Ray Tracing Complex Objects Using Octrees

Josh Wolper and Jacob Carstenson
Swarthmore College
{jwolper1, jcarste1}@cs.swarthmore.edu

ABSTRACT

In this paper we outline an implementation of an octree complete with a raytracer that parses complex objects and renders them using octree traversal. We wrote a complete octree class from scratch with the help of some online resources and additionally wrote an object parser that incorporates the parser from our ray tracer project. Ultimately, this project stands as a large extension of the midterm raytracer in its goal to render more complex objects and speed up the rendering.

Unfortunately our octree implementation, though seemingly thorough and complete, is buggy. It works and gives significant speedups but is finicky and sometimes renders incomplete images. Thus only our object parser remains as a fully functioning element, it runs extremely slowly (highlighting the need for our octree implementation). We believe our octree class itself to be almost completely correct and useable, however our traversal algorithms/methods are likely incorrect somewhere as this seems to be where shapes are being dropped and not rendered. In the end it came down to time but we could not figure out how to fix the octree completely despite endless debugging. Despite this, we will go into extreme detail outlining the methods and ideas behind our traversal algorithms as we believe they only falter in our implementation.

1. INTRODUCTION

Ray tracing has been around since its proposal by Arthur Appel in 1968 [1] and since has become one of the most common ways to render images. In combination with the classic Blinn-Phong model of lighting we created a small CPU based ray tracer for our midterm project. As a final project for this course we decided to pursue a large extension of this tracer by adding more complex concepts and more intensive computations. This entails adding object parsing, complex objects, and data structures such as octrees to speed up hitTimes.

Octrees are data structures that divide 3D space into a tree- each node having exactly 8 children save for the leaf nodes. This way of organizing space will allow our ray tracer to run more quickly as we may divide each object in the scene into "nodes" and instead of linearly searching for hits on objects we may instead narrow searches down to a few "pierced" nodes. Ideally a ray will enter one octree root, determine which child it hit, then which child of that child it hit, and so on. The higher depth to which we define our tree, the more nodes there will be and thus the program will theoretically run faster and faster as depth increases.

The object parser incorporates the parser from our old ray tracer and adds some commands to allow object file uploads. These object files define vertices and triangle fragments that our ray tracer may then render as a complete, complex object.

The primary challenges we faced were designing the octree itself and implementing an efficient way to determine ray-box intersections for the tracer. Despite having all the code necessary, our octree does not work perfectly but does create significant speedups. [2]

2. RELATED WORK

Due to the abstract and complex nature of 3D data structures, we needed to do some research into implementations of both the structure itself and also algorithms for traversal using a ray tracer. A student at the University of Clemson, Brandon Pelfrey has a multitude of free graphics resources, including some related to octrees in particular. His in depth discussion of what an octree is and how it functions helped us to understand the basic ideas behind the implementation and allowed us to formulate our own solution [2]. Additionally, he implemented his own basic octree that we used as a model for our structure implementation [3]. Unfortunately, his structure is extremely basic when compared with what we wished to do with ours as his octree is coded to store "point" data types and thus the methods relating to node intersections and inserting data were far more simple than what we needed our octree to do as we added entire shapes to our octree and determining whether a bounding box for a shape is intersecting with a node is far more complex than simply determining whether a point is within a node. Despite this, the basic structure of his octree proved extremely useful in creating a skeleton for our implementation, especially concering what data members we needed to create and what methods we needed to implement.

While the structure of the tree itself was difficult to visualize, we were able to grasp it with relative ease. This was not the case with traversal algorithms. There are many

Course paper for Swarthmore College course CS 40: Computer Graphics

methods for traversing 3D space in an efficent way, but the main idea for connecting our octree and our ray tracer was to determine which leaf nodes a ray pierces as it goes through the octree. Though seemingly a simple task, upon research we discovered that the most efficient way was to use complex parametric algorithms to compute face intersections. This is all outlined in our most important resource, "An Efficient Parametric Algorithm for Octree Traversal", an article from the Journal of WSCG written by three Spanish researchers in 2000 [4]. This paper outlines numerous equations we implemented and connected to our ray tracer in order to calculate face intersections, ray exits, and second node intersections. The combination of all this math and pseudo code allowed us to ultimately create what we believe to be a nearly fully functioning octree, and it would have been impossible without these related works.

