ISBN 978-93-84468-20-0
Proceedings of 2015 International Conference on Future Computational Technologies (ICFCT'2015)
Singapore, March 29-30, 2015, pp. 123-129

Volume Rendering Using Hierarchical Spatial Correlation and Occlusion Sorting

Arnab Mitra, and Som Bose

Department of Electronics and Electrical Engineering, IIT Kharagpur

***Abstract*:** Use of voxels for rendering objects in computer graphics is useful for complex geometries. However the hardware support for such rendering algorithms doesn't exist. In this paper we present an algorithm, which is bound by the order of image and can support dense processing units as the hardware required is for simple arithmetic and logical operations. Moreover, the algorithm discussed in the paper describes an easy way using logical operations to perform occlusion sorting.

Keywords: *Occlusion sorting, Sparse Voxel Octree Rendering, Spatial Correlation.*

1. Introduction

Object rendering traditionally [1] is done by converting the objects into triangles for rendering. Triangles have been used for rendering because of their compactness in representation of surfaces which works well with simple geometry, but not so when the number of vertices in an object increase. This advantage is becomes less significant when the geometry becomes more complex [2] and has many triangles, hence vertices. The orders of such algorithms, in such a case depend heavily on the geometry.

Voxels provide an alternative way of storing object [3]. Traditionally, voxels were used to represent volumetric data especially in medical imaging etc. Recent trends in computer graphics has been towards using object represented as a voxel for volume rendering. Several algorithms already have made their strides in this direction.

Since we aim to have spatial correlation some methods that use ray casting techniques with objects represented as Sparse Voxel Octrees [4] can be ignored for implementation of an algorithm, as ray casting is an independent pixel by pixel rendering, which works well when run in parallel. There also has been methods [5] which had a hierarchical approach toward rendering, which inherently takes into account the spatially correlated information, but doesn't take into account the occlusion test that can be performed to save computations. There have been methods sort first and sort last rendering [6], which demands identification of primitives which makes such an option for occlusion sorting less effective as again occlusion sorting depends on the geometry.

In this paper we present an alternative approach towards volume rendering using a hierarchical algorithm to tap spatial correlation and also a new method for occlusion sorting, which makes the algorithm efficient. The algorithm presented in the paper follows image order and does not depend on geometry. Occlusion sorting is implemented by use of simple logical operations and independent of the geometric complexities of the object. Moreover, the entire algorithm needs just basic arithmetic and logical operations, which shows that such a method if supported by hardware can yield very good performance results.

2. Definitions

The algorithm presented in the paper assumes certain conventions and definitions which is explained in this section

2.1 Voxel

Our approach uses voxels as a rendering primitive. Voxels represent a cubical subspace in the object space and are only characterized by colour and does not have position and size information.

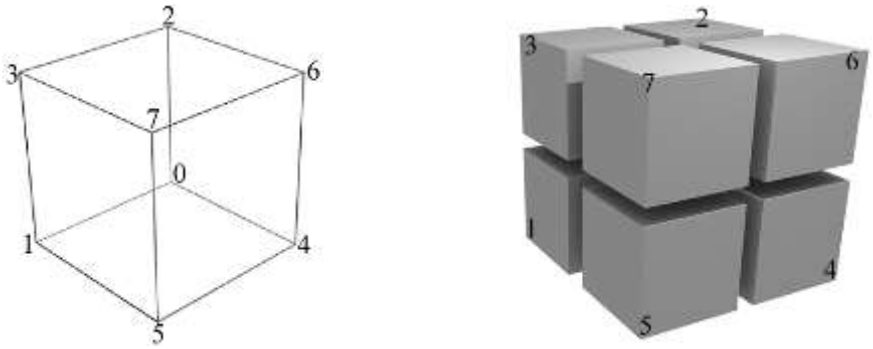


Fig. 1: Voxel (left) and children (right). Children are numbered according to vertices they are connected to.

2.2 Voxel Graph

A voxel graph consists of voxels at each node. Every node has 8 children which splits the voxel's subspace into eight equal and symmetric subspaces each represented by a child voxel. Child order is shown in Fig 1. Child 0 is hidden in the figure. For our purposes, this graph is simply an octree [3, 4], but leaf nodes point to themselves. It is also to be noted that our approach holds true for any general voxel graph where a node may point to any other node which may even be its ancestor. Such graphs may be used to compress the object octrees with repeated structures.

