.3: Differences in pixel area

coordinates on the screen to create a third point. These three points would then be used to create a plane, who's surface area encapsulates the entire surface area of the cuboid.

This approach is wrong for two major reasons. Firstly, the total area would occupy more space than the actual area used by the object. This leads to inaccurate calculations and observations, as it gives us a lot of undesired space. Secondly, the relative size of the undesired space occupied varies immensely based on the perceived orientation of the object, as illustrated in 3.3, which shows the same cuboid from two different angles, and the undesired space is colored pink. As can be seen, the undesired space vary greatly in size depending on the angle, so we need a more detailed way to get the pixel area of our cuboids.

However, any cuboid in 2D can be described as either a rectangle or a hexagon.[3] If the cuboid is viewed from an angle perpendicular to any of the surfaces, it's 2D projection will be a rectangle. Should it be viewed from any other angle, it will form an irregular hexagon. The length of each side of the hexagon will vary depending on the length of the cuboid edges.

When viewed in a perpendicular angle, we have the naive case, where the pixel area is a perfect rectangle. In any other case, it will give us a hexagonal shape of some sort. Following the basic rules of geometry, every hexagon can be broken

down into four triangles.

The pixel area on the screen occupied by our cuboid is the sum of the area of each of these triangles. The area of each triangle can be easily calculated. Since we know the length of all the edges on our triangles, we can use Heron's formula[1] to calculate the area of each triangle.

3.5 Using Pixel Area for Octree Traversal

Once the pixel area for all our objects have been calculated, we identify the largest and smallest area in the scene. These two are used as a base to determine what depth will be used by our ObjectTrees for computing collision. The object with the largest area will always traverse the full depth of three, and the object with the smallest area will always only use a single level. For every other object, the depth to be used is calculated using the following formula:

$$Depth_{Object} = \frac{Area_{Object}}{Area_{Smallest} + Area_{Largest}} * Depth_{Max} + Depth_{Min}$$

The first part of the algorithm will give us a value between n and m, where:

$$n = \frac{Area_{Smallest}}{Area_{Smallest} + Area_{Largest}} \qquad m = \frac{Area_{Largest}}{Area_{Smallest} + Area_{Largest}}$$

Since n and m are between 0 and 1 as so: $1 > m > n > 0$, we know our number to be in this range as well. Therefore, we multiply it by the maximum allowed depth of the tree, to place it in the range of 0 to Max_{Depth}. While not illustrated in the algorithm, this number is then rounded down to the nearest integer, because the depth levels we use are always complete integers. Finally, we add the minimum depth, since we always want to go down to a minimum level of the tree. A final check is made, if the final value is more than the maximum allowed depth, we reduce it to the maximum depth, three, since going beyond that would be illegal.

3.6 Setting up the Test Scenes

In order to find out if the pixel area can be used effectively, we need to set up a scene for the objects to collide in. In this scene, objects are placed with varying model size and shape in different positions. Some of these objects are given directions and spaces to move in, to create collisions.

For the purpose of the experiment, the octree containing the objects will be static, meaning it won't grow or shrink should objects move outside it's border, or towards the center. This is because recalculating the size of the tree dynamically

is costly, as all objects would have to be re-placed in the nodes. Doing this every frame is not favorable. To avoid this, eight corner-objects are placed in the world, at points forming a large cube, in which we place the objects we use for our test data.

From these eight corners, we now have an established space to place our objects in, where we can allow them to move around. In the first test, 100 non-moving objects were placed in the scene, and 50 moving objects were placed. This amount of objects were chosen because the computer running the test would be able to run at a stable framerate of 30 frames per second with this many objects. 50 objects were placed to move, as in many video games, there are in general more stationary than moving objects. The test was executed three times using the pixel area to reduce octree depth, and three times without reducing the depth, going full depth in each tree during collision. Each of these executions ran for four minutes, or 240 seconds. In the second test, 140 non-moving objects were placed in the scene, at different locations than the previous test, and 70 moving objects were placed. These amounts were chosen as an extension of the previous testing scene, simulating a more crowded environment. These were also placed at different locations, and were all set to move in different directions and with different velocities. The objects were all given a maximum depth of three on their individual octrees.

