View Frustum Culling with Octree

Raka Mahesa 13508074
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
if18074@itb.ac.id

***Abstract*—Many people are mesmerized by the beautiful world of Pandora in the Avatar movie, yet few people realize that such beauty is brought by the power 3D rendering. And there are even fewer people who know that 3D rendering is a very complicated matter that keeps evolving. Each day new techniques are being tested to improve or optimize 3D rendering. With this paper, the author tries to dive into the dark, murky water of 3D rendering in order to find out how to optimize 3D rendering. The search for the ultimate rendering technique still goes on…**

***Index Terms*—AABB, Culling, Divide and Conquer, Octree, View Frustum, 3D.**

I. Introduction

Fig. 1. Crysis, a game with a very realistic environment.

Currently, three dimensional (which we'll refer to 3D from now on) applications are common things. We see it in many aspects of our life, from a simple animation in Microsoft Power Point, to an artist modeling a new car, a two hours-long 3D movie, to a beautiful world rendered in a 3D game that can viewed smoothly on high-end computers. Yet, 3D rendering is still an evolving topic, every day new researches and experiments are made to push its boundary. So with this paper, I was hoping to experiment on a technique for 3D rendering so I can contribute to the ever-growing world of 3D rendering.

There are a lot of problem still being researched in the world of 3D rendering such as how to render a dynamic shadow, how to render objects with realistic lighting, etc. However, I'll only write about rendering optimization in this paper instead of including those other stuff, since it is a fundamental aspect for optimizing the whole rendering process, so all those effects like shadow and lighting can be applied even on low-end machines. Even though rendering optimization seems like a simple matter already, it isn't, there are a lot of ways to optimize it, and each optimization has a different way to do it. So instead, in this paper I'll restrict myself to write about optimizing 3D rendering with view frustum culling.

Well, since people have been trying to optimize 3D rendering from more than a decade ago, there are already a few techniques on how one can optimize view frustum culling. Most of those techniques are based on Divide and Conquer, like Portal Rendering and Octree Rendering, two most widely-used view frustum culling techniques. This time, I'll write about how Octree Rendering can be used for view frustum culling, and how to optimize the octree itself.

II. Fundamental Theory

3D rendering is a process mainly done in the video adapter instead of the processor, and we can't really mess with how the video adapter renders as a programmer, so how are we going to optimize it? Well, we're not going to be optimizing the rendering process; instead, we'll be optimizing what the video adapter will render. You see, if we just tell the adapter to render every single piece of object in our 3D world, it'll be a waste of time, since there are some things that don't need to be rendered, like some objects behind our point of view. To only renders the objects in our view is the goal of view frustum culling.

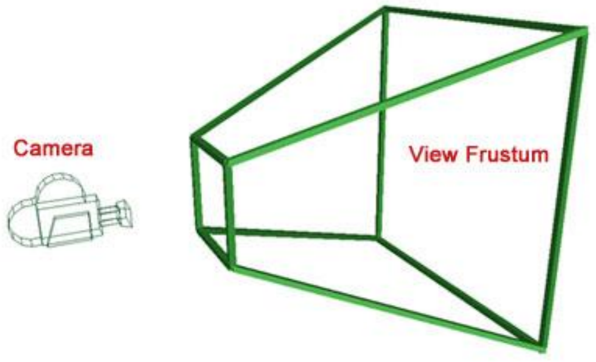


Fig. 2. View frustum.

View frustum is like your field of vision, a region of space in the world that will appear on the screen. Only objects inside the frustum will be seen by you. So, ideally, only objects inside this frustum should be passed to the video adapter and rendered, since rendering objects that wouldn't be seen is a waste of power. This is exactly what view frustum culling does; it culls the objects outside the view frustum, and the rest was passed to the video adapter so it would only have visible objects to render.

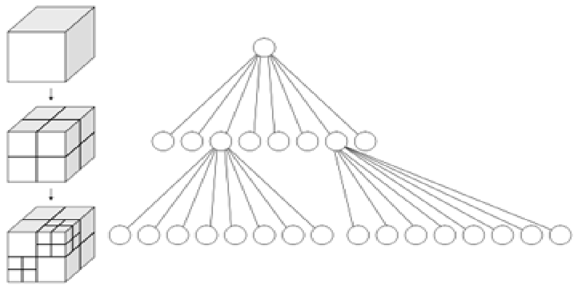


Fig. 3. Octree.

