Semestral work assignment number 1
An Efficient Parametric Algorithm for Octree Traversal

Ondrej Perny[1]
Faculty of Electrical Engineering, CTU Prague

Abstract

Assignment of this work is to implement a ray tracer using octrees described in the article: J. Revelles, C. Urena, M. Lastra: "An Efficient Parametric Algorithm for Octree Traversal". Then test the implementation on scenes (consisting of triangles) with different number of objects and report the performance for at least primary and shadow rays (point light sources).

Keywords: ray tracing, octrees, parametric algorithm

1. Introduction

In this assignment I will introduce the algorithm described in "An Efficient Parametric Algorithm for Octree Traversal". It is a fast algorithm for traversing octrees using the parameters rather then specific position of octree nodes. Compared to kD-tree might traversing be slightly slower, but may have different advantages. Following section describe the implementation of the octree and the traversal algorithm and issues that I have encountered. My goal is to make working and time efficient implementation.

2. Algorithm Description

An efficient parametric algorithm for octree traversal is top-down algorithm for traversing hierarchical spatial structure octree. Every node in octree is rectangle in space with all edges of same length a. Every node can be divided to 8 sub-nodes where each sub-node has edge length $a/2$. While octree is structure using spatial division, for traversing with this parametric algorithm, only initial position of the root node is necessary. For further sub-nodes traversing are used parameters that decide which node is intersected next. For that reason is important to have sub-nodes labeled accordingly to the axis, as shown in figure 6.

The algorithm will be explained for 2D case. 3D works similarly just with addition of one axis. Ray is defined ad (p, d) where p is the origin point and d direction vector, both with one component for each axis. Point position can be calculated as $x_r(t) = p_x + td_x$, for each component. Then if for each component apply rule $x_0(o) < x_r(t) < x_1(o)$ then at least one real value t exists. The algorithm is called a parametric algorithm because all computations use t such that $(x_r(t), y_r(t))$ is a point on a node boundary [1]. Similarly when we have node o and ray r, then the parameters which define where ray intersect with the boundary of the node are $t_{x0}(o, r), t_{y1}(o, r)$. Parameters are explicitly defined as:

[1] B4M39DPG – Ondrej Perny, winter semester 2020/21

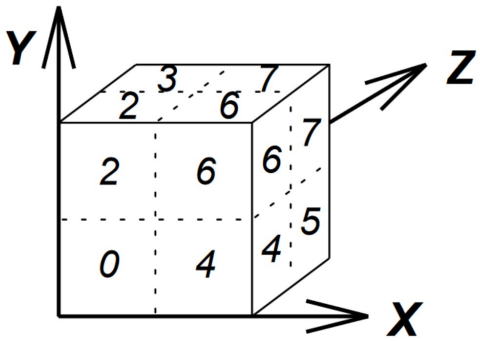

Figure 1: Labeled octree.[1]

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

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

where i is 0 and 1. There parameters are calculated for each node. They are computed for the root node initially and then incrementally for sub-nodes. For the selection of the intersected node we need to calculate $t_{min}, t_{max}, t_{xm}, t_{ym}$, where last two values parameters are for the horizontal and vertical line respectively, that divides node in the halves.

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

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

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

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

Then we can compare those calculated parameters and based on that decide which node is intersected next as shown in figure 2.

For the 3D version of the algorithm on the octree, parameters are calculated same, however parameters for the additional z axis is required. So same as shown for x, y, parameters t_{z0}, t_{z1}, t_{zm} are calculated. For the 3D implementation different approach is suggested using states for nodes and transitions between them.

To find the first state - first intersected node, the entry face is needed. For we compute $max(t_{x0}(o), t_{y0}(o), t_{z0}(o))$. If $t_{x0}(o)$ is max then corresponding plane YZ is entry plane, alike for other two planes. Then based on the specific entry place we compare the parameters according to the table in figure 3. We have 3 bit value where we switch (XOR) the first,

if $t_{min}(q,r) < t_{max}(q,r)$ **then**
  if $t_{ym}(q) < t_{xm}(q)$ the ray crosses q_2, and
    if $t_{x0}(q) < t_{ym}(q)$ the ray crosses q_0.
    if $t_{xm}(q) < t_{y1}(q)$ the ray crosses q_3.
  else if $t_{xm}(q) < t_{ym}(q)$ the ray crosses q_1, and
    if $t_{y0}(q) < t_{xm}(q)$ the ray crosses q_0.
    if $t_{ym}(q) < t_{x1}(q)$ the ray crosses q_3.

