Universiteit Leiden

Computer Science

Optimizing octree updates for visibility determination on dynamic scenes

Name: Hans Wortel
Student-no: 0607940

Date: 28/07/2011

1st supervisor: Dr. Michael Lew
2nd supervisor: Dr. Erwin Bakker

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Chapter 1

Introduction

Visible surface determination concerns the selecting of those parts of a scene that are visible from a certain viewpoint. When this has been determined only those parts need to be rendered. Rendering the scene will usually be done by the graphics processing unit (GPU). While the GPU will take care of rendering the scene correctly, this can be done much faster when less scene geometry is passed to the GPU.

Finding the visible objects in a scene efficiently can be done using a hierarchical data structure like a octree, kd-tree or a bsp-tree. The objects in the scene are kept in the nodes of such a tree. If a node is not visible than its children will not be visible either, which means the visible objects can be found without checking every object separately.

When objects in the scene are moving the data structure needs to be updated which can take a lot of time. In this project I will attempt to find a data structure and method of updating that will allow efficient visible surface determination on dynamic scenes.

1.1 The rendering pipeline

During the process of rendering, a 2d image will be rendered from 3d scene. Using the known scene geometry and an viewpoint in the scene, an image will be created that shows the scene as it would look from the viewpoint. The following steps will be taken when rendering an image in real-time.

1.1.1 Step 1: Preparing scene data

The scene geometry consists of a large number of polygons which are collected in objects. These objects have a certain position in the scene which, in case

of a dynamic object, will change over time. The first step is to determine for all these objects what their location for the currently rendered frame should be.

1.1.2 Step 2: Updating data structures

The objects may be kept in a data structure that allows for a more efficient rendering process (see chapter 3). In the case of a scene with dynamic objects in it this data structure may require updating. When rendering in real-time, recreating the data structure for every frame will usually be too slow. Depending on the data structure the update may be very fast or can otherwise take a significant amount of time.

1.1.3 Step 3: Collecting visible objects

Of all the objects in the scene, some will affect the image that is being rendered, while others will not. An object may be behind the current viewpoint or it may be hidden behind a larger object. In such a case the final image will be unaffected by the object's existence in the scene. To allow efficient rendering these objects may be recognized and discarded early in the rendering process, so no further processing will be done on them.

To find the objects that will be visible a data structure may be used. If it allows for faster processing, too many objects may be selected at this point. This will not affect the image, while making the whole rendering process faster.

1.1.4 Step 4: Sending the visible objects to the GPU

At this point the polygons of the objects selected in step 3 can be send to the GPU, which will render them. As communication between the GPU and CPU is slow, many polygons will usually be sent to the GPU at once, rather than sending few polygons many times.

1.1.5 Step 5: Creating the image

The GPU translates the polygons with respect to the viewpoint. At this point a depth for every polygon is known. Every polygon is then drawn on the output image, using z-buffering (see section 2.2) to ensure that every pixel only shows the closest one of the polygons that are located on that pixel.

1.2 The goal of the project

In this project the goal is to improve the efficiency of step 3 of the rendering pipeline of section 1.1, by creating a data structure and update method that results in little time being spent on the data structure update, while not having an adverse effect on the whole rendering process.

1.3 Related work

Temporal bounding volumes

In [9] a temporal bounding volume (TBV) is used to avoid having to do many updates on the data structure. A TBV is a volume that an object is guaranteed to be in for some number of frames. This object is inserted into the tree instead of the object itself. While this TBV remains hidden there will be no updates necessary for this object, unless the TBV needs to be updated. This works best when the movement of the objects is well known.

Dynamic irregular Octree

The dynamic Irregular Octree[8] is a tree in which objects are inserted in the root and moved down only during a visibility query. Objects are moved down until a leaf node is reached or a slitting plane prevents the object from moving down further. Nodes are split when the number of objects in the node exceeds a threshold value, at which point the objects in the node are used to choose a splitting point, rather than dividing the node in eight equal volumes. Updates are done by removing the object and reinserting it. Disadvantages of this tree are that many more meshes are found than are really visible, the tree can get unbalanced over time and in some cases many objects need to be moved down the tree at once.

Regular grid

In [2] the use of a regular grid instead of a hierarchical data structure is suggested. Fast updates can be done on such a grid, while losing the advantages of an hierarchical data structure. Their algorithm also uses TBVs.

Chapter 2

Visibility determination methods

In this chapter I will briefly discuss visible surface determination methods in order to establish the role of the data structures in these algorithms. Many surveys of visibility determination have been done [4, 7].

2.1 Frustum Culling

Frustum culling is used to find the objects that are visible from a certain camera location in the scene. A volume, called the view frustum, is constructed that contains the part of the scene that is visible. A hierarchical data structure can then be used to efficiently determine the objects contained in the view frustum. Starting from the root node an intersection test between the volume of the tree node and the view frustum is done. If the node's volume is outside the view frustum the objects in the node's subtree will not be visible. If the node's volume is fully contained in the view frustum no more visibility test need to be done on its children as they will all be within the view frustum. If only part of a node's volume is contained in the view frustum intersection tests will be done for all the node's children.

Although frustum culling removes many of the objects in the scene it leaves in the objects that are obscured by other objects. Removing these objects is referred to as occlusion culling, for which many methods have been developed. Frustum culling and occlusion culling are illustrated in figure 2.1. Figure 2.2 illustrates frustum culling using an octree.

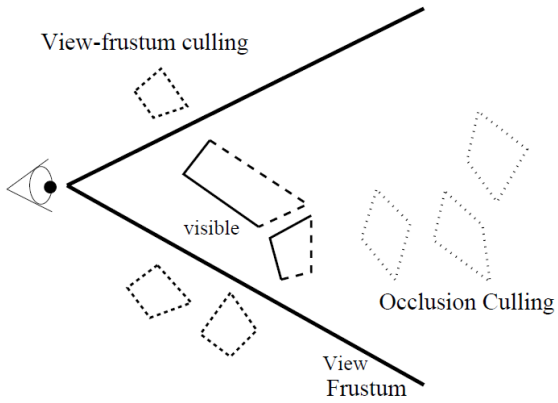

Figure 2.1: Frustum culling and occlusion culling. Taken from [4].

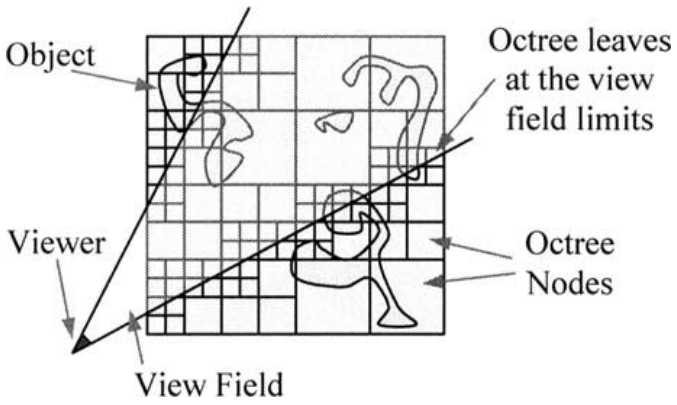

Figure 2.2: Frustum culling using an octree. Taken from [7].