So far we haven't touched the other subject, the octree. Well, as the name suggest, octree is another variant of tree structure, with every node has either eight children or no children at all. However, when we're talking about the 3D world, octree usually refers to a technique that recursively partitions a space by subdividing it into eight equal octants. Data structure of an octree can be seen on the right side of Figure 2, while the left side features how a space is being partitioned by an octree.

Each node of an octree is a cube area, which brings us to our next problem, which is how to represents a cube in 3D space. There are a lot of methods for it, like listing all the points that make a cube, or just save one of the corners position and the cube size. All those methods are all right, however, when we're dealing with culling, we would also deal with a lot intersection test, so we'd like to keep that in mind when choosing our cube representation method.

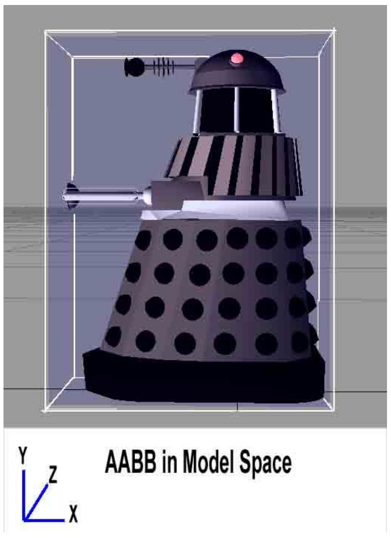


Fig. 4. Axis-Aligned Bounding Box.

Fortunately, since an octree is usually axis-aligned, we do have a way to represents cube that matches the requirement, which is Axis-Aligned Bounding Box (would be abbreviated as AABB from this point on). Axis-aligned means that the object is aligned with every axis, not oriented in any way, while bounding box just mean a box that is used as some kind of boundary. AABB is a structure that has a minimum point and a maximum point of a cube, which makes it easy to test whether it intersects with another AABB or not.

To test whether an AABB intersects with another AABB or no, we would check if it doesn't intersect with another AABB instead. If an AABB, let's name it "A" does not intersect with another AABB named "B", it would fulfill at least one of these requirements:

```
Minimum X of A > Maximum X of B
Minimum Y of A > Maximum Y of B
Minimum Z of A > Maximum Z of B
Maximum X of A < Minimum X of B
Maximum Y of A < Minimum Y of B
Maximum Z of A < Minimum Z of B
```

In turn, failing all these tests would mean that the AABBs intersect with one another. Of course, these same tests can easily be used to test whether a point is inside an AABB or no.

III. Implementation

With all of the fundamental theories covered, it's time to give view frustum culling a try. The easiest solution would be to use brute force algorithm and test each object in the 3D space, checking whether it is inside the view frustum or not. Another solution is to use an octree to quickly eliminate points that are far away from the view frustum and search for nodes that intersect with it.

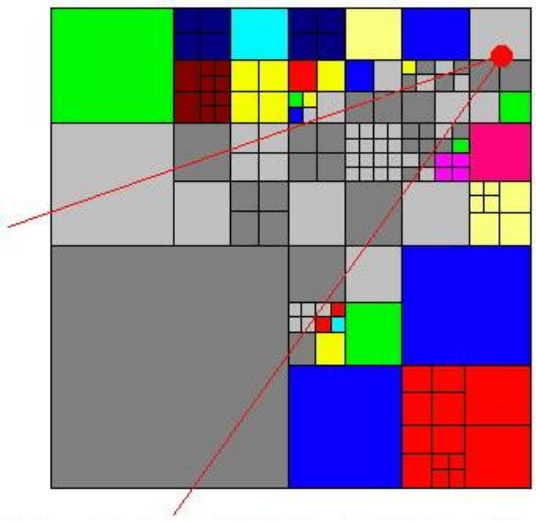
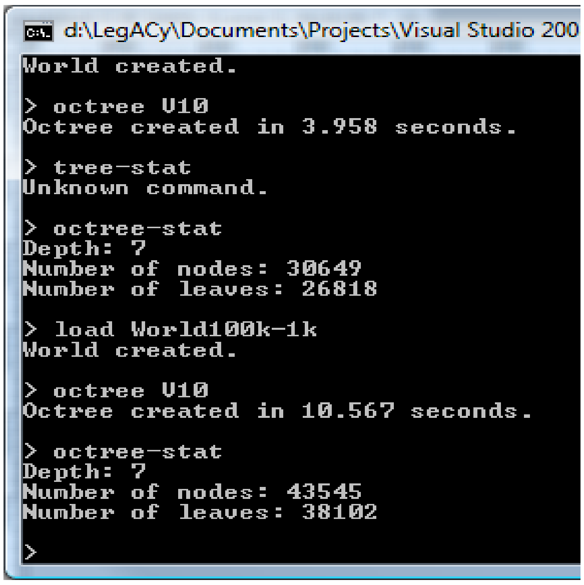


Fig. 5. View frustum culling.