The moving objects each had their individual directions and velocities, independent of the other objects. These were all different from each other to give the scene more diverse possibilities of impacts, and allowing many different objects to collide.

For the test, different object models were used, to give objects close to each other different sizes and complexity. The models used can be seen in 3.4. The first model is shaped like a tree, with a very thin body. This means the octree for this model will have most of it's non-empty boxes on the top side, leaving the bottom mostly empty. This can be seen in 3.2, as it shows the tree model with the generated octree boxes rendered over it. The second model is shaped like a human with arms stretched to the side. The octree generated for this model will be mostly empty on the outer corners, as most of the model's vertices are focused near the middle. The outstretched arms give the model a larger width than if the arms were hanging down, this creates a lot of empty space at the eight corners of the model. The third model is a simple frustum, it will generate a mostly full tree, with the top corners left empty. This is a very simple model yet it creates a fairly full tree. The fourth model is of an elephant. This model has a lot of vertices, mostly evenly spread out through the model. Since the ears of the elephant are quite large, they create a gap between the rest of the model and the outer edges of the octree generated, with exception by the legs and trunk, who

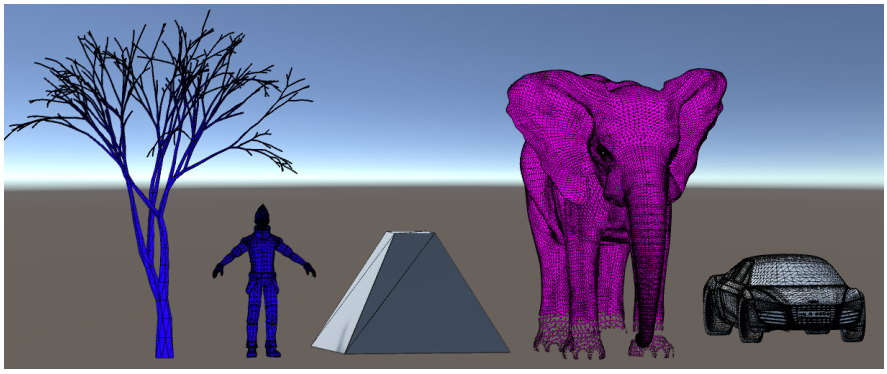

Figure 3.4: Models used for the tests

all reach the lower edges. It also leaves some hollow space between the legs and the trunk. The fifth and last model used is of an Audi R8 car. Since the roof of the car is fairly low, and since it's lower body is mostly rectangular in shape, it will generate a mostly full rectangular tree, with empty spaces in front of and behind the roof.

These models were placed at point in the world, within the eight corners we designated as edges of our worldtree. All models had copies that were stationary, and copies that moved around. As stated earlier, many instances of these objects were created, different amounts for each of our test scenes.

Chapter 4

Results

4.1 The First Test

Six executions were made, three using pixel area to simplify the collision boxes, and three without using pixel area. The test scene had 100 non-moving objects and 50 moving objects. Each object used one of the five models shown previously. Each execution ran for 240 seconds. The results are as follows:

With Pixel Area

Six executions were done on the first testing scene, three of these were made using the pixel area algorithm. The results of these are as follows:

In the first execution. there were a total of 632 collisions registered. In 420 of the collisions, the objects used different levels of depth, while in 212, both objects used the same level of depth. In the second execution, 633 collisions registered, 421 using different levels of depth and 212 using the same level of depth. In the third execution, 635 collisions registered; 422 using different levels of depth and 213 using the same depth. Below is a table showing the complete list of registered collisions during each of the tests. The table below shows the depths used for the ObjectTrees during each collision. The first row shows the depths of the octrees used by the objects during collision. Collisions always happen between two objects. The depth of the first object during collision is the first number, and the number after the dash is the depth of the second object during the same collision. The numbers on rows 2 to 4 show the total number of collisions for each pair of depths used by the octrees.

	Total	1-1	1-2	1-3	2-2	2-3	3-3
Execution 1	632	103	44	214	37	162	72
Execution 2	633	103	44	215	37	162	72
Execution 3	635	104	44	215	37	163	72

Even though the same scene was executed three times, with the same features and settings, we can see there is still a slight change in the number of collisions registered.