Figure 2: Parameter comparisons.[1]

Entry Plane	Conditions to examine	Bit affected
XY	$t_{xm}(o) < t_{z0}(o)$	0
	$t_{ym}(o) < t_{z0}(o)$	1
XZ	$t_{xm}(o) < t_{y0}(o)$	0
	$t_{zm}(o) < t_{y0}(o)$	2
YZ	$t_{ym}(o) < t_{x0}(o)$	1
	$t_{zm}(o) < t_{x0}(o)$	2

Figure 3: Parameter conditions.[1]

second or third based on the conditions. The numeric value, after all conditions are checked, correspond to the current sub-node state.

When we have first intersected sub-node, following intersected node can be find by deciding which face it will use to leave current sub-node. To find that we need $min(t_{x1}(o), t_{y1}(o), t_{z1}(o))$ of the currently intersected sub-node, where if $t_{x1}(o)$ is min then exit plane is YZ, respectively for other planes. Respective transitions are shown in figure 4.

Current sub-node (state)	Exit plane YZ	Exit plane XZ	Exit plane 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

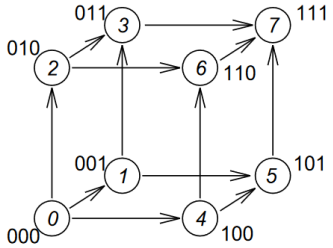

Figure 4: State transitions (END means leaving parent node) and sub-nodes graph.[1]

There are two special cases where this algorithm as is, would fail, that is if one main axis is parallel with ray, or when ray has negative directions. Those cases can be solved in

implementation.

3. Implementation details

Base data structure used for this assignment is the octree with edges of same length, build recursively with stop condition tree depth.

Ironically biggest issues (and most time spent) were during programming naive solution to check if my octree is build correctly. I spend a lot of time fixing more smaller issues some in the traversal and some in the tree itself. When I look back at it now, there was not any specific complex problem, but rather my inexperience with programming non-trivial 3D data structures made it hard for me.

Big help was GLV visualization program, thanks to which debugging was a lot easier since I could see the problems visually.

When I was sure my octree is build correctly I started implementing parametric algorithm based on the given paper. Due to my previous experience of debugging the naive algorithm and octree, implementing this was relatively seamless, until the last part, when I needed to alter ray direction to be positive. When I rendered scene with ray having positive direction, everything worked correctly, but when even one axis needed direction swap, rendered scene was broken. Due to changing orientation of the rays, and therefore whole octree, visualization was not very useful this time. So I spend time browsing the paper and realized that my octree was not aligned to any special position, and therefore rotating octree around axis that is not in the middle of the octree shifted its position and rays were missing the octree. I corrected this by shifting the center of the octree to the position $x = 0, y = 0, z = 0$ and everything worked properly.

Then I spend a lot of time trying to optimize memory and speed, reducing octree node size to minimum, removing unnecessary and pre-compute values to avoid making redundant calculations.

The algorithm is described as recursive in the paper. I implemented both variants recursive and non-recursive. While non-recursive version is probably more robust and save in terms of possible preventing possible stack size issues that recursive version could have, it is slightly slower than recursive version, due to the necessary overhead in terms of copying parameters to the queue. For results recursive version is used.

I assume that the time I have spent working on this assignment is approximately 100 hours.

4. Results

Measured times were on the machine with specification:

- **OS** Windows 10, 64bit version
- **Compiler** ISO C++17 Standard (std:c++17), from the toolset version: Visual Studio 2019 (v142), compiled in release mode with the O2 speed optimization
- **CPU** Ryzen 5 3600, hexa-core, 3.6 GHz, L1 384KB, L2 3MB, L3 32MB
- **RAM** 16Gb dual-channel DDR4-3600