2.2 Z-buffering

In a z-buffer a depth value is kept for every pixel that is being rendered. This depth value is used to determine if an object should be drawn. If the object is closer than the object currently shown on a pixel, the object is drawn and the z-buffer is updated. This algorithm ensures that the correct objects are drawn and is implemented in the GPU for this task, however it requires that all objects are evaluated which will take a long time when many objects are passed to the GPU.

2.3 PVS/Portal rendering

Often used methods for occlusion culling are PVS rendering and portal rendering. In these methods the scene is divided into convex sectors that can be stored in a kd-tree or a BSP tree. Every sector is only visible from other sectors via so called portals, for example windows and doors. The set of visible sectors is either precomputed and stored as a potentially visible set (PVS) [1, 10] or computed on the fly [6]. This method works best for indoor scenes and requires that sectors and portals are defined for the scene. It is not very well suited to dynamic scenes and therefore it will not be considered for this project. This method is illustrated in figure 2.3

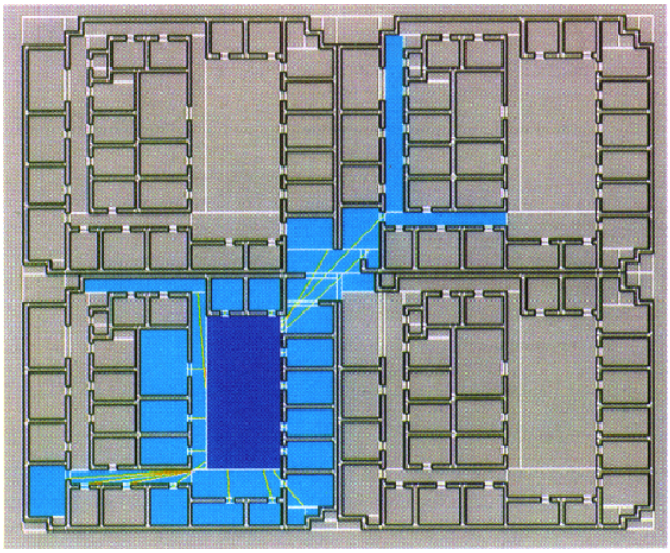

Figure 2.3: PVS rendering: The sectors that are visible(light blue) from a source sector(dark blue). Taken from [10].

2.4 Hierarchical z-buffer

Another occlusion culling method is the hierarchical z-buffer [5]. In this method a hierarchy of z-buffers is used where each element in one z-buffer holds the furthest value of a 2x2 block in the z-buffer one level lower. The high level z-buffers can thus be used to quickly cull large occluded volumes. The visible objects are found by traversing a scene octree front to back. For every node it is determined if the node is visible using the hierarchy of z-buffers, before drawing the objects associated with the node. As it requires reading the z-buffer this method would require a hardware implementation in order to work efficiently.

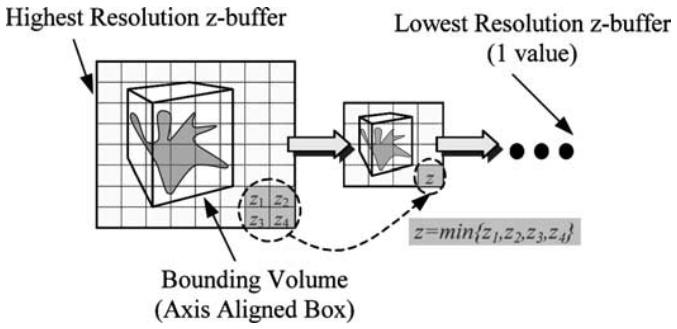

Figure 2.4: The hierarchical z-buffer. Taken from [7].

2.5 Hardware occlusion queries

A similar method is using the hardware occlusion queries that are supported by modern hardware. While traversing a octree front to back occlusion queries can be used to determine the visibility of the nodes before they are drawn. Although the hardware occlusion queries are slow, an algorithm like Coherent Hierarchical Culling [3] shows that this can be efficiently used.

Chapter 3

Data structures and update methods

Many algorithms for visible surface determination have been developed and many of these use octrees or kd-trees. These methods are usually only applied to static scenes. For a dynamic scene such a data structure would have to be updated or recreated for every frame, but this can be slow. In this project some algorithms have been developed with the intention of finding an implementation of an octree that may be updated faster for dynamic scenes.

The octrees store objects that are determined to either be completely visible or not visible at all. A test scene will contain up to several hundreds of thousands of meshes. Axis-aligned bounding boxes are used to represent the volume of the objects.

The implementation of these octrees allows them also to be used as quadtrees. Quadtrees are better suited to some of the tests.

3.1 Algorithm: Octree 1

Several algorithms have been implemented. The most standard one uses an octree in which every object is stored in the smallest node that can contain it, which is the node that has a splitting plane that intersects the object. This means that many objects will be stored in the high (large) nodes as there are likely many objects intersecting their splitting planes, which means that with this octree often many more object than necessary will be drawn. Possible solutions for this are splitting these objects into two objects that fit into the lower nodes, or allowing the whole object to be stored in several nodes. As both of these solutions seem to make the moving of the objects

more complicated they have not been implemented. Splitting the object may have to be done every time it is moved while storing the object in several nodes means that more nodes need to be updated.

When an object moves and thereby leaves the volume of the node that the object is kept in, the tree needs to be updated. This update can be done by removing the object, then updating its location and inserting it in the octree again. As the objects will usually only move a small distance in one frame, this update can be done more efficiently by stopping the removal step when a node is reached that contains the object at its new location and inserting the object starting in this node. This update method has been implemented. Since in this octree objects are moved down in the tree as far as is possible it can also be that after an object moves it can be moved to a lower node in the tree. If this is not done whenever possible some objects may stay in high nodes for a long time after updating, as these nodes are much larger and it may take a long time for a moving object to leave its volume.

3.2 Algorithm: Octree 2

The tree that is used in this algorithm is a modified version of octree 1 that uses nodes that are larger then they would usually be. Child nodes then overlap so objects will be more likely to fit into one of a node's child nodes. This way objects will be kept lower in the tree and less objects will be found to be visible when they are not. With this larger tree with overlapping nodes it does take longer to find the visible meshes. Updating the tree when objects move will also take longer when the tree is larger. In order to avoid the tree getting too large a maximum depth can be set. The extra size of the nodes is a fraction of their normal size (the size they would be if it were a standard octree). A size divisor value can be set as a parameter of this tree to determine the extra size of the nodes. The normal size of the octree node divided by this size divisor is added to its volume at both sides of the volume, meaning that the final size of the node's volume is its normal size plus two times that size divided by the size divisor.

Object depth

While in octree 1 any object can be as high as the root of the tree or as low as the smallest node that may contain it, the overlap in nodes in octree 2 means that some objects can always be moved down from a certain depth.

During many tests the size divisor value that was used was 2, making every node twice as large as it would ordinarily be. This means that the full

size of a node's children equals the node's own normal size. In this situation an object can always be moved down the tree while its size is less than or equal to the normal size of the next node, regardless of its exact location, as even if the center of the object is on the border of the node's normal volume, the extra size will be enough to contain the object. When for the first time while moving down the tree a node is encountered of which the normal size is not larger or equal to the object's size, the object may still be moved down if it is sufficiently close to the center of that node. This is always possible as, if the current node's normal size is larger or equal than the object's size, the full size of its children will be large enough to contain the object as well. Moving down yet further is always impossible as that node's full size will equal that normal size that was found to be smaller than the object. Thus depending on its exact location any object will be at one of two depth levels in this tree.