Without Pixel Area

When not using the pixel area, all objects always used the full depth of the tree to measure collision. Three executions were done when not using the pixel area, all used the same scenario and the same settings. In the first execution 629 collisions were registered. In the second execution, 631 collisions were registered. In the third and final execution, 629 collisions were registered.

All of these were less than the amount detected when using the pixel area, with the largest recorded difference being between the first execution without pixel area, and the third execution with pixel area. These gave 629 and 635 collisions, respectively, amounting to a total difference of six registered collisions. Since all executions ran for 240 seconds, or four minutes, this gives us a registered difference of 1.5 collisions per minute, in our most extreme recorded cases.

4.2 The Second Test

The second test also had six executions made. The placement of the objects have been changed since the first test. The directions and velocities of the moving objects have also been changed. More objects have also been added to the scene, giving us a total of 140 non-moving objects and 70 moving objects. The results are as follows:

With Pixel Area

Three executions were made using the pixel area to calculate octree depth, giving us the following results:

In the first execution. there were a total of 868 collisions registered. In 655 of the collisions, the objects used different levels of depth, while in 213, the objects used the same level of depth. In the second execution, 871 collisions registered, 657 using different levels of depth and 213 using the same level of depth. In the third execution, 893 collisions registered; 668 using the different levels of depth and 225 using the same levels. Below is a table showing the complete list of registered collisions during each of the tests. The table below shows the depths used during each collision. The table is structured in the same way as the previous table.

	Total	1-1	1-2	1-3	2-2	2-3	3-3
Execution 1	868	81	171	265	46	219	86
Execution 2	871	81	171	267	46	220	86
Execution 3	893	85	174	271	48	223	92

Even though the same scene was executed three times, with the same features and settings, we can see there is still a change in the number of collisions registered, execution 3 notably had 25 more collisions registered than execution 1, and 23 more than execution 2.

Without Pixel Area

We used the same maximum depth in this test as in the previous one, meaning without the pixel area, all objects checked to depth three of their octrees.

Three executions were done when not using the pixel area, all used the same scenario and the same settings. In the first execution 864 collisions were registered. In the second execution, 869 collisions were registered. In the third and final execution, 855 collisions were registered.

The first and third executions gave us less collisions than when using the pixel area, but the second execution gave more than the first execution with pixel area. The third execution gave only 855 collisions, nine less than execution 1, and 14 less than execution 2. The largest recorded difference of this test is much greater than in the first test, which had six more collisions registered on the greatest record than the least. In this test, the fewest registered collisions were 855, on the third execution without pixel area, and the most registered collisions were 893, on the third execution with pixel area. Giving a total difference of 38 collisions. Since the tests ran for four minutes each, this means we had a difference of 9.5 collisions per minute, in the most extreme case.

Chapter 5

Analysis and Discussion

5.1 Interpreting Test Results

As seen in both our tests, when using our pixel area calculation method, more collisions were registered. Since these were not registered when using the full depth of the tree, as shown by our test, it can be concluded that these collisions are so-called "false positives", meaning there was no actual intersection between the graphical models of the two objects.

The largest noted difference was in the second test, between execution 3 without the pixel area, which registered 855 collisions, and execution 3 with the pixel area, which registered 893 collisions. This is 38 fewer collisions. However, both of these executions showed drastically different results from the other four executions of the second test.

These other four executions gave us 864 and 869 collisions when not using the pixel area; and 868 and 871 when using the pixel area. These four executions have a largest difference of seven collisions, between execution 1 without pixel area, and execution 2 with pixel area. Since we have a difference of five between execution 1 and 2 when not using pixel area, we can assume this difference is not caused by our implementation, but by an outside factor, such as a background process running on the computer. The two executions with drastically different collisions registered, execution 3 without pixel area and execution 3 with pixel area, are probably caused by similar reasons.

The first test showed much more consistency, with all six executions being close to 630 collisions registered, and the greatest difference being between execution 1 when not using pixel area, at 629 collisions, and execution 3 when using pixel area, with 635 collisions. Giving us a largest difference of 6 collisions between the two, or 1% more collisions. These collisions are so-called "false positives", meaning there is no actual overlap between the models of the objects, but we still register collisions because of our simpler collision boxes.

Since the second test showed such drastic differences, but the first test showed very little difference, it can be concluded the differences are caused by outside factors, since both tests were made using the same algorithms. The only difference between the tests being the number of objects placed in the testing environment, and their relative position and movement.

