Implementing Sparse Voxel Octree

Cheng-Tso Lin
University of Pennsylvania

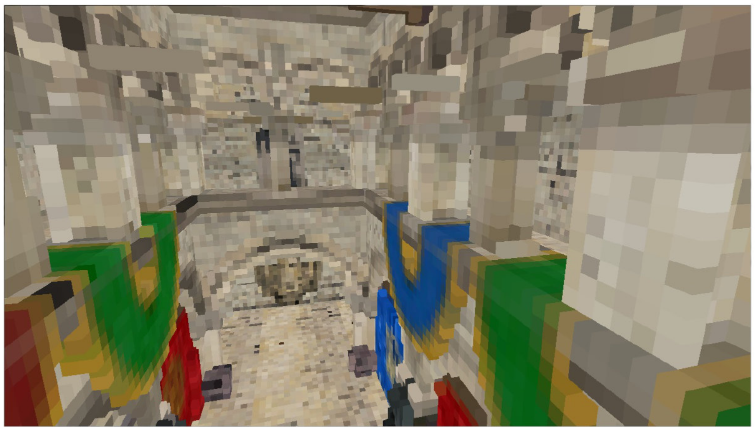


Figure 1. Voxelized Sponza model.

Abstract

This paper presents the work of our final project at CIS-565 *GPU programming and Architecture* course at University of Pennsylvania. In this final project, a method to construct a sparse voxel octree is implemented.

1. Introduction

A voxel represents an element on a regular grid in three dimensional space, and is commonly used to represent discretized 3D data. Memory consumption, however, is very high if voxels are stored in a regular grid structure. An octree is a popular way to hierarchically represent spatial data, and commonly used in computer graphics applications like collision detection and level-of-detail. But the memory consumption of an octree is even higher, since it needs to maintans data in multiple resolutions. Sparse Voxel Octree (SVO) exploits the fact that a polygonal object is usually hollow and the majority of octree nodes in an octree representing a polygonal scene are empty. With empty nodes eliminated, the memory consumption of a SVO is usually an order of magnitude smaller than that of a normal octree.

In our final project, we implement the SVO construction method described by Crassin et al. [Crassin et al. 2011]; we also utilize this SVO data structure to implement voxel cone tracing [Crassin and Green 2012], although it is still under development at the time of writing this paper.

2. Scene Voxelization

2.1. Hardware Rasterization

In the voxelization stage, the scene we want to render is voxelized by using hardware rasterization. First, the screen resolution is set as the dimension of our desired voxel grid, then a draw call is issued to render the whole scene, in which each triangle is orthographically projectd, in geometry shader, along the dominant axis of its plane normal vector. This way, a good deal of scene surfaces could be sampled in one single draw call. Figure 2 shows what the framebuffer would look like when using this proejction method.

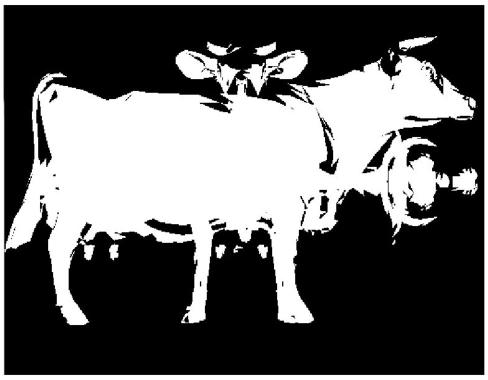


Figure 2. Framebuffer contents when each triangle is projected to maximize its projected area.

The voxels generated are called voxel fragments and stored in a voxel fragment list in GPU memory.

2.2. Conservative Rasterization

While hardware rasterization is fast, it produces holes or cracks on voxelized thin surfaces. The reason this problem occurs is that hardware rasterization will not generate fragments in pixel locations where pixel center is not covered by projected primitives. To mitigate this problem, a conservative rasterization method [Hertel et al. 2009] is employed to enlarge each triangles, making sure a pixel location have fragments generated if it is covered by triangles.

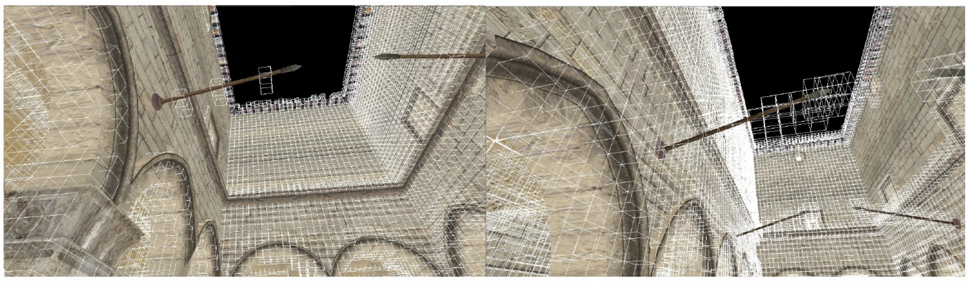


Figure 3. Left: Naive rasterization. Right: Conservative rasterization.