If a larger size divisor value is used, the extra size will be small relative to the node's normal size. In this case objects can get stuck high in the tree when close to border of a node's normal volume, while if closer to the center of lower nodes it may still move down more than one level. Thus in this case an object may be kept in more that two different levels depending on its location. This is the case with octree 1, which could be seen as a version of octree 2 with an extra node size of 0.

If a size divisor smaller then 2 is used this results in nodes much larger than their normal size. In such a tree it is possible that a certain object can always be moved down from a particular depth because of the large extra size of the children at that depth, while it can never be moved down further as those nodes will be too small to contain the object. Thus in this situation objects may exist that will always be kept the same depth.

In the case that an object of a particular size can only be kept at a limited number of depths, this can still change when the object rotates, as when an object is rotated the size of its axis-aligned bounding box may change.

Tree Updates

Updates on this tree are done as with the octree 1. When an object moves out of the volume of the node it is kept in, the object is moved up in the tree until a node is encountered that he object does fit in, from where the object is then moved down again as far as it can. Note that while moving up in the tree, the normal volume of the nodes is used instead of the larger volume used in this octree, as this will often allow the object to move down further to a lower level of the tree. As with octree 1, when objects move an update on the tree may be required to move the object down if this is possible.

3.3 Algorithm: Octree 3

3.3.1 version 1

This algorithm uses an octree that has a limited number of object associated with every node. This octree avoids keeping many objects in high level nodes while still keeping every object in only one node. Instead of using the object's bounding volumes to insert them into the tree, insertion in this tree is done using the object's location, while the size of a subtree's bounding volume is kept separately using the bounding volumes of the objects in the subtree. Thus the child nodes of one node may have gaps between their bounding volumes or these volumes may overlap while the point that defines the location of an object is always within the node's proper volume. Many meshes in a node results in a smaller tree that can be updated faster, but which will determine many object to be visible when they are not. Few meshes per node will result in a smaller number of meshes being drawn, but updates will take longer. Another way the number of meshes found visible by this method can be decreased is by forcing the highest levels of the tree to be empty. A start depth can be set from which the nodes will contain meshes, higher nodes will always be empty.

Tree updates

Updating this tree involves updating the node's and parent node's bounding volumes and moving an object up in the tree if necessary. The object will then be inserted in the smallest node that contains its location and has less than the maximum number of meshes per node. This update method is relatively complex; the bounding volumes need to be updated every time a mesh moves to another node but also when a mesh's movement causes the bounding volume of its node and that if its parent nodes to change.

This update can be made faster by not always shrinking the bounding volumes when this is possible. Not only will this save the time of shrinking a node's bounding volume (and possibly that of its parents), it also means the bounding volumes will not have to grow as often. The disadvantage of this is that the meshes in a node may seem visible when actually the node's bounding volume should be smaller. Thus a shrinking step is done during visibility determination instead of during the tree update step. With this update method the movement of a mesh will lead to an update of the tree less often, while the part of the tree corresponding to the visible part of the scene will remain updated as necessary. The visibility determination step will take more time with this method, but since the meshes are still always

updated when moving to a different node, this shrinking step should never take too much time.

3.3.2 version 2

A different version of this algorithm(3b) was made that allows for faster updates. This tree does not keep an exact bounding volume for every node, instead every node only keeps track of the size of the largest meshes in its subtree, keeping a minimum and maximum value in every dimension. The bounding volume of a node is then determined as its normal volume with added to it these size values. This way the tree does not need to be updated as long as meshes stay within the volume of their nodes, making the updates faster, however this tree will perform worse at visibility determination.

3.3.3 version 3

A third version of this algorithm(3bR) is similar to 3b but has been somewhat simplified. Instead of keeping minimum and maximum values in every dimension the nodes in this tree only keep the radius of the largest object (the radius of its bounding sphere) in its subtree. This method is used for scenes with many rotating objects. As these meshes constantly decrease and increase in size in particular dimensions, they might cause updates to a subtree on every frame in the other versions of this tree. This simplification to the update process would make this tree faster to update in general, while performing worse on visibility determination.

3.4 Algorithm: Octree 4

In octree 3 the size of the nodes is dictated by the meshes kept in them while meshes are kept in the first available node. In octree 4 the size of the nodes depends on their depth, with meshes kept at the depth of the smallest nodes that may contain them. This means that a particular mesh should always be kept at the same depth in the tree. As with octree 3 the meshes are inserted using their location, while the depth that the object will end up at is determined by its radius still being small enough to have the whole mesh fit in the node. Object do not need to be moved up and down as often in this tree as in octree 1 as they always stay at the same depth, furthermore this tree does not have the limitation of having nodes with a fixed object capacity like octree 3.

The size of a node is determined as its normal size plus one value that is added to the size in each dimension twice, as this value represents how much of an object is allowed to be outside the normal volume of the node while the object's center is within the normal center of the node. This value is obtained by taking the largest value in the two or three dimensions of the node size and dividing it by a size divisor value that can be set during the creation of the tree. This means that this extra size for each node is half of that of its parent and that if one mesh is twice as large as another it will be placed one level higher in the tree. A maximum depth can be set for this tree that prevents that very small objects are placed in very deep nodes that are smaller than is useful for visibility determination. This depth limit together with the size divisor value also allows the tree to be arranged such that updates will be fast for a particular scene, while the performance of visibility determination is still good.

While this is how the node size is determined in this implementation, different ways of determining node size could be used that will result in objects being kept at different depths.

This algorithm is somewhat similar to octree 2 in which there is also a limited number of depths at which some object may be kept. A difference between the trees is that octree 2 checks object's bounding volumes while octree 4 only checks object locations, what is comes down to is that in tree 4 objects are moved down to the smallest node that will contain them regardless of the object's exact location within the node, while in octree 2 objects may be moved down further if at their current location a lower node will also contain it. This means that in octree 4 objects may sometimes be kept higher that they would be in octree 2, which is bad for visibility determination, but it means less updates need to be done and the depth of any particular object will be known from the start regardless of their exact location or rotation.

Although every node has a depth at which it should usually be kept, when a new node is created during insertion of an object in an (sub)tree, this mesh will be put in this new node instead of more nodes being created on a path down to the proper depth for that mesh. When during visibility determination a leaf with one mesh is found, the visibility test is done with the bounding volume of that mesh, rather than the volume of the node. This ensures that objects placed higher than they should be are not found to be visible more often then necessary. When during mesh insertion a leaf node with one mesh is encountered with that mesh being higher in the tree that it should be, that mesh is put at least one level lower to ensure that a mesh can only be kept in a higher node than it should be kept in when it is the only mesh in a leaf node and visibility determination can be correctly done. While kept in such a high node the mesh is less likely to move outside of the

node's volume and require an update on the tree.

3.5 Optimizations on several trees

3.5.1 Depth reduction

In any algorithm that keeps objects at all time in nodes small enough to allow for effective visibility determination, regular updates will be required. In practice many of the updates will not have been necessary as the object already requires another update before that part of the tree was used. To get a significant improvement in performance it would be useful to reduce the number of updates done. This could be done by keeping objects in higher, larger nodes for most of the time, while still being kept in low nodes when this is useful for visibility determination.

This can be done by moving objects down to an appropriately small node only when their visibility is being considered. If this would require a significant part of the octree to be created during visibility determination this would take too long, however moving objects just a few steps down in the part of the tree of which the visibility is being considered may not take much time.