2.3 Object

Objects are characterized by a cubical axis aligned bounding box (AABB) and a node of a voxel graph. This node is aligned to the AABB and the size and position of each voxel in the graph can be determined at runtime by traversing the voxel graph.

Voxels are mapped to a set of pixels by its axis aligned bounding box in image space. Voxel mapped to a single pixel are drawn.

2.4 Record

A record is an entry in a stack used in the algorithm and consists of

a) Object space minimum and maximum vertices.

b) Projected vertices stored in Morton order [9].

c) Quadtree Node index.

d) Octree Node index.

3. Algorithm

The input is an object described in Section 2.3, a model view matrix to transform the object to global coordinates and a camera matrix for perspective projection. From this, we find the projected vertices in the image space. The image space is stored as a quadtree, which is pre-computed before the actual algorithm. This pre-process requires multiplication but step needs to be done only once per object and can be done on CPU.

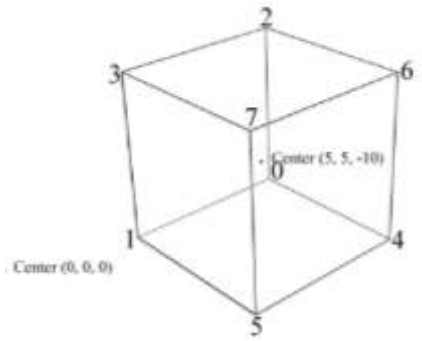
3.1 Overview

Once the input in the form of an object and the quadtree of the image space is constructed and the object's initial projected vertices are computed, a record with quadtree root, octree root and object minimum and maximum vertices is pushed into a stack and the following algorithm is executed:

```
while( stack is not empty)
{
    Top = stack.pop();
    if( top.quadtreeNode is drawn)
    {
            end iteration;
    }
    if( Top.voxel maps to one pixel)
    {
            draw the pixel with voxel color and normal
            update filled information
            end iteration
    }
    Center = ComputeObjectSpaceCenter (Top.MinMax)
    Top.children.OcclusionSort (Center, CameraPosition)
    VertexPool =  PrecomputeChildVertices (Top.Vertices )
    for ( i = 8 to 1 )
    {
            CurrChild = i^th ChildInOcclusionOrder (Top.children)
            CurrChild.MinMax = ComputeObjectSpaceMinMax (Top.MinMax, Center)
            CurrChild.Vertices = ComputeImageSpaceVertices(VertexPool)
            CurrChild.BB = ComputeImageSpaceBB(CurrChild.Verices)
            CurrChild.quadTreeNode = SmallestUnfilledFittingQuadtreeNode(CurrChild.BB)
            If(CurrChild.quadTreeNode != NULL)
                    stack.Push(CurrChild)
    }
}
```

3.2 Occlusion Sorting

Traditionally depth sorting has been performed for front to back ordering of primitives. [6][8] describes variations in depth sorting such as sort first or sort last approaches. Triangle rasterization resorts to sort last approach using a depth buffer. [7] approach this problem by remapping children indices and then sorting them in object space itself.



Octant = 110
Order = 110 XOR 111 = 001 = 1
Order = 110 XOR 110 = 000 = 0
Order = 110 XOR 101 = 011 = 3
Order = 110 XOR 011 = 101 = 5
Order = 110 XOR 100 = 010 = 2
Order = 110 XOR 010 = 100 = 4
Order = 110 XOR 001 = 111 = 7
Order = 110 XOR 000 = 110 = 6

Fig. 2: Occlusion ordering. Octant bits are determined from whether centre is left or right of camera in axial directions.

In our approach, we have avoided sorting altogether by relaxing the criteria for sorting from depth order to occlusion order. Objects sorted in occlusion order have the property that $voxel_i$ may be placed after $voxel_j$ as long as $voxel_i$ does not occlude $voxel_j$. For voxels which do not occlude each other, the order may be arbitrary.