Figure 5 is a planar representation of a view frustum culling with octree. The two red lines represent the shape of the view frustum, with the view point starting from the red circle. All of the colored rectangles is culled and will not be rendered, while the grey rectangles will be rendered.

Note that for the sake of simplicity, I used a cube to represents a view frustum instead of a real frustum so I can easily check whether a node intersects with the frustum or not using AABB intersection detection. Using real frustum would means that the intersection detection algorithm must be changed (frustum intersection test must check intersection against all 6 planes of the frustum) but it would not be much of a difference. Another thing to keep in mind is that I used points for objects in 3D world, while most 3D applications would have real 3D objects. And again, there would not be much of a problem since checking 3D objects would also use AABB intersection test.

Octree is a divide and conquer algorithm, so it would have two different algorithms for dividing and for conquering. The dividing algorithm is the one used when the tree is constructed from a list of points in the 3D space, while the conquering algorithm is the one used in view frustum culling. For the purpose of experiment, I've decided to separate the two algorithms and measure their timing individually.


```
d:\LegACy\Documents\Projects\Visual Studio 200
World created.

> octree U10
Octree created in 3.958 seconds.

> tree-stat
Unknown command.

> octree-stat
Depth: 7
Number of nodes: 30649
Number of leaves: 26818

> load World100k-1k
World created.

> octree U10
Octree created in 10.567 seconds.

> octree-stat
Depth: 7
Number of nodes: 43545
Number of leaves: 38102

>
```
Fig. 6. Screenshot during the octree creation.

Constructing a tree is quite simple. We already have the root node, which is the whole world, now we just need to split it into 8 octants. After splitting into 8 octants, the list of points inside the node should also be splitted according to the space of the octants and inserted to the corresponding octant. For each 8 octant, a new node will be created as a child to the parent node. The process will keep continuing until the number of the point inside the node is low enough, or the size of the node is small enough, or both. When the node creation process has stopped, the octree would be constructed. Below is a table and a chart containing the duration of the tree generation process for different number of points

Table 1. Tree construction durations with minimum points per node is 10.

Points in world	Duration (s)
5.000	0,457
15.000	0,988
40.000	3,000
100.000	6,751

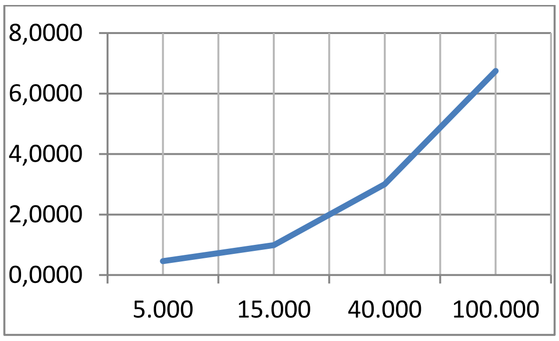


Fig. 7. Table 1 chart.

In this first test, I used a world with width, height and depth of 1000 units and various amount of points. We can see from the chart that the tree construction time increases exponentially. I used 10 as the minimum amount of points inside a node for it to stop creating children nodes. What if I also limit the node by size, will it becomes faster? Well, that's exactly what I did for my next test.

Table 2. Tree construction durations with minimum points per node is 10 and minimum size of 100.000 units.

Points in world	Duration (s)
5.000	0,316
15.000	0,537
40.000	1,316
100.000	3,093

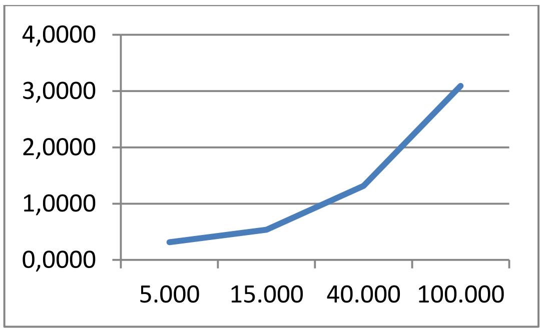


Fig. 8. Table 2 chart.

The second test has yielded its result; I created several octree with minimum number of points of 10, like the last test, and minimum size of nodes of 100.000. It's clear by limiting the size of the node; the tree would be constructed quicker because the depth of the tree is also limited. However, less depth means less accuracy, which in turns, would also means that when the octree is being culled, more points has to be tested to know if they are in the view frustum or not.