On octree 4

Octree 4 is particularly well suited to such a optimization as in this tree every object has an proper depth at which it should be kept in the tree. A reduction to this depth can be set that will be used whenever a mesh is moved. This means that when a object moves out of the volume of its node it may only be moved up in the tree to the larger node that it does fit in, or otherwise not be moved down at much as it usually would when reaching that node. During visibility determination if a node is only partly visible then the objects in the node that are supposed to be lower in the tree are moved down one level, however if the node was already found to be completely visible the objects can instead be marked as visible without being moved in the tree. This means that when using a higher depth reduction (DR) the number of nodes that need to be visited will often be less. The time that visibility determination takes will increase from the objects that need to be moved down, but at the same time the smaller number of nodes that is visited will make it faster.

On octree 3

Although this methods was originally only used for octree 4, a similar optimization can be done on octree 3 using the start depth that can be set to affect the visibility determination performance of this tree. Beyond this start depth any objects is allowed at any depth, however objects can also be put some levels above this depth using a depth reduction variable. During mesh insertion and tree updates caused by moving objects the reduced start depth is used, while during visibility determination objects too high in the tree are moved down if the node is only partly visible.

Other algorithms

Octree 1 and octree 2 do not implement this optimization as the object in those trees do not have a clear depth at which they should be kept in the tree. An implementation of this optimization on these algorithms would therefore probably be more complex, thus this method is only tested on octree 3 and octree 4 in this project.

3.5.2 Object checklist

When an object moves it may leave the volume of the node it is kept in in the octree, which then requires an update. This means that when an object moves it needs to be checked; that is its location or bounding box needs to be checked against the node's volume. If a maximum speed is known for the object it may be known that the object will not move out of a node's volume for a certain number of frames. In that case no check needs to be done on this object for some frames.

This has been implemented in some of the algorithms. After every time an object is checked it is determines what the minimum distance between the object and one its node's bounds is. The object is then set to be checked again only once in may have reached this border. Exceptions are when the object is moved down in the tree during visibility determination if depth reduction is used or when an object jumps from one location to another, further than its usual maximum speed. In these cases the object is set to be checked during the next tree update.

This optimization works well together with depth reduction, as when objects are kept in large nodes they will often be very far away from the node's borders.

Implementation

This optimization has been implemented as a number of lists of objects that should be checked at a certain iteration. Every iteration the first of these object checklists (OCL) is used and deleted while the other lists are shifted one iteration forward and a new, empty list is created for the furthers iteration for which updates are being considered. This means a lot more work needs to be done for every check when mesh checklists are used then when they are not. Thus using mesh checklists is only beneficial when it does result in significantly fewer checks. The implementation of the octrees allows also to be used without these lists.

Only octree 3 and octree 4 implement this optimization. In octree 1 and octree 2 object are moved down when they can, thus the tree requires updates when an object leaves a nodes volume or when it becomes completely contained in the volume of a lower node. This would make determining the minimum distance the object may move to require an update take more time while the distance will often be smaller. Furthermore as those trees do not have depth reduction many objects will be kept in smaller nodes.

3.6 Overview

This section provides an overview of the used methods and their designations as used in the test section of this report.

Octree 1 Basic octree, objects as deep as possible.

Octree 2 Octree with larger nodes, node size multiplied, objects as deep as possible.

Octree 3 Limited number of meshes per node, objects put in first node with room left, node size determined by meshes in subtree, node size partly updated during visibility determination.

Octree 3b Limited number of meshes per node, objects put in first node with room left, node size that of normal octree plus size of largest object in subtree.

Octree 3bR Limited number of meshes per node, objects put in first node with room left, node size that of normal octree plus radius of largest object in subtree.

Octree 4 Node size normal plus fraction of size in longest dimension added in each dimension, object location within normal octree node size, object depth lowest possible while object radius not larger than extra node size.

Chapter 4

Testing methods and setup

4.1 Definitions

Some definitions of some of the terms used in the next chapters.

Move time (MT) The time it takes to determine the proper location in the scene for every object. This does not include any processing on the data structure yet.

Update Time (UT) The time the update of the octree takes when objects have moved. This includes checking whether objects are still in the right node in the tree and making all the changes that are required to put all objects in the right node.

Visibility determination time (VT) The amount of time it takes to determine which objects may be visible according to the algorithm's visibility determination method. This means traversing the tree starting at the root, visiting all nodes that need to be visited according to the algorithm and adding the found objects to a list of visible objects.

Render time (RT) The time it takes to send the object data to the GPU as well as the time it takes the GPU to render the final image.

Tree traversal steps The number of steps taken through the tree, only while updating the tree. This amounts to the number of times a higher or lower node is checked in order to find the node an object would be moved to.

Tree updates Number of tree updates. One tree update is when one objects needs to be kept in another node after it moved through the scene.

Node visibility checks The number of times it is determined if a node's objects are visible. This includes nodes that are already known to be visible because their parent node was found to be completely visible.

4.2 Test Scenes

The data structures are tested on two scenes. One of this scenes is a city scene. In this scene there are a number of city blocks with between them roads with cars and sidewalks with pedestrians. The people in this scene are composed of several animated objects. This scene has objects of different sizes moving at several different speeds. For the tests this scene is usually generated with 250000 objects. Figure 4.1 shows a screenshot from this scene, from the level of the viewpoint during the tests. Figure 4.2 gives a better view of what the scene looks like. The 250000 objects of the tests results in a city scene of 19 by 19 blocks.

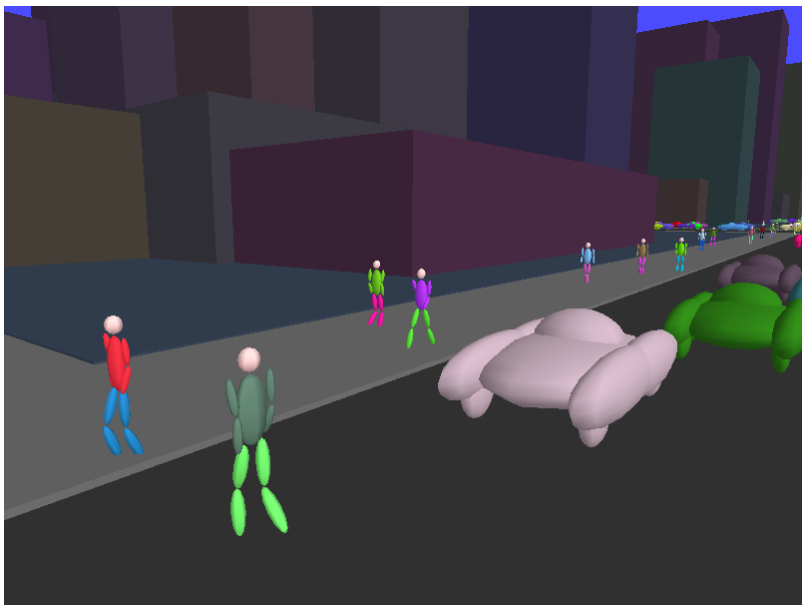

Figure 4.1: City scene.