5.2 Simplifying Collision Detection Using Octrees

Our first research question has three subquestions, which when answered, will provide us with a concluding answer for the main question. The three subquestions are:

- Is the reduction of collision checks dependent on the size of the object?
- Is the reduction of collision checks dependent on the complexity of the object?
- Do the number of objects affect the reduction of collision checks?

All of these questions can be answered separately, to give us our final answer to the major research question.

5.2.1 Collision Check Reduction Based on Object Size

In our test, we have implemented an algorithm that calculates how large area on the screen, in pixels, an object occupies. This is for the purpose of simplifying the collision geometry of the object if it occupies a small area, based on the notion that if an object occupies very few pixels on the screen, details of the object will not be visible.

Based on our tests run, it can be concluded that simplifying the collision geometry of the object -in our case by reducing the depth of an octree, thus using larger boxes to represent the model- is possible without losing much accuracy. Reducing the depth of an octree is an optimization, because each level is eight more boxes to check intersection between for the two objects. Reducing the depth by one level means $8*8 = 64$ fewer checks before it can be determined if a collision happened or not. Since we could reduce the depth of the tree, with only a small reduction in accuracy, we can conclude that we can positively optimize collision, based on the size of the object on the screen.

However, the actual size of the object in 3D space is irrelevant. As a larger object far away will still have a big pixel area, and will use the full depth of it's octree. So it can be stated that the object's size can be used to optimize collision,

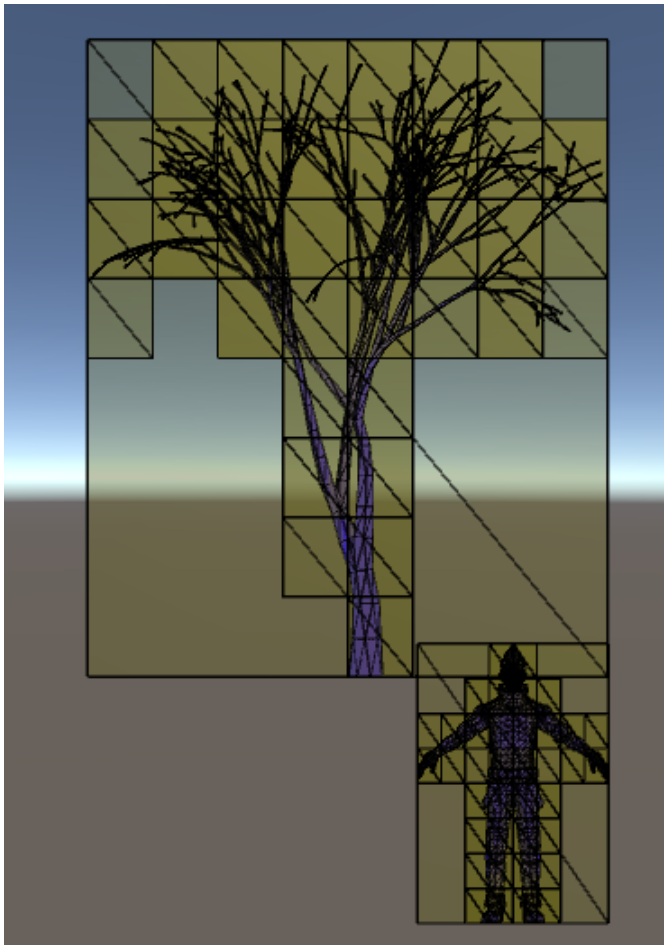

Figure 5.1: Example of a "false positive" intersection

but it's position is also important, since it is a combination of the two that gives us the final depth to use for collision.

5.2.2 Collision Check Reduction Based on Object Complexity

Each object's octree is calculated using the vertices of the object, and contains data for these vertices. Since we have defined complexity of objects as not only the number of vertices they have, but also the position of the vertices and their distance to each other.