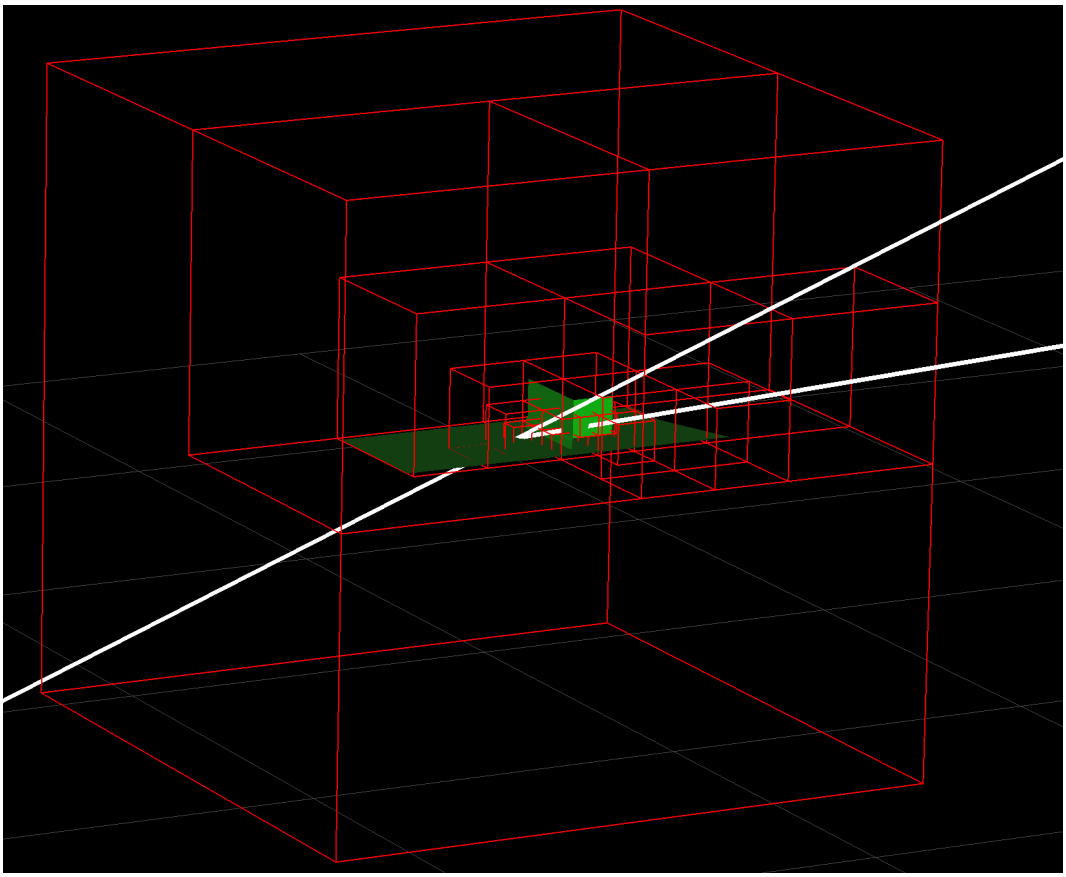

Figure 5: Visualised traversing for primary ray and shadow ray.

Comparison with original paper is not possible as only given information is cpu time per octree depth level, however due to hardware differences, speed of my solution is orders different. Plus paper does not mention resolution rendered images.

All my renders are in 800x800 resolution.

5. Conclusion

My final implementation is octree with efficient parametric algorithm (both recursive and non-recursive version) in the Nanogolem framework.

Measured results seems satisfying.

Programming 3D data structure and corresponding algorithm was new for me, therefore probably harder than it should be. I feel I have gained sufficient experience to be more comfortable working with similar data structures in future.

As for recommendation for repeating the implementation, I believe only think that could help me is better theoretical preparation/learning, both general data structures and this specific paper I have implemented.

Looking back, I believe that programming similar assignment could take me far less time than I programming this work. Due to the time spend, I have finished it very late, but I am satisfied with the current solution, hoping it meets all requirements of the assignment.

model	Size	T_B [ms]	T_R [ms]	N_IT	N_TR	N_Q
A10	29174	660	0.00046964	9.7573	11.889	717578
Armadillo	345944	964	0.0010601	25.682	18.328	789572
asianDragon	7219045	8912	0.0041251	192.04	15.025	785431
city	68489	564	0.00069	13	23	101801
city2	75420	84	0.0021563	162.54	20.429	914973
park	29174	314	0.0010596	37.605	42.672	1055079

Table 1: Table with measured results for octree depth 8.

model	Size	T_B [ms]	T_R [ms]	N_IT	N_TR	N_Q
A10	29174	2048	0.00045291	5.8894	12.964	717578
Armadillo	345944	2047	0.0010461	14.379	20.978	789572
asianDragon	7219045	11650	0.0021555	57.214	16.668	785431
city	68489	2087	0.00077896	11.242	25.858	1018019
city2	75420	114	0.0013028	72.746	22.268	914973
park	29174	1651	0.001125	31.342	52.281	1055079

Table 2: Table with measured results for octree depth 9.