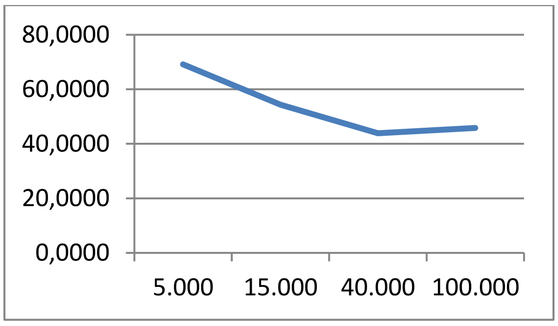


Fig 9. Time-decrease (in percentage) chart.

Another thing I observed is that the bigger the number of points in the world, the decrease in duration is also increasing (don't let the chart fool you, it's a comparison of new/old duration). Unfortunately the chart rebounds at 40.0000 points in the world, we don't know yet whether this is truly a rebound, or just an anomaly.

There's also one more thing that I would like to comment on, it's the amount of time needed to generate a tree. A normal game level would consist of around 5.000 objects, including walls, floors, pillars, etc. Now, 3D games are usually played at least at 35-40 frames per seconds, which would means that each frame has around 0,025 seconds of processing time. From both test I had run before, the amount of time needed to construct an octree from a world with 5.000 points is at least 0,36 seconds, which would mean that octree is much more suited to be generated once when the game level is loaded. It would works fine for static objects, and most games has more static objects than moving, dynamic objects too. However, as games get more sophisticated, the amount of moving objects in a level has also increased due to physic engine being implemented in the game. So another problem has arisen, how to create octree for dynamic objects? We won't discuss this problem further in this paper as it is not the purpose, though I can tell you that one of the solutions for creating octree for dynamic objects is to modify the tree instead of fully rebuilding it.

With octree generation done, it's time to move on to the culling. View frustum culling with octree is even simpler than its creation. Starting from the root, we check a node whether it intersects with the view frustum or not, if it does not intersect, we don't need to bother with its children and stop there. However, if the node intersects, its children would also be checked for intersection if it has any children, though if he does not have any more children, the points attached to the node will be copied to the list of drawn points. These points will then be drawn sequentially.

This time I would also compare the time against a brute force culling, where we check every single point to cull it instead of searching for nodes first. Another thing, to add more resolution to the timer, I've decided to run the culling 10 times with the same view frustum because sometimes it's so fast that the timer resulted in 0.

Table 3. Culling with vertex minimum 10
with brute force and octree, size is 100 x 80 x 40

Points	Octree	Brute
5.000	0,0004	0,0123
15.000	0,0005	0,0468
40.000	0,0005	0,1452
100.000	0,0013	0,4209

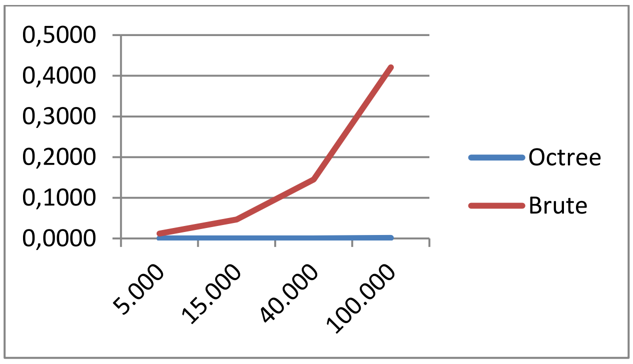
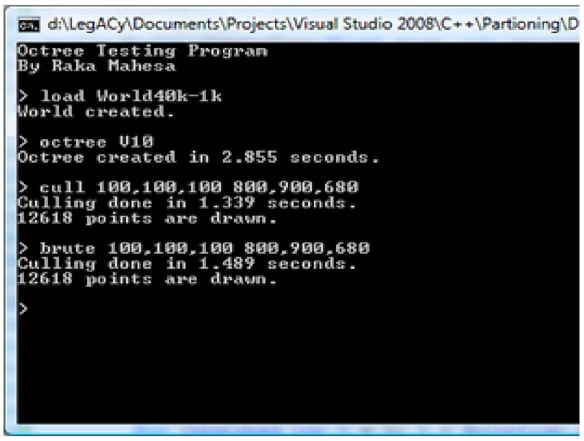


Fig 10. Table 3 chart.