The other test scene is a cloud of spheres of which half move randomly. Some spheres are larger than others, the largest spheres are 4 times as large as the smallest ones. All moving spheres move at the same speed. This is a simpler scene but it is less flat and may make better use of an octree. This

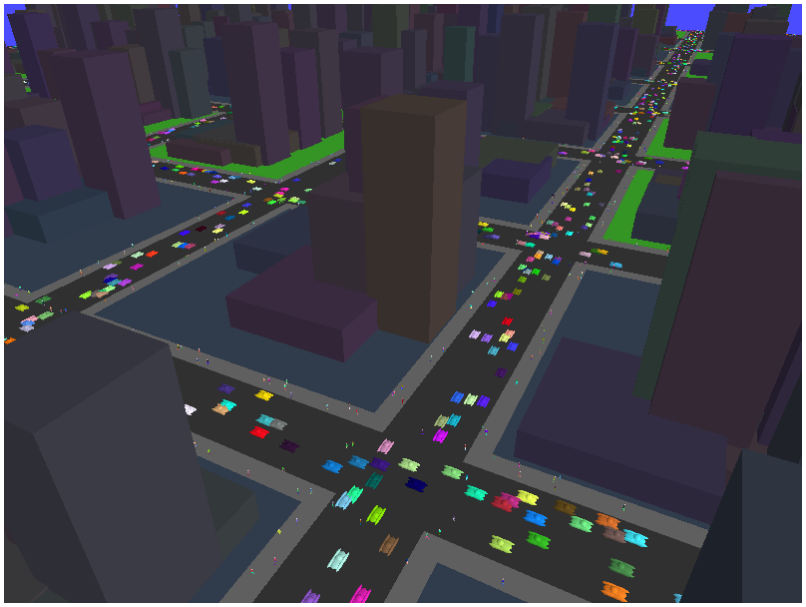

Figure 4.2: City scene.

scene is usually generated with 200000 objects for the tests. Figure 4.3 shows a screenshot from one of the tests on this scene.

In both scenes the viewpoint starts close to the center of the scene. As the performance of some of the tested methods depends on what part of the scene is visible, the viewpoint moves through the scene. The viewpoint moves along a straight line while rotating.

4.3 Testing methods

The scenes can be generated with any number of objects. Then the data structure is initialized and the initial tree is build. After that the main loop is started which executes these steps for a certain number of frames:

Moving objects The locations and bounding boxes of the objects and the viewpoint are updated.

Updating data structure It is determined if any of the objects should be kept in different nodes and the tree is updated accordingly.

Determining visible objects Frustum culling is used to determine the objects that should be drawn from the current viewpoint.

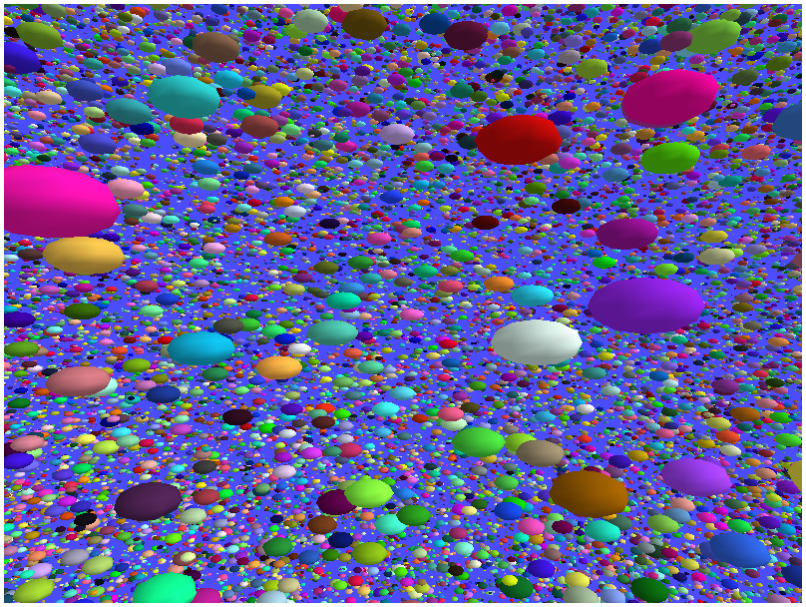

Figure 4.3: Sphere scene.

Rendering The visible meshes are rendered using OpenGL. (optional)

The time that each of these steps takes is measured and averaged to get the results of the test. Rendering can be done during the tests but it can also be turned of, so many tests can be done somewhat faster.

The main measure in these test is the time it takes to update the data structures. However the time it takes to find the visible objects is also important as some of the methods do some of the work during this step. Furthermore fast tree updates are not very useful if visibility determination takes a long time on that tree. Thus in practice the goal is to minimize the sum of the visibility determination time and the update time. The number of meshes found is also important to note as a data structure that can be updated fast is only useful if it still performs well on visibility determination. The methods have some parameters that can be adjusted to change the performance of the tree. These will be set so that the tree's performance is as good as possible while still performing well on visibility determination.

Visibility determination is done using frustum culling with a distance limit. Octrees that can be updated fast are also useful for more advanced methods of visibility determination (see chapter 2), but for this project frustum culling is enough to make sure the quality of the octree is sufficient.

Chapter 5

Tests

5.1 Visible mesh count

Every method has some parameters that can be used to find a balance between good visibility determination performance and fast updates. To be able to compare the update times of the octrees the parameters are set to find a number of visible meshes that is about 1.1 times the minimum number of visible meshes, as determined by checking every mesh against the view frustum. This number is not a strict maximum as parameters like maximum depth do not allow for small changes.

Figures 5.1 and 5.2 shows the number of visible meshes found using the different algorithms on the two scenes. For most methods this number is around 1.1 times the actual number of visible meshes, however octree 1 always finds too many meshes regardless of its maximum depth setting. This is because objects stay in high nodes when they do not completely fit into one of the node's children.

This is demonstrated in figures 5.3, which shows a top down view of the sphere scene with the visible meshes as determined by octree 1. The viewpoint is in the middle of the scene. Note that this means the viewpoint is on the border of many of the node's in the bottom half of the screen, if the viewpoint was moved forward only a little, many of those meshes would not be visible. During the test the camera moves instead of standing still in the center of the screen. Figure 5.4 shows the same situation only with just the actually visible meshes drawn. The visible meshes found using octree 4 are shown in figure 5.5, in which about 1.1 times as many meshes are shown as in figure 5.4, which was the goal for this test.

It seems that octree 1 without any adaptations to make it better at visibility does not perform well enough for it to be possible to compare its update

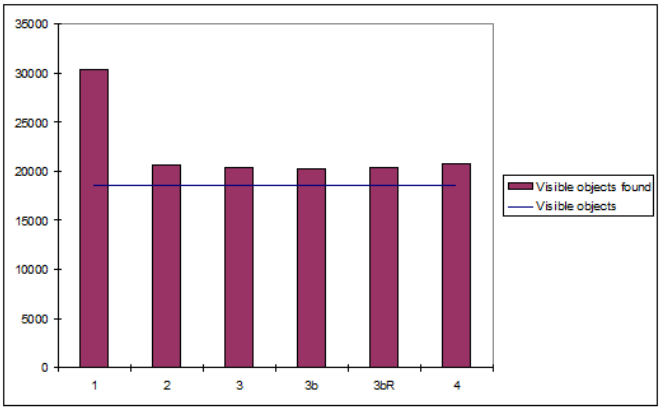

Figure 5.1: Number of meshes found visible and actual number of visible meshes in the sphere scene.