3. DESIGN

For this project we are using a previously written CPU based ray tracer and as such will not go into depth on its creation; instead, we will focus on our octree design and how it interacts with the tracer. For our design we decided to skip creating an octree node class, but instead to just create an octree class. This is due to the nature of octrees, each octree is in itself a "node" as it will point to its children and any data it may store. Thus, instead of having an octree of nodes we will design our tree as an octree of octrees. Of course, the leaves will be octrees that are simply root nodes.

Our private data members include a list of child pointers, a shape pointer, a member for the origin of the sector in space, and a member to store half the dimension of the sector (half the width or height since they will be cubes). We need some efficient methods to make this data structure effective- the first being an insert method which will allow us to insert data into the tree.

There are three potential cases that may occur when calling insert: the first is that we hit an interior node. We are designing our octree to have no "data" associated with interior nodes, but instead only with our leaf nodes. Thus if we hit an interior node we must follow its child pointers until we find the node we want to insert into based on its origin and whatever data we have (in our case a triangle, so we will look at triangle coordinates and insert triangles into nodes based on comparisons with the node bound box compared with the triangle bound box). The second case is the node is a leaf and has not filled up its data, and thus we add the triangle to the list of triangle pointers. Third case however, is that we hit a leaf node and it already has its max amount of data. This would entail adding a level of depth to the tree (inserting a new tree to each of the this node's children) and taking all of its data and distributing it within its childrens' octrees. This is done using a function that takes a bounding box (taken from each shape) and tells which children are contained within this box. We then add the shape to all of these children. Bound boxes are defined by lower left corner and upper right corner and are found in various ways depending on shape- spheres are the simplest shape to find a bound box for as the min is simply the center point - a vector of the radius, and the max is the same but with the radius vector added. Every shape based on a plane is more complex, for these we took the minimum x value, minimum y value, minimum z value and put these in as our min point. And similarly we used the max x, max y, and max z for our max point. These bounding boxes allow us to complete our getChildrenInsideBox function which is needed two times in our insert function.

Next we added our method to determine which child a given point is in (getOctant). We did this by assigning the children in a systematic way suggested by our resources. Having the children ordered by a pattern allows us to know where/what to index when getting to later methods we implement. We used the following pattern where a "-" means less than the origin in that dimension and vice versa.

Index	0	1	2	3	4	5	6	7
x	-	-	-	-	+	+	+	+
y	-	-	+	+	-	-	+	+
z	-	+	-	+	-	+	-	+

Thus our function becomes quite simple as it can be achieved through nested if statements with 8 potential cases- one for each child. Furthermore, each child can be indexed by a 3-bit binary number, with each bit signifying x, y, and z relative positions (+ or - the origin of the octant). This allows us to use bitwise operations to check each axis independently. These bitwise operations are used in the input function when creating the 8 new child octrees. We use them to determine which child's number (0-7) gets which octanct, following the table above.

Our final and perhaps most important method of the octree we need to implement is one to determine which child of an octree was hit by the ray. We will do this by comparing where our ray is passing through with the origins of the top level of nodes, then with the next level down until we reach a leaf. Then we know that we have our deepest node once we've followed this path through the tree. It is also possible that multiple nodes will be struck by a ray. Testing for hits is extremely complicated and uses some complex parametric algorithms [4].

The final aspect of the Octree implementation is the ray-octree intersection that needs to be calculated in order to know which leaf nodes are pierced by a ray. This is done by taking the ray parametric function:

$$r_x(t) = p_x + td_x$$

$$r_y(t) = p_y + td_y$$

$$r_z(t) = p_z + td_z$$

We know that an intersection has occured if there exists one t that satisfies:

$$x_0(o) \leq r_x(t) < x_1(o)$$

$$y_0(o) \leq r_y(t) < y_1(o)$$

$$z_0(o) \leq r_z(t) < z_1(o)$$

where (x_0, y_0, z_0) and (x_1, y_1, z_1) are the bounding box for any given octree node o. Knowing this, we can calculate the times for when the specific ray cross each the min and max values for each axis in the octree using the inverse of the first equation above:

$$t_{xi}(o, r) = \frac{x_i(o) - p_x}{d_x}$$

$$t_{yi}(o, r) = \frac{y_i(o) - p_y}{d_y}$$

$$t_{zi}(o,r) = \frac{z_i(o) - p_z}{d_z}$$

Where p is the origin of ray r, and d is the direction vector. This is done for i values 0 and 1 to give six independent times used in the ray-octree intersection algorithm. Now that we have all of the t_0 and t_1 values. To find the first time the ray hits the octree and the time when it exits, you take the max and min of respectively of each component time value:

$$t_{min}(o,r) = max(t_{x0}(o,r), t_{y0}(o,r), t_{z0}(o,r))$$

$$t_{max}(o,r) = min(t_{x1}(o,r), t_{y1}(o,r), t_{z1}(o,r))$$

Now we have sufficient information to know weather or not a ray interesects a given octree by checking the condition:

$$t_{min}(o,r) < t_{max}(o,r)$$

If this is true, interesection occurs. Another useful piece of information to remove redundant calculations are the time values of when the ray crosses the median of each axis called t_m. We find this using the following equation:

$$t_{xm}(o,r) = \frac{t_{x0}(o,r) + t_{x1}(o,r)}{2}$$

$$t_{ym}(o,r) = \frac{t_{y0}(o,r) + t_{y1}(o,r)}{2}$$

$$t_{zm}(o,r) = \frac{t_{z0}(o,r) + t_{z1}(o,r)}{2}$$

Now we have all the information we need to find the pierced children of an octree. The first thing to do is find the first node intersected by a ray. The first thing to do is take t_{min} and figure out which compenent of t_0 it equals. Then use the following table to determine the entry plane:

Maximum	Entry Plane
t_{x0}	YZ
t_{y0}	XZ
t_{z0}	XY

Once you have determined the entry plane, there are two conditional statments that determine the state of two bits in a 3 bit number that decides which child octant is the first pierced node. Use the following table:

Entry Plane	Conditions	Bit
XY	$t_{xm} < t_{z0}$	0
	$t_{ym} < t_{z0}$	1
XZ	$t_{ym} < t_{y0}$	0
	$t_{zm} < t_{y0}$	2
YZ	$t_{ym} < t_{x0}$	1
	$t_{zm} < t_{x0}$	2

The next part of the algorithm is finding the next pierced nodes which can be found by finding the exit node of the current pierced node. This is done by finding t_{max} of the current child node. Similar to finding the entry plane above, the exit plane is determined by which component came out as the minimum value, with x maping to the YZ plane, y mapping to the XZ plane, and z mapping to the XY plane. Now, use the following table to find which is the next node pierced by the ray by finding which node you are currently on and what the exit plane is:

Current	YZ	XZ	XY
0	4	2	1
1	5	3	END
2	6	END	3
3	7	END	END
4	END	6	5
5	END	7	END
6	END	END	7
7	END	END	END

This is done for all of the visited nodes until the end is reached which signifies the ray exiting the octree. Now all of these pierced nodes will be checked for their hit times to find which ones were hit, and should be drawn.

This implementation works under the assuption that every component of the direction vector is positive. If a negative component is encountered the ray needs to be redifined to r' with the following equations:

$$d'_e = -d_e$$

$$p'_e = p_e - 2(p_e - o_e)$$

with o being the origin of the octree. This is performed for all components $e \in x, y, z$ where

$$d_e < 0$$

This effectively reflects the ray across the tree on the given axis, requiring a reordering of the child node numbers. This function is used for a given child node i to convert it to the actual child node referenced:

$$f(i) = i \oplus a$$

where a can be found using:

$$a = 4s_x + 2s_y + s_z$$

where s_e is 1 if $d_e < 0$ and 0 otherwise. This information on the ray-octree traversal was adapted from the Revelles et. al. paper[4]

Finally, we also decided to add an implementation of a parser for .obj files to make testing of the octree easier. We did not implement all aspects and functionality of .obj files, just vertex and triangle definitions, marked by the v and f commands respectively. A dictionary is kept of all the vertices with indexes starting at 1 used as the keys. The triangles are defined using these indexes as references to each point of the triangle. Since parsing any file format is similar in that it is a line by line process, many snippets of code are borrowed from Andrew Danner's input parser from the raytraycing lab, especially parseFile and parseLine, which have the anlogs of parseObj and parseObjLine. Furthermore, parser.h is included in the parser's implementation and parseFloat and parseInt are used in the .obj parser extensively.

4. IMPLEMENTATION

We used Qt and the Qtcreator IDE to implement all of our code. Nearly all of our code is either from our midterm project or originally written by us for this project. However, we did use pseudo code for a travesal algorithm from a paper [4] and all object files used are courtesy of public domain resources on the internet [5] [6].

For the object parser we implemented it in much the same way that the parser itself worked from our raytracer. We

designed it as a data structure so that we could simply add a new "cmd" to the parser in our raytracer that would be triggered by the command "obj" and creates an object parser "object" and then creates a vector of triangle objects while reading through the object file. Then that list of triangles is entered into the object list in our ray tracer.