3. Spare Voxel Octree Construction

Several steps are needed for the SVO construction: First the scene is voxelized using the method described in Section 2, then the octree is constructed from top to bottom with the following three steps executed in each octree level. Figure 3 shows the difference between naive hardware rasterziation and our conservative rasterization implementation.

1) Node tagging: to find the octree node a voxel fragment node falls into, a compute shader is launched and each voxel fragment is handled by one thread. A tag bit of a octree node is set if that node is non-empty. There is no synchronization issue at this step.

2) Node allocation: we pre-allocate a linear buffer serving as a node pool, and an offset pointer is maintained to record the remaining free space. For each tagged node, an tile in the size of 8 nodes is allocated, and the start address of that tile is assigned to the child pointer of the tagged node. To allocate node space parallelly, a compute shader is launched and each node in non-empty tiles is handled by one thread. an atomic counter is used to atomically accumulate the offset pointer.

3) Node initialization: Again a compute shader is launched to initialize the values of the newly allocated nodes parallelly.

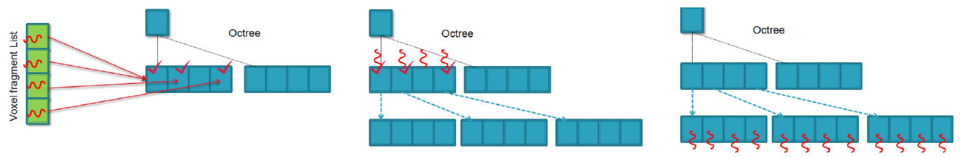
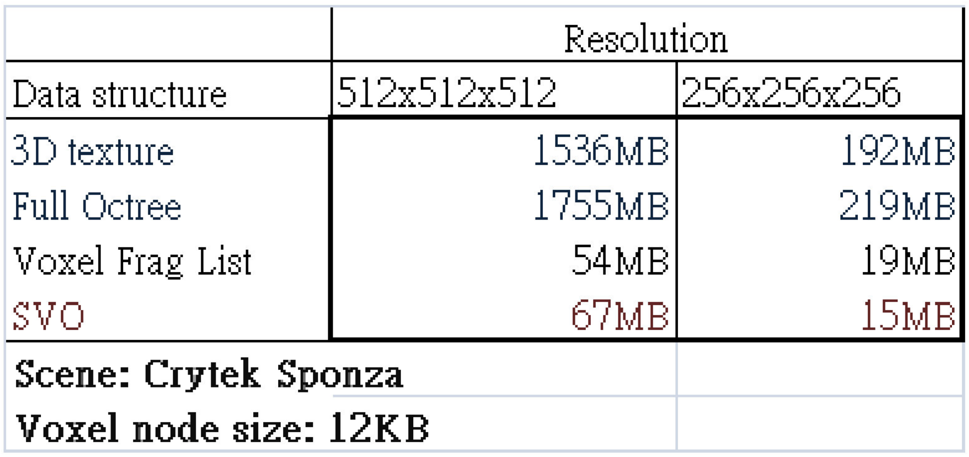


Figure 4. Left: Node tagging. Center: Node allocation. Right: Node initialization.

The value of voxel fragments are then injected into the leaf nodes and mipmapped into the interior nodes once the octree construction is completed. If multiple voxel fragments fall into the same octree node, their values have to be atomically added and averaged.

4. Performance Evaluation

Figure 5 shows the comparison of memory consumption bewteen 3D texture and SVO in storing voxelized Sponza model. SVO greatly reduces memory consumption when representing voxelized polygon scenes, since polygoned objects are hollow.



	Resolution	
Data structure	512x512x512	256x256x256
3D texture	1536MB	192MB
Full Octree	1755MB	219MB
Voxel Frag List	54MB	19MB
SVO	67MB	15MB
Scene: Crytek Sponza		
Voxel node size: 12KB		

Figure 5. Comparison of memory consumption between 3D texture and SVO

5. Future Work

The ultimate goal of this final project is to utilize SVO for real-time global illumination using voxel cone tracing. We expect to complete implementing the voxel cone tracing part before 2013 fall semester ends.

References

CRASSIN, C., AND GREEN, S. 2012. Octree-based spare voxelization using the gpu hardware rasterizer. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, July, 303–319. `http://www.openglinsights.com/`. 2

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. *Computer Graphics Forum (Proc. of Pacific Graphics 2011)*. 2

HERTEL, S., HORMANN, K., AND WESTERMANN, R. 2009. A hybrid gpu rendering pipline for alias-free hard shadows. In *Proceedings of Eurographics 2009 Area*. 2

Received:	December 14, 2013		
Recommended:	????-??-??	Corresponding Editor:	??? ???
Published:	????-??-??	Editor-in-Chief:	Morgan McGuire