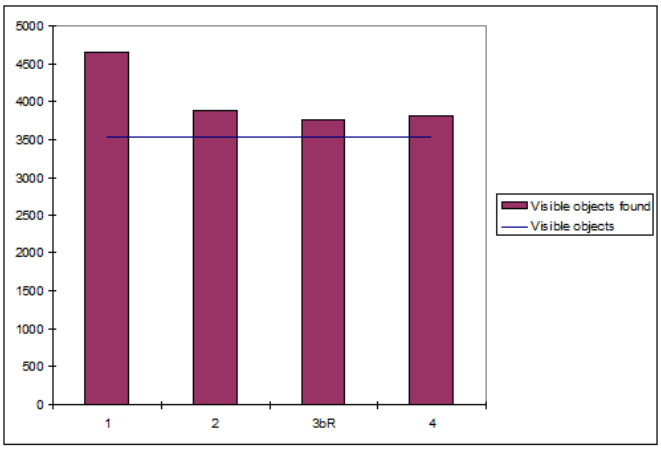

Figure 5.2: Number of meshes found visible and actual number of visible meshes in the city scene.

performance with that of the other methods. Octree 2 is a modified version of octree 1 that uses larger nodes, which allows it to find much fewer visible meshes.

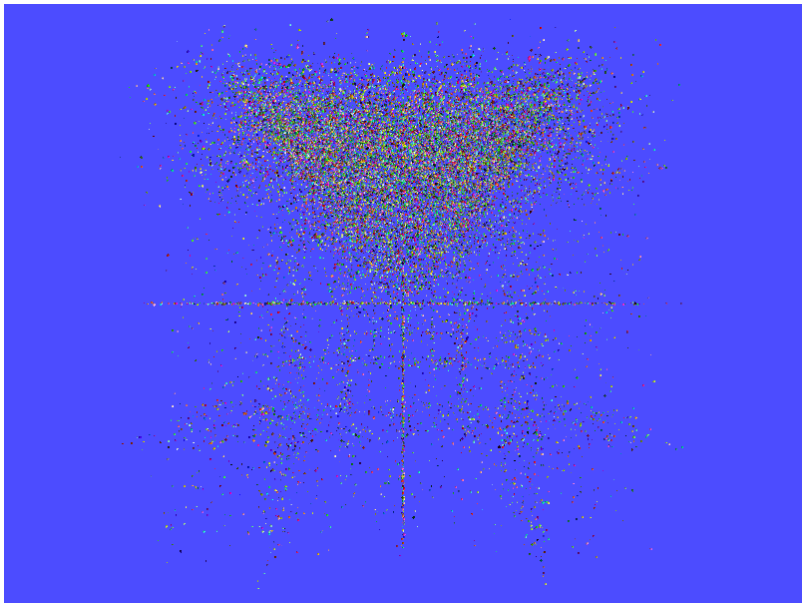

Figure 5.3: Top down view of the sphere scene when using octree 1.

5.2 Tree updates

5.2.1 Test results

Figure 5.6 shows the performance of the algorithms on the sphere scene. The parameters used for these tests were:

1 Maximum depth: 8.

2 Maximum depth: 7, size divisor: 3.

3 Meshes per node: 2, start depth: 5.

3b Meshes per node: 2, start depth: 7.

3bR Meshes per node: 8, start depth: 7.

3bR DR/OCL Same, but uses depth reduction of 3 and OCL.

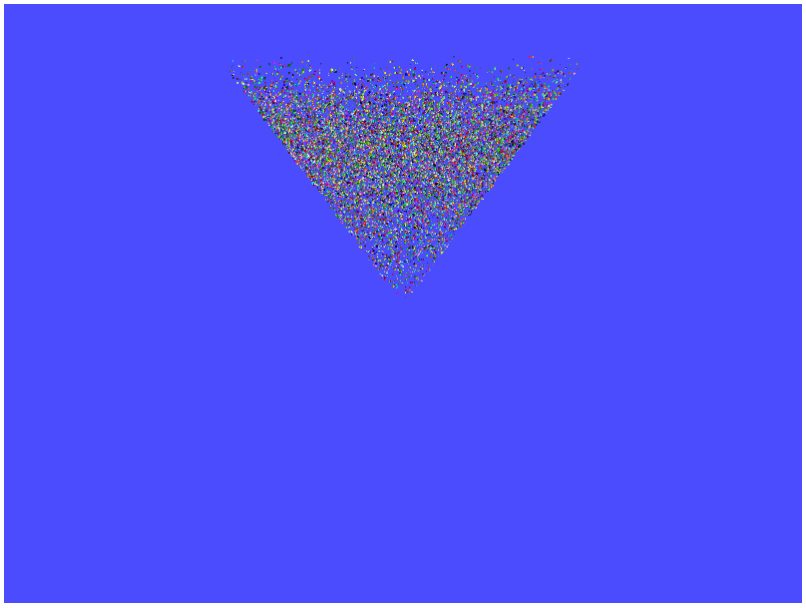

Figure 5.4: Top down view of the actually visible meshes in the sphere scene.

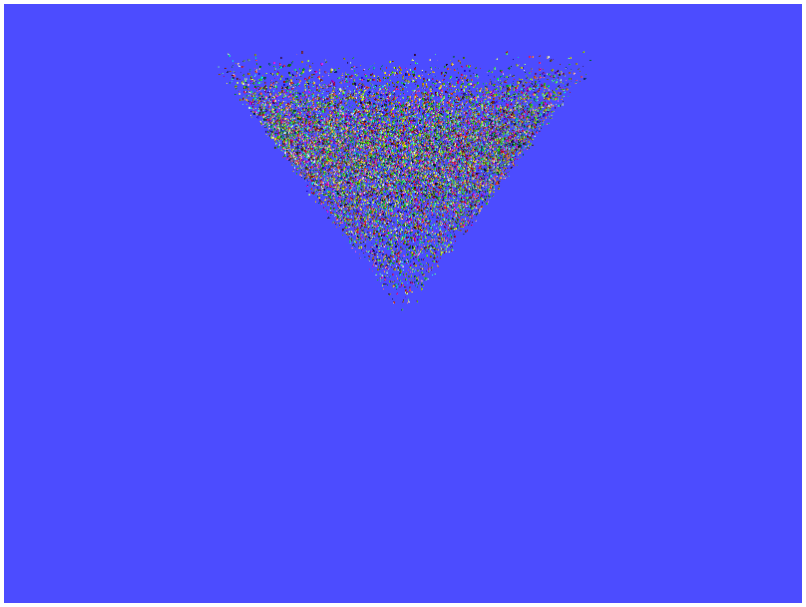

Figure 5.5: Top down view of the sphere scene when using octree 4.

4 Max depth: 6, size divisor 6.

4 DR/OCL Same, but uses depth reduction of 3 and OCL.

Figure 5.8 shows the performance of the algorithms on the city scene. The parameters used for these tests were:

1 Maximum depth: 10.

2 Maximum depth: 9, size divisor: 2.

3bR Meshes per node: 168, start depth: 9.

3bR DR/OCL Same, but uses depth reduction of 3 and OCL.

4 Max depth: 10, size divisor 2.

4 DR/OCL Same, but uses depth reduction of 4 and OCL.

In figures 5.7 and 5.9 some statistics about some of the trees are shown. Every value of octree 2 has been set to 100 so that the different statistics can easily be shown and compared in one graph. Shown are the number of times than an object moved and was found to still be contained in its node in the tree, the number of nodes in the tree at the end of the test, the number of steps up and down the tree that were done during the updates on the tree, the number of times the tree needed to be updated because an object had left the volume of its node and the number of nodes that were visited during visibility determination.