The relative position of the vertices is especially important. Our model of a tree, shown in 3.4 has a lot of vertices on the top, forming the branches, but very few at the bottom. It could thus be argued that the tree has very little complexity as a model. Despite this, the tree model generates an octree which is mostly

empty on the lower half, and mostly full in the top half, as shown in 3.2. The elephant model, also seen in 3.4, has a lot more vertices, who are spread out from each other in a more advanced shape than the tree. Despite this, the tree has a lot fewer empty spaces in it's octree, as can be seen in 3.1. A complex model might generate a more complete octree, based on the position of the vertices. Since we worked with a set of models in our tests, and did not do any generation of complex models as part of the investigation, it is inconclusive if the complexity of the model affects the collision.

However, in our tree model, the vertices on the bottom half are all located towards the center, leaving a lot of empty spaces on the lower end. If the pixel area algorithm we use determine that the tree model shall use a depth of one, and another object intersects a tree from the bottom, it will more often give a false positive. In 5.1, we would register an intersection if both objects used an octree with depth one, but if both objects used an octree with depth three, it would not register an intersection, which is correct, since the objects do not actually overlap. For that reason, we can argue that object complexity affects the depth level negatively, giving false positives, if the vertices are distant from each other and leave a lot of empty space. While a complex object, but with fewer and smaller empty spaces affect the depth level positively. Such as the elephant model used, as we see in 3.1, the elephant is a complex model, yet the overall shape leaves very few empty spaces, lessening the chance of a false positive collision.

5.2.3 Collision Check Reduction Based on Number of Objects

As seen in the results of the experiments run, the pixel area algorithm holds the same accuracy with 150 objects in the scene as it did with 210 objects in the scene. However, since each object has it's own octree generated, the program turned increasingly slow at start-up. The scene with 210 objects took 17 minutes to start on the test computer. This can be alleviated by having pre-generated octrees for each object, which there was no time to implement in this study.

Once the program execution has started and all octrees are generated, the program ran without issues and could document a satisfying number of collisions, both when using and when not using our pixel area algorithm, as seen by the experiment results. However, since no experiment was made using very few objects, 10 or less, we can not know for certain if the number of objects hold a positive or negative effect to the optimization level, as we do not know whether the pixel area algorithm is more or less expensive and inaccurate when there are very few objects in the testing scene. Based on our two experiments run, however, we

can state that the pixel area algorithm holds a satisfying level of accuracy in a scene with many objects, as it registered very few false positives on the executions. Though the number of false positives appeared to increase on the second test, meaning the amount of objects can negatively impact the number of false positives.

5.2.4 Reducing Collision Checks for Collision Detection Using Octrees

We have answered every sub-question to our first research question, and will now answer the main question. Every sub-question we posed to this research question has been answered positively, with the possible exception of the second one; "Does the complexity of the objects affect the reduction of collision checks?", as the tests we have done so far leaves it inconclusive.

Since the other two sub-questions could be answered adequately, and a partial answer was given for the third sub-question, we can conclude that yes, we can optimize collision detection and response based on camera distance when using octrees. As it has been determined that we can reduce complexity of octrees on far away and tiny objects on the camera, by lowering the depth of the octrees on objects occupying fewer pixels on the screen, and still retain an acceptable level of quality.

5.3 Calculating Octree Depth Based on Pixel Area

Our second research question we have tried to answer is related to the first one, it is formed as: "Is it possible to calculate the depth of an octree bases on pixels occupied on the screen?" In our implementation, we have created a way to do this. On page 10, we explain how this process works. Since the algorithm uses several steps and might take some time to execute, we only calculate the pixel area of an object on two occasions. When the program starts and when an object enters a new node in the WorldTree. This is to ensure that every object has it's pixel area calculated, and kept updated. We avoid updating the pixel area every frame, since it is unnecessary and will yield approximately the same answer, since the object has not moved very far from it's original position if it is in the same node in the WorldTree. For that reason, we update the pixel area of an object when it has entered a new node, only then has it's size on the screen changed enough to justify the new calculation.

Based on the collected data about which depth objects used during collision calculation, we can determine that the algorithm that calculates pixel area works properly, since we clearly see collisions between objects using different depths

in the tree, and our algorithm is the only factor deciding which depth the trees will use. With this information, we can positively confirm that our algorithm for calculating pixel area works properly, and that it fulfills it's purpose of calculating which depth the octrees should use. This in turn leads to a cheaper collision on the octrees using a reduced depth, though with an increase in false positives, contributing to a positive answer on our first research question.