We find the center of the current voxel in camera space. Accordingly we calculate a 3 bit vector Octant whose 2^{nd} bit, 1^{st} bit and 0^{th} bit is high if respectively x, y and z coordinates are positive. The i^{th} child in occlusion order is simply given by Octant XOR Order(i) where Order is the array: {7, 6, 5, 3, 4, 2, 1, 0}. An example is shown in Fig 2.

As a proof of this, let us consider 1 of the 3 orthogonal planes (in object space), which split the voxel into its children. 4 children lie on either of the half spaces of this plane. If the camera lies on a half space, say the negative half space as in the figure, the 4 children in the other half space cannot occlude the ones which lie on the camera's half space. The voxel centre position directly tells us which half space the camera lies in. XOR simply determines whether the child lies in the same half space as the camera or not. Generalizing this to all three planes, we will get a children which lies in front of all planes (111), 2 planes(110, 101, 011), 1 plane (100, 010, 001) or no planes (000). This order is stored in Order(i). Octant XOR ChildIndex = Order. This implies ChildIndex = Octant XOR Order. This ensures occlusion ordering in our algorithm. It is important to note that ordering occurs directly in object space.

3.3 Updating filled information

Each quadtree node has a 4 bit draw vector which determines which of its nodes are completely filled. When a pixel is drawn, all bits are set on the leaf quadtree node.

When all bits of a quadtree node's draw vector is set, it indicates the quadtree node is completely filled and the parent's corresponding bit in the draw vector is set. This is repeated recursively.

3.4 Computing child vertices

The child vertices are computed by a simple principle that co-linearity is invariant to projective transform. We start with the parent vertices in image space and compute 27 vertices (8 corners, 12 edge centres, 6 face centres and 1 cube centre) that will be used for the eight children. Each child's vertices are obtained from this set of vertices.

3.5 Finding smallest unfilled quadtree node

We compute the two lines, the horizontal and vertical lines bisecting the quadtree. We test the voxel with these two lines and use the unfilled information computed in section 3.2 to determine unfilled quadtree nodes which overlaps with the voxel. The quadtree node that is being processed is updated if the voxel lies in only one quadrant and the process is repeated again till we find no unfilled quadrant or we reach the leaf quadtree node or it overlaps with multiple quadtree nodes.

3.6 Implementation for a light hardware

Algorithm implementation in computer graphics is usually done by having a keen eye on the hardware requirements. Some works are also based on just developing hardware architectures for ray casting algorithm. [10][11]

The algorithm is implemented with a keen eye on demanding very light hardware support. Additions and shifting require lesser chip area and is cheaper than multipliers. Our implementation expresses the vertices in the projected space as fixed point numbers. By doing this, we have removed all the floating point operations. This also enables us to find mid points for computing vertices (section 3.4) by simple right shift operations rather than floating point divisions.

Sorting also needs a lot of branches which detriments performance. By relaxing the depth sort criteria to an occlusion sort we can achieve constant time sort for the children of a voxel by just knowing centre location with respect to the camera position and XOR operation. Another major part of the algorithm is rejection of image

space where a particular voxel will be drawn. This process needs comparison of bounding of the child of the voxel in consideration with the quadtree bisectors and simple bitwise AND operation with the 4 bit fill vector. Thus in the entire algorithm we don't need heavy hardware support.

4. Results and Analysis

In this section we will show that the complexity of rendering depends on the order of image and not the complexity of the object. Moreover we will show that the without applying occlusion sort the time required to draw is significantly larger. The voxel images were generated by a voxel editor, Voxel Shop and exported to a file which is then parsed to generate the octree that we need. Some sample images as generated by the voxel editor and by our code are shown below, which proves the correctness of our implementation.