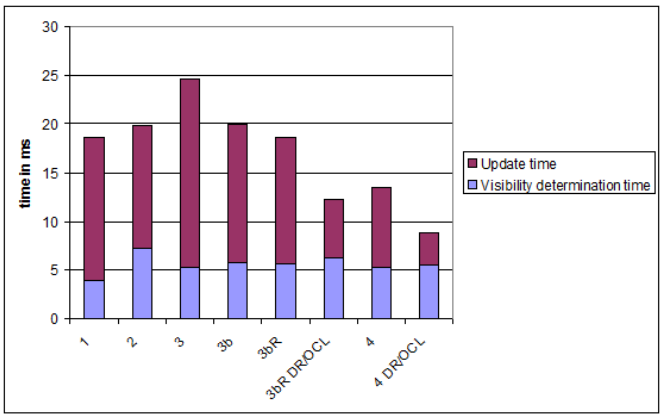

Figure 5.6: Update and visibility determination times of several methods on the sphere scene.

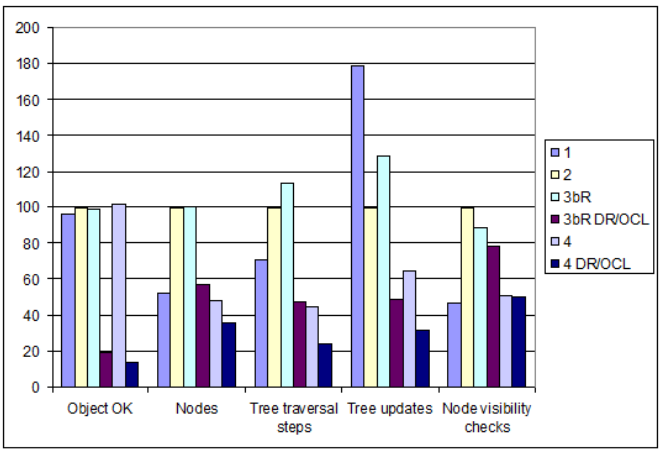

Figure 5.7: Statistics of several methods on the sphere scene.

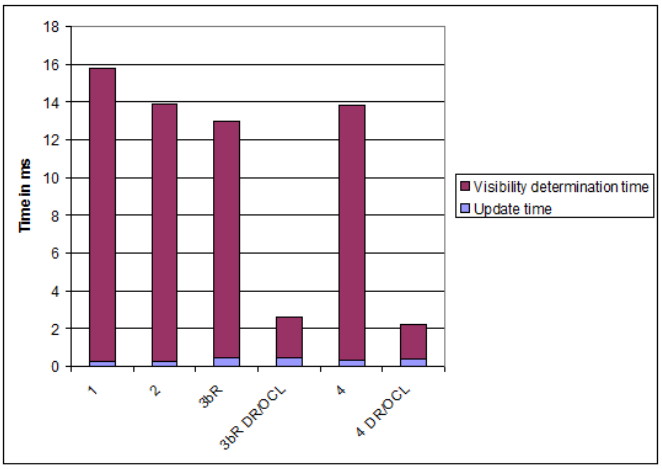

Figure 5.8: Update and visibility determination times of several methods on the city scene.

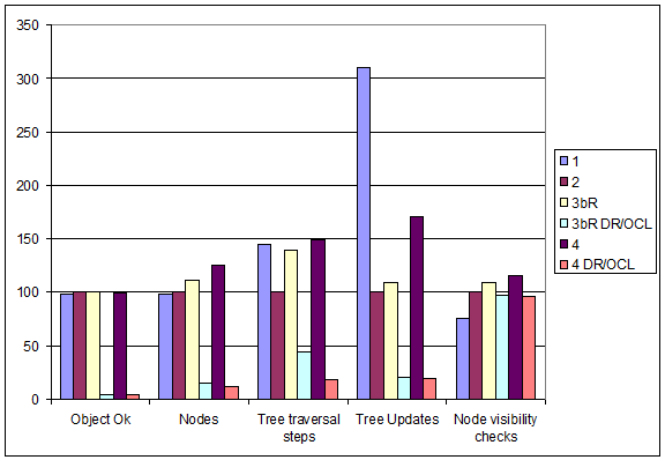

Figure 5.9: Statistics of several methods on the city scene.

5.2.2 Result analysis

Sphere scene tests

Octree 1 Octree 1 takes the least amount of time to find the visible meshes, which is to be expected as that octree is smaller than the others and finds many more objects than the other methods. Still octree 1 takes more time to update than octree 2. When comparing the statistics of the two trees it can be seen that while octree 1 is smaller and less steps through the tree are taken to update it, the number of times that an update needs to be done is much higher than it is for octree 2. This is because in octree 1 when an object leaves the volume of a node it will usually be only partly in another node at the same depth, meaning that whenever an object has to be updated it initially ends up higher in the tree. Eventually the object will be able to move down the tree again, meaning that several updates are done for every time an object leaves a node. The other trees avoid this by somehow putting an object in a node that it will probably be able to stay in for some time after it leaves the volume of its node.

Octree 3 Octree 3 is slower than any other, even though its visibility determination takes less time than most other trees. The reason octree 3 performs well on visibility determination is because its nodes are only as large as they need to be to contain the objects in their subtree. This is what allows octree 3 to put objects in its nodes starting at a depth of 5 rather than 7 like the other versions of octree 3, because the visibility checks are done on smaller,

more accurate volumes the number of meshes found with this octree will be relatively small at a given start depth. This tree checks and updates the volume of the nodes in part during visibility determination, which does not seem to slow it down much. The complicated updates on the node's volume do make this tree take very long on tree updates, node volumes may need to be updated even when meshes stay within the node's normal volume.

In octree 3 only 2 objects can be kept per node because it starts keeping objects in nodes relatively high in the tree. Allowing it to keep more object in a node means too many will be kept at such high levels and thus found to be visible when they are not. In order to get good visibility determination performance octree 3b and octree 3bR can only keep objects starting at depth 7, which is deep enough that nodes will be small enough so that there will not be many objects in one node regardless of the node's capacity. Octree 3b therefore has a node capacity of 2, more would only waste space. Octree 3bR on the other hand has depth reduction implemented in it, making it beneficial for nodes to have a larger capacity so that many objects from different low level nodes may be kept in one higher level node.

While octree 3b and octree 3bR have a start depth of 7, octree 2 uses a maximum depth value of 7. In all these octrees most objects are probably kept at that level. In octree 2 however some objects may be kept slightly higher while in 3b and 3bR some are kept slightly lower. At that depth the nodes of octree 2 will be slightly larger, as the nodes in octree 3b and octree 3bR are only as large as they need to be for the objects kept in those nodes, while the extra size of the nodes at depth 7 of octree 2 is apparently large enough to allow objects in them. This means octree 2 takes somewhat longer at visibility determination, for which it visits more nodes, while updating a little faster, as object are slightly higher in the tree, and have more room to move within their nodes. The difference in performance between these tree is however not very large. Octree 3bR is slightly faster then octree 3b, which is likely because its updates are simpler.

With DR and OCL octree 3bR updates takes less than half the time they otherwise take. Visibility determination is slightly slower but overall this method works quite well.