Our project runs by simply running ./makescene <input .txt file>. The txt file is in the style of the txt file from the midterm raytacer but includes an extra command, "obj" to add a .obj file for our parser. This command can be entered in the file as obj bunny.obj for example. We also added a command, "oct" that should be included in the input file if the user wants to use octrees to render.

5. EVALUATION

The project was both a success and a failure: we managed to implement a fully functioning object parser that allowed us to render complex objects in our raytracer through the use of .obj files. And while our octree implementation did not work perfectly in the end, we feel it is extraordinarily close to being a complete implementation and we learned a lot from designing it and working through the equations and algorithms necessary to make it efficient and effective. Normally we would have gauged the success by operational speed ups and how much faster the octree could make our ray tracer. Unfortunately, due to the unwieldy and intricate nature of the structure, we found it nearly impossible to debug and discover the issues buried deep within it. Fortunately we at least had the parser as a shiny new feature of our ray tracer and the octree does speed it up considerably, but it does not always render complete images unfortunately.

Our octree does appear to work but it has some issues with using small values for maxData and dropping shapes in general. It definitely speeds up rendering by a huge degree, but it commonly drops shapes from the scene which is a pretty glaring issue. If we had to determine the source of our errors in the octree, we believe we could narrow it down to either the traversal algorithm we used or/and the insert function implementation. While debugging we discovered that for some reason we were getting negative t values where we shouldn't have been getting negatives due to the assumptions of our parametric algorithms, but we could not find the source of this as it seemed to be nearly random. As for the insert function we tested it thoroughly and believe it to be working, yet strange problems begin to occur if we set the max number of shapes in a node to lower numbers and we believe this could have caused problems when there were shapes very close together and thus going deeper and deeper down into further octree root nodes. However, after countless hours debugging we decided to leave it alone. So in its current state it renders faster with octrees but they are usually incomplete.

The parser is harder to determine the success of as it was very difficult to test since most objects took hours to render so we could only test so many times. We spent a lot of time messing with coordinates trying to make a test scene look good, but this was again difficult as it took hours at a time to render between tries. It would have been great to get the octree working as this would allow us to get much better images from our parser.

6. CONCLUSION

Overall the project was not a complete failure. We created what we believe to be an almost fully functional octree which in itself is an extraordinarily complicated structure to both create and to traverse. In addition we did end with one visible productâĂŞ our object parser.

We learned a lot from this trial and error, both about navigating 3D space and about designing a new data structure basically from scratch. A tremendous amount of thinking goes into simply thinking about organizing 3D space, let alone traversing it efficiently and this could easily be considered one of the biggest gains of the project, especially considering we were ultimately unable to get it to work.

This project should stand testament to the unfortunate reality that creating new things is super hard; not only is it hard starting a data structure from scratch, but even harder to do is start a relatively undocumented data structure from scratch is far worse. Finding documentation on octrees and how to traverse them effectively was pretty difficult, which was shocking as we thought that it would likely be a commonly used structure. As it turns out it appears that octrees are a somewhat outdated structure that has been replaced both by other similar structures like KD-Trees as well as parallel programming artifacts like CUDA.

Ultimately, if we were to continue working on our project in its current state we'd love to just continue debugging until we discovered what the issues are. Perhaps a misuse of a vital equation, or perhaps a simple missing line of code. We have no idea, but we wanted to see our baby work and that would easily be the goal for future work. As an even further extension we wished to test our octree on reflections and shadows as well, as both of those features required a search of the object list adding the octree would make it much faster (especially reflections as they take forever). Perhaps we could even try adding the Stanford bunny to a mirror room and see it in all its infinitely reflected glory.

7. REFERENCES

[1] A. Appel. Some techniques for shading machine renderings of solids. *Proceeding AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968.

[2] Brandon Pelfrey. Coding a Simple Octree. http://www.brandonpelfrey.com/blog/coding-a-simple-octree/.

[3] Brandon Pelfrey. SimpleOctree. https://github.com/brandonpelfrey/SimpleOctree/blob/master/Octre

[4] J. Revelles and C. Urena and M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. In *Journal of WSCG*, pages 212–219, 2000.

[5] Morgan McGuire. Unit-Volume Cube. http://graphics.cs.williams.edu/data.

[6] Unknown. Stanford Bunny OBJ. http://graphics.stanford.edu/ md-fisher/Data/Meshes/bunny.obj.