As you can see, culling with octree wins by a landslide, it's not even fair. Even at the lowest number of points, culling with octree only takes 0.4 ms, while using brute force takes 12.3 ms, which means using octree is 30 times faster. Though it is interesting to see that between 15.000 and 40.000 number of points, brute force duration decrease a lot, while using octree the duration remains the same. Unfortunately, we cannot see how they would compete when the number of points is in the area of hundreds, since the duration would be measured in nano seconds. However, we could take a different approach, see my next test, it's pretty interesting.



```
d:\LegACy\Documents\Projects\Visual Studio 2008\C++\Partioning\D
Octree Testing Program
By Raka Mahesa

> load World40k-1k
World created.

> octree V10
Octree created in 2.855 seconds.

> cull 100,100,100 800,900,680
Culling done in 1.339 seconds.
12618 points are drawn.

> brute 100,100,100 800,900,680
Culling done in 1.489 seconds.
12618 points are drawn.

>
```

Fig 11. Screenshot during view frustum culling.

Table 4. Culling with vertex minimum 10
with brute force and octree, size is 700 x 800 x 580

Points	Octree	Brute
5.000	0,0177	0,0161
15.000	0,0492	0,0497
40.000	0,1339	0,1489
100.000	0,2936	0,3372

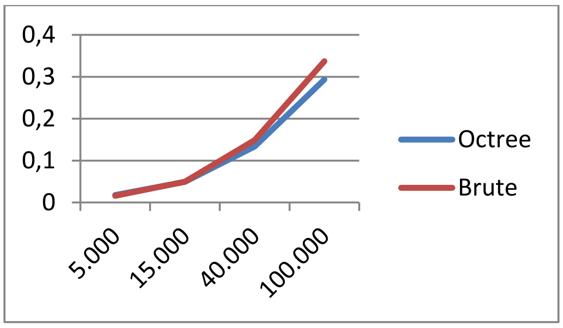


Fig 12. Table 4 chart.

Lo and behold, who would have thought that brute force algorithm could almost fight off a divide and conquer one like octree. As we can see in our fourth test, there isn't much difference between the duration of octree culling and brute force culling. The key here is the size of the frustum; the region used for culling was big, 700 x 800 x 580, which would make it around 32.5% of the whole world. For a big frustum (or a small world), using octree doesn't give much advantage over using brute force since they would need to manually check a lot of points anyway.

Table 5. Culling with vertex minimum 10 and
With vertex minimum 10 + size minimum 10.000.000

Points	**Min points**	**Min size**
5.000	0,0003	0,0008
15.000	0,0005	0,0022
40.000	0,0006	0,0061
100.000	0,0014	0,015

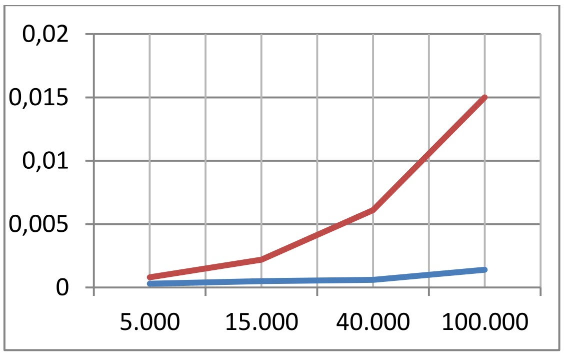


Fig 13. Table 5 chart.

The last test on this paper, fortunately, this proves to be an interesting one. Earlier, I've been wondering if it's good to limit octree to a certain minimum size, and this test has proven that it's not such a good idea, as the culling works a lot faster on the octree limited only by point number per node. Though, I have a doubt that it's probably because of the frustum, as limiting the octree would make each cell bigger, thus making it more ideal for handling bigger frustum.

Well, that should conclude my papers on view frustum culling with octree.

IV. CONCLUSION

There are several conclusions we can get from a number of tests I conducted:

1. The time it takes to construct an octree grows exponentially as the number of objects in the 3D space grows linearly.
2. Having more parameters to limit the creation of children node helps makes the creation of octree faster.
3. The creation of octree takes some time, making it unsuitable to be created every frame.
4. When the frustum is small, culling with octree is more than 30 times faster than using other.
5. When the frustum is big, there is no significant difference when culling with octree and without.

REFERENCES

[1] http://en.wikipedia.org/wiki/Octree accessed on 8/12/2010.
[2] http://www.toymaker.info/Games/html/collisions.html accessed on 8/12/2010
[3] http://www.jaapsuter.com/1999/04/13/octree-forest/ accessed on 8/12/2010

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2010

[illegible]

Raka Mahesa 13508074