Chapter 6

Conclusions and Future Work

The experiments have shown that the algorithm tested is valid and may be useful in larger simulated environments. Based on the tests made, we can state that it keeps an acceptable accuracy. Since executions with and without the pixel area algorithm held the same frame-rate, it can also be concluded that the algorithm for calculating the pixel area is not very expensive, as long as it is not run every frame. Each depth level reduced for a collision test gives $8 * 8 = 64$ less checks to determine a collision, meaning an optimization has been achieved when using a reduced depth of the octrees.

6.1 Validity Threats

A lot of the collisions between objects used the depth of one for both objects' octrees. Despite this, the accuracy remained similar to when going full depth all the time. This could potentially mean the models used for the experiments were poorly chosen. It could also mean the collisions checked for were not proper for the testing case, as the model of a tree especially should give false positives to objects approaching from below. Further testing would need to be made to deny or confirm this. It is also uncertain whether our algorithm is useful even in cases with very few objects, as no test has yet been made for this.

6.2 Future Work

The octree used in the study only contain axis-aligned bounding boxes, a potential way to improve this is implement a tree structure with oriented bounding boxes, for further improved accuracy. A way to create pre-generated octrees for the models, and loading them in at program start instead of generating new octrees on each execution could also potentially be a way of improving this algorithm.

Similarly, different applications for the algorithm can be made. In this experiment, studies have only been made on objects placed in a 3D world. Similar tests could be made in a 2D environment. There is also potential for the algo-

rithm to be used in particle systems, to check for internal intersection between the particles, as octrees have been shown to be effective on these before.[13]

References

[1] Heron's formula. Available at: http://mathworld.wolfram.com/HeronsFormula.html, accessed June 12, 2016. Cited on page 12.

[2] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal. Boxtree: a hierarchical representation for surfaces in 3d. volume 15, pages 387 – 96, UK, 1996. Cited on page 5.

[3] M.H. Brill. Chromaticity contour map of the rgb cube: a simple algorithm. *Color Res. Appl. (USA)*, 27(6):421 – 4, 2002. Cited on page 11.

[4] P. Brunet and I. Navazo. Solid representation and operation using extended octrees. *ACM Transactions on Graphics*, 9(2):170 – 97, 1990. Cited on pages 1 and 5.

[5] Zou Chengming and Tang Zhiyong. The collision detection algorithm based on the combination of two-dimensional and dynamic octree. volume vol.1, pages 470 – 3, Piscataway, NJ, USA, 2009. Cited on page 5.

[6] J.-C. Delannoy. The reality of virtual environments. *IEEE Potentials*, 24(3):37 – 9, 2005. Cited on page 1.

[7] Rui Huang. Optimizing collision detection in 3d games with model attribute and bounding boxes. pages 589 – 91, Piscataway, NJ, USA, 2012. Cited on pages 1 and 5.

[8] D. Jung and K.K. Gupta. Octree-based hierarchical distance maps for collision detection. volume vol.1, pages 454 – 9, New York, NY, USA, 1996. Cited on page 5.

[9] B.J. Lucchesi, D.D. Egbert, and Jr. Harris, F.C. A parallel linear octree collision detection algorithm. *International Journal of Computers and their Applications*, 21(4):230–243, 2014. Cited on pages 1 and 5.

[10] Feixiong Luo, Ershun Zhong, Yuefeng Huang, Hui Guo, and Junlai Cheng. Real-time collision detection and response in virtual global terrain environments. pages 2257 – 62, Piscataway, NJ, USA, 2009. Cited on page 1.

[11] F. Policarpo and A. Conci. Real-time collision detection and response. pages 376 –, Los Alamitos, CA, USA, 2001. Cited on pages 1 and 8.

[12] J. Porter-Sobieraj, S. Cygert, D. Kikoła, J. Sikorski, and M. Słodkowski. Optimizing the computation of a parallel 3d finite difference algorithm for graphics processing units. *Concurrency Computation*, 27(6):1591–1602, 2015. Cited on page 5.

[13] Fan WenShan, Wang Bin, J.-C. Paul, and Sun JiaGuang. An octree-based proxy for collision detection in large-scale particle systems. *Science China: Information Sciences*, 56(1):012104 (10 pp.) –, 2013. Cited on pages 5 and 25.