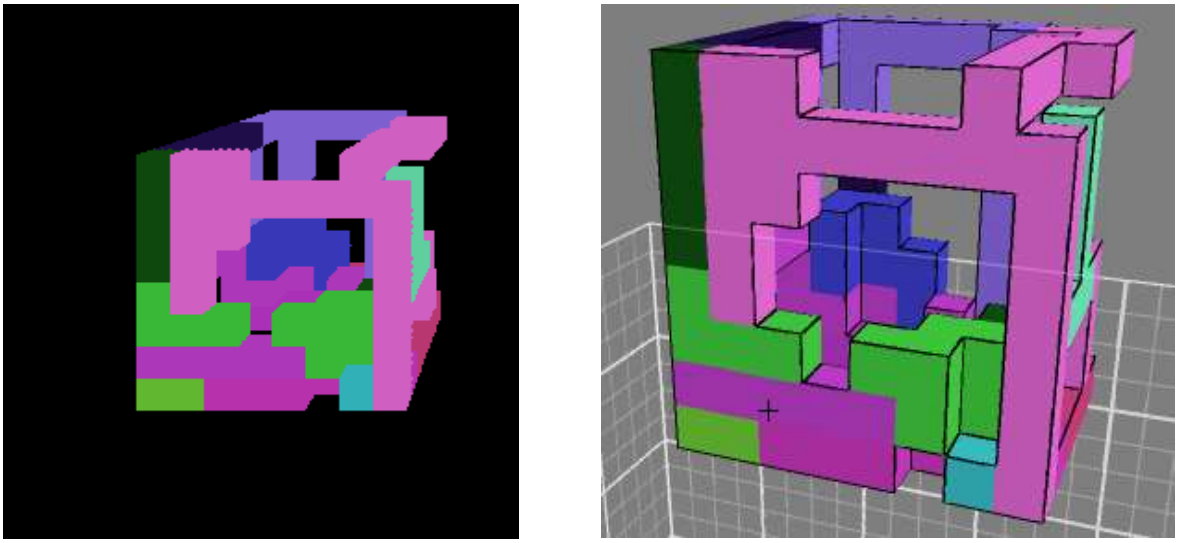


Fig. 3: Sample output of a complex object (left) and input geometry (right).

This image needed 119ms to render on an Intel Pentium dual core processor. Two simple images as shown below needed 110ms and 60ms to render on the same platform. This shows that the rendering algorithm doesn't depend on the complexity of image. Moreover, we can also see that the spatial information is also taken into account by the hierarchical algorithm.

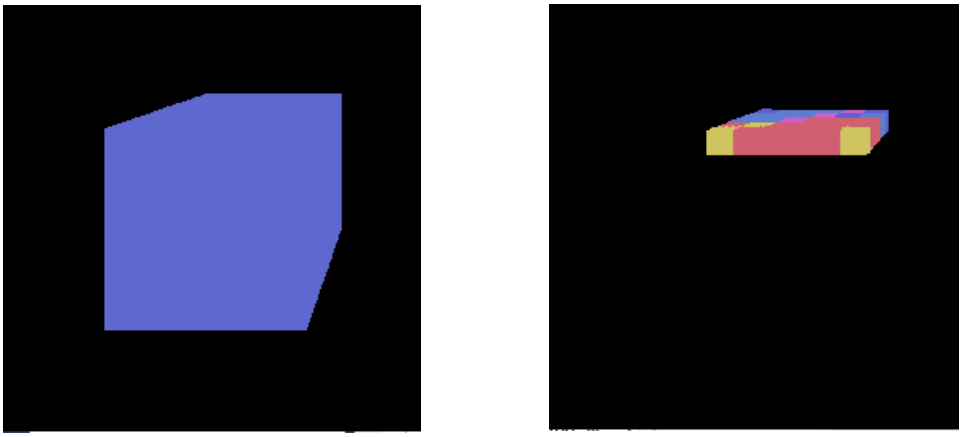


Fig. 4: Sample output of a simple inputs.

The order of the algorithm can be determined by changing the size of the image that is being rendered, as we can already see that the complexity of the octree doesn't matter. Thus the following table summarizes the results when the simple rendered figure xyz and changes the size of the image.

TABLE I: Render time

Resolution	Time (ms)
64 X 64	0 (<1)
128 X 128	20
256 X 256	110
512 X 512	462
1024 X 1024	1772