Acknowledgment

I would like to thank prof. Ing. Vlastimil Havran, Ph.D. for his advices about debugging this data structure, especially suggestion of the GLV visualization tool.

References

[1] J. Revelles, C. Urena, M. Lastra. An Efficient Parametric Algorithm for Octree Traversal (2000).

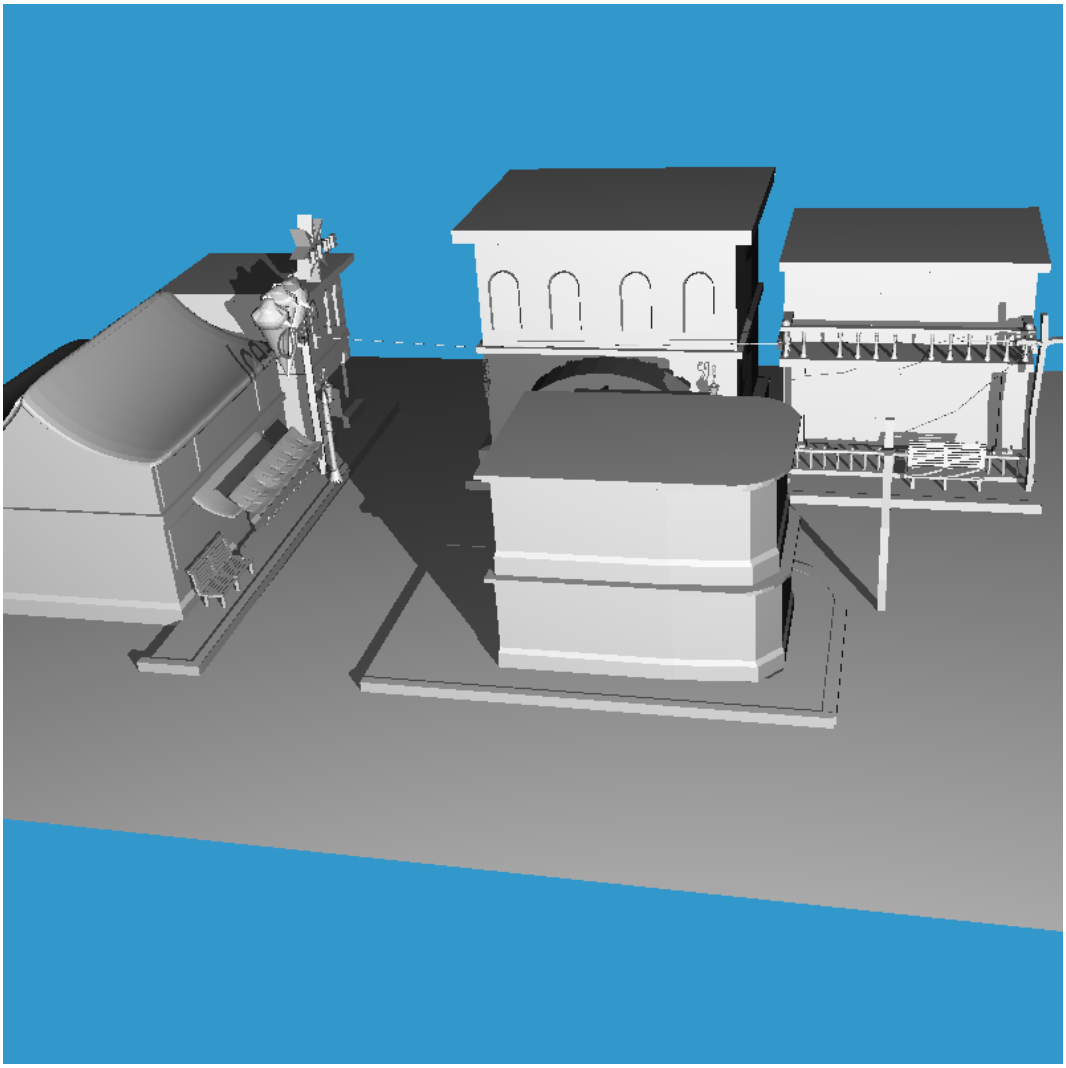

Figure 6: Example of rendered model City.