Octree 4 Octree 4 is about one third faster on this scene than octree 2. The reason for this is in part that objects in octree 4 may be kept in higher nodes that usual if they are the only mesh in that node's volume. To prevent these meshes from being found visible too often the visibility check on such a node is done on the object's volume instead of on the node's volume. Although meant mostly for those meshes that are higher in the tree than they should

be, this check on only the object's volume also decreases the number of visible meshes found in general. This allows this octree to work well with a maximum depth of only 6 and nodes that are smaller that those of octree 2.

Using DR and OCL on octree 4 makes it yet a lot faster. The updates on the tree take on average only 3.33 ms, which is about one fourth of the time the updates take on octree 2. Visibility determination takes slightly longer then when not using depth reduction, but it still takes less time than with most other octrees. Together updating the tree and determining the visible meshes takes about half the time it takes with octree 2.

DR/OCL The result of depth reduction on both octree 3bR and octree 4 is that the tree is smaller and requires less updates. The number of nodes visited for visibility determination is also slightly lower as when objects are kept higher in the tree and nodes are completely visible, creating and visiting lower nodes is not necessary. Using object checklists means that checks on objects need to be done less often, especially for objects kept in high nodes. This can be seen in the 'Object Ok' statistic in figure 5.7 which shows the number of times a check was done on an object when an update was not necessary.

City scene tests

The tests on this scene were done with quadtree version of the methods rather than octrees as it is a very flat scene for which a quadtree is better suited than an octree. Another significant difference is that this scene is much bigger and less objects are visible at once. The result of this is that visibility determination is much faster on this scene than on the sphere scene, as can be seen in figure 5.8. The higher depth limits used on this scene is also a result of that; as these are quadtrees the number of nodes in the trees is no more for this scene than for the sphere scene.

Octree 1 performs relatively well on this scene. This tree is no smaller than the other trees, with some object being kept very far down in the tree. Those objects that end up higher in the tree still keep it from performing well on visibility determination. As before when an object leaves a node it often requires several updates on the tree, making updates on the tree slow.

The performance of tree 2, tree 3bR and tree 4 is very similar on this scene. They all keep the meshes at about the same depth. For tree 2 this means a maximum depth of 9, resulting in slightly fewer nodes and updates for this tree. Tree 3bR keeps nodes starting at depth 9, but keeps many objects per node is this scene. This large number of nodes is once again useful when using depth reduction, while without DR it results in all objects being kept

at depth 9. Octree 4 requires a maximum depth of 10 on this scene. This small difference may be because this tree keeps some of the larger objects in the scene, like cars and the static parts of the scene, in higher nodes than tree 2, which keeps them as far down as possible. To compensate for those objects being visible too often the smaller meshes in the scene may be required to be kept deeper in the tree.

DR/OCL Using depth reduction and object checklists makes trees 3bR and 4 much faster in this scene. The visible object determination does not take much time on this scene and keeping meshes higher in the tree does not change that, while the objects in higher nodes require much fewer updates. As this scene is very large many of the objects are never visible, which makes depth reduction work very well on this scene. For tree 3bR the start depth is reduced by 3, and the number of meshes set to 168, resulting in most meshes being much higher in the tree than they would usually be. For tree 4 the depth reduction is 4, but this is with a maximum depth of 10 at which many objects would otherwise be kept in this tree.

5.3 Results

In table 5.1 the time that the steps took is shown. Also shown is the percentage increase in performance of the tree operations, being any necessary updates on the tree and visibility determination. This performance increase compares the times for the algorithms with octree 2, the most standard octree implementation with comparable visibility determination performance.

Algorithm	MT	UT	VT	RT	total	%tree	%total
1	70.00	15.6	0.26	29.96	115.81	-14.84	3.47
2	69.41	13.52	0.28	36.75	119.98	0	0
3bR	69.8	12.51	0.45	36.19	118.95	6.14	0.85
3bR DR/OCL	69.57	2.13	0.49	35.85	108.04	81.02	9.94
4	70.50	13.75	0.37	35.76	120.38	-2.26	-0.33
4 DR/OCL	70.35	1.87	0.41	35.53	108.16	83.47	9.84

Table 5.1: The times that the different steps took with the algorithms on the city scene. As well as percentage improvement over octree 2 All time values in ms.

The fastest algorithm is octree 4 with depth reduction and using object checklists. This algorithm is 83.47% faster than octree 2 on the octree operations. The performance increase for the whole rendering process is 9.84%.

More time is spent on the object movement step than on any other. To improve the total processing time of this implementation some improvements to this step would be possible. The animated limbs could be processed a lot faster if the animation was only done for nearby objects or made much simpler by using a larger bounding volume that contains the small movement of the limbs. However for this implementation it was more interesting to calculate the smaller bounding volumes of the limbs as this makes more use of the data structure. The object movement step could also be made much faster if done multithreaded.

Chapter 6

Conclusion and Future work

6.1 Conclusion

A standard octree and a modified version of this octree that finds fewer visible meshes, as well as two alternative octrees were tested on two scenes. The tests show that the trees may be updated faster on dynamic scenes by keeping objects higher in the trees and lowering them if necessary during visibility determination. For this a proper depth for objects has to be set which can be done in different ways. When kept in such a high node it may not be necessary to consider an objects at all for some number of frames if something about the object's speed is known. By using octree 4, there was a 83.47% improvement over the standard octree algorithm with respect to stages 2 and 3 of the graphics pipeline. This corresponds to an improvement of 9.84% for the entire pipeline.

Although otherwise the performance of the octrees was fairly similar, it does seem that some performance gain may be achieved in several other ways. Keeping track of node size depending on the objects in a subtree can make nodes smaller and visibility determination faster. Determining the depth an object should always be kept at may reduce the number of updates as objects are immediately moved to a proper node when they require an update, as well as allowing for other optimizations on the tree, and objects do not need to be moved down from a leaf if any required visibility tests can be done on the object's volume.

6.2 Future work

Different node sizes for octree 4 could be tried. It may in some scenes make more sense to put more objects all at the same depth or it may be possible to

keep track of the largest object in a subtree of octree 4, as is done in versions of octree 3. This would make the high level nodes smaller and make visibility determination faster on this tree.

In these tests only octree 4 did visibility tests on leaf nodes on the volume of the object if there was only one object in the node. This may be a good thing to do in other octrees, as the number of visibility tests stays the same while the results are more accurate.

Depth reduction of objects was only done on octree 4 in which the depth of objects was known and on the start depth of octree 3bR. It could also be implemented in other octrees by reducing the maximum depth that is used during tree updates.

Bibliography

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina, 1990.

[2] H. C. Batagelo and S.-T. Wu. Dynamic scene occlusion culling using a regular grid. *Graphics, Patterns and Images, SIBGRAPI Conference on*, page 43, 2002.

[3] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[4] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9:412–431, 2003.

[5] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 231–238. ACM, 1993.

[6] D. Luebke and C. Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, I3D '95, pages 105–ff., New York, NY, USA, 1995. ACM.

[7] I. Pantazopoulos and S. Tzafestas. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent & Robotic Systems*, 35:123–156, 2002.

[8] J. Shagam, J. Pfeiffer, and Jr. Dynamic irregular octrees, 2003.

[9] O. Sudarsky and C. Gotsman. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics*, 5:13–29, January 1999.

[10] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, pages 61–70, New York, NY, USA, 1991. ACM.