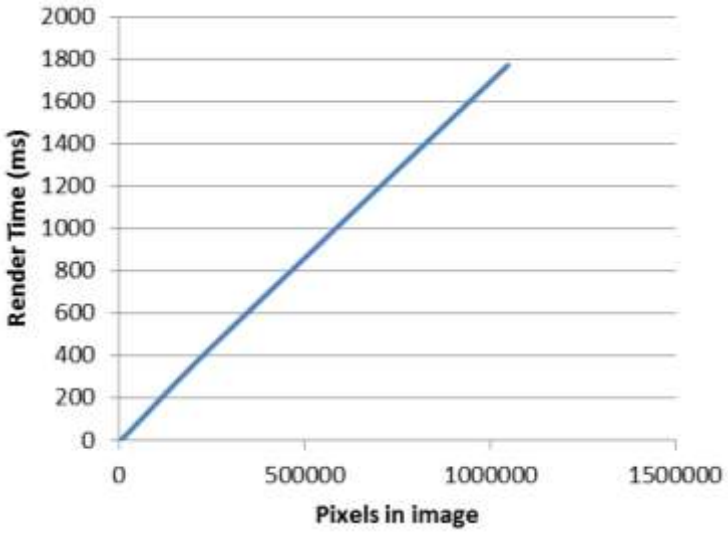


Fig. 5: Plot of render time against pixels in image.

5. Conclusion

The results show that the algorithm has achieved image order render time and is not dependant of object complexity. Also this algorithm requires only addition, bitwise, shift and relational operations which remove the need for complex multi cycle operation hardware such as MACs, divide or floating point operations.

From occlusion ordering we had seen that child voxels of order 100, 010, 001 or of orders 110, 101, 011 render completely independently of each other, thus allowing for them to be processed in parallel as well. Also when any of these voxels are further split they generate even more voxels that can be processed in parallel. Present GPUs however are not very suitable for handling such forking. Multi frame rendering with coherent camera movements can be sped up by utilizing time coherency, by directly rendering voxels which were rendered in the last frame similar to Hierarchical Z-Buffer algorithms.

6. References

[1] William J. Schroeder, Jonathan A. Zarge, William E. Lorensen, "Decimation of Triangle Meshes", SIGGRAPH '92 Proceedings of the 19th annual conference on Computer graphics and interactive techniques, pp. 65-70.

[2] Aaron Lee, Henry Moreton, Hugues Hoppe, "Displaced Subdivision Surfaces", SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 85-94

[3] Samuli Laine, Tero Karras, "Efficient Sparse Voxel Octrees", IEEE Trans. on Visualization and Computer Graphics, Vol.17 pp. 1048-1059, Nov 2010.

http://dx.doi.org/10.1109/TVCG.2010.240

[4] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees -- Analysis, Extensions, and Implementation. NVIDIA Technical Report. 2010.

[5] Szymon Rusinkiewicz, Marc Levoy, “QSplat: A Multiresolution Point Rendering System for Large Meshes” in SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 343-352.

http://dx.doi.org/10.1145/344779.344940

[6] Carl Mueller, “The Sort-First Rendering Architecture for High-Performance Graphics”, I3D '95 Proceedings of the 1995 symposium on Interactive 3D graphics, pp. 75-ff

[7] Sharat Chandran Ajay , Ajay K. Gupta , Ashwini Patgawkar, “A Fast Algorithm to Display Octrees”, . Indian Conference in Computer Vision, Graphics and Image Processing (ICVGIP) 2000, organized by CAIR, DRDO.

[8] Gideon Frieder, Dan Gordon, R. Anthony Reynolds, “Back to-Front Display of Voxel Based Objects”, IEEE Computer Graphics and Applications, Volume 5, pp. 52-60, January 1985

http://dx.doi.org/10.1109/MCG.1985.276273

[9] Jeroen Baert, Ares Lagae and Philip Dutre, “Out-of-Core Construction of Sparse Voxel Octrees”, HPG '13 Proceedings of the 5th High-Performance Graphics Conference, pp. 27-32

[10] Urs Kanus, Gregor Wetekam, Johannes Hirche, Michael Meißner, “VIZARD II: An FPGA-based Interactive Volume Rendering System”, in Field-Programmable Logic and Applications (Montpellier, France, Sept. 2002), Proc. of the 12th International Conference on Field Programmable Logic, pp. 1114–1117.

http://dx.doi.org/10.1007/3-540-46117-5_120

[11] Harvey Ray, Hanspeter Pfister, Deborah Silver, Todd A. Cook, “Ray Casting Architectures for Volume Visualization”, IEEE Transactions on Visualization and Computer Graphics Vol 5, pp. 210-223, July 1999

http://dx.doi.org/10.1109/2945.795213