Sparse Voxel Octree (SVO)

Eric Knapik

WHAT

A pixel is the smallest representation used in two dimensional space. Similarly a voxel is the smallest representation used in three dimensional space.

A sparse voxel octree is a way of representing three dimensional data, voxels. These are then placed in an efficient storage structure the octree. What I have done is create a SVO on the gpu utilizing the hardware rasterizer. This octree can then be used for improved global illumination lighting calculations. There is an incredible parallel between GPU octree creation and with standard ray tracing and global illumination techniques paper describes the process of using the GPU for said octree creation where it can be later used for realtime global illumination.

HOW

Hardware Rasterization

The big idea is how to create voxels from normal triangle data that is typically used under traditional forward or deferred rendering. To render a voxel representation of the world the viewport corresponds to the amount of voxels that will make up the world, this is similar to how the viewport will make up the pixels that represent a screen.

Since the entire world is attempted to be voxelized an orthogonal projection is used so that triangles are not altered depending on their distance from the camera. When rendering the the camera is at the edge of the scene looking into the center of the world. To ensure that a triangle is converted into the best voxel representation each one is rendered according to its dominant axis. That means the camera is looking down the X, Y or Z axis for its position. OpenGL insights gives a good explanation of the process here: https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf
Conservative rasterization has two parts, triangles being rendered along their dominate axis and the enlargement of triangles to avoid holes in the voxelization process. Holes can be created similar to how triangles can be under represented on a pixel grid for which pixel to color, the same thing can happen here since the hardware rasterizer is being used. There are several ways to enlarge triangles, for this project the triangle center is found then the vertices of the triangles are pulled away from this point slightly enlarging the overall triangle utilizing the geometry shader.

Scene	Triangles	Voxels Created	Time
Cornell Box	3136	86664	1 millisecond
PBR Demo	54720	53856	2 millisecond

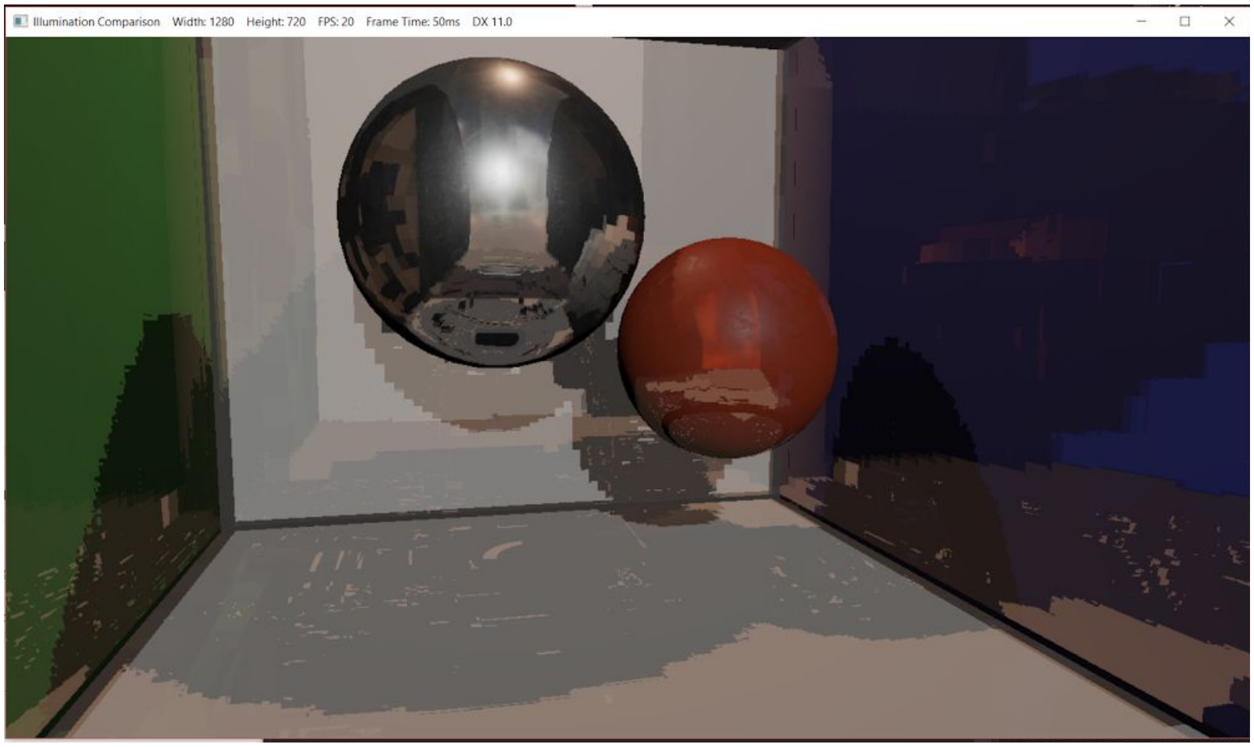

Illumination Comparison Width: 1280 Height: 720 FPS: 20 Frame Time: 50ms DX 11.0

Figure 1: Cornell Scene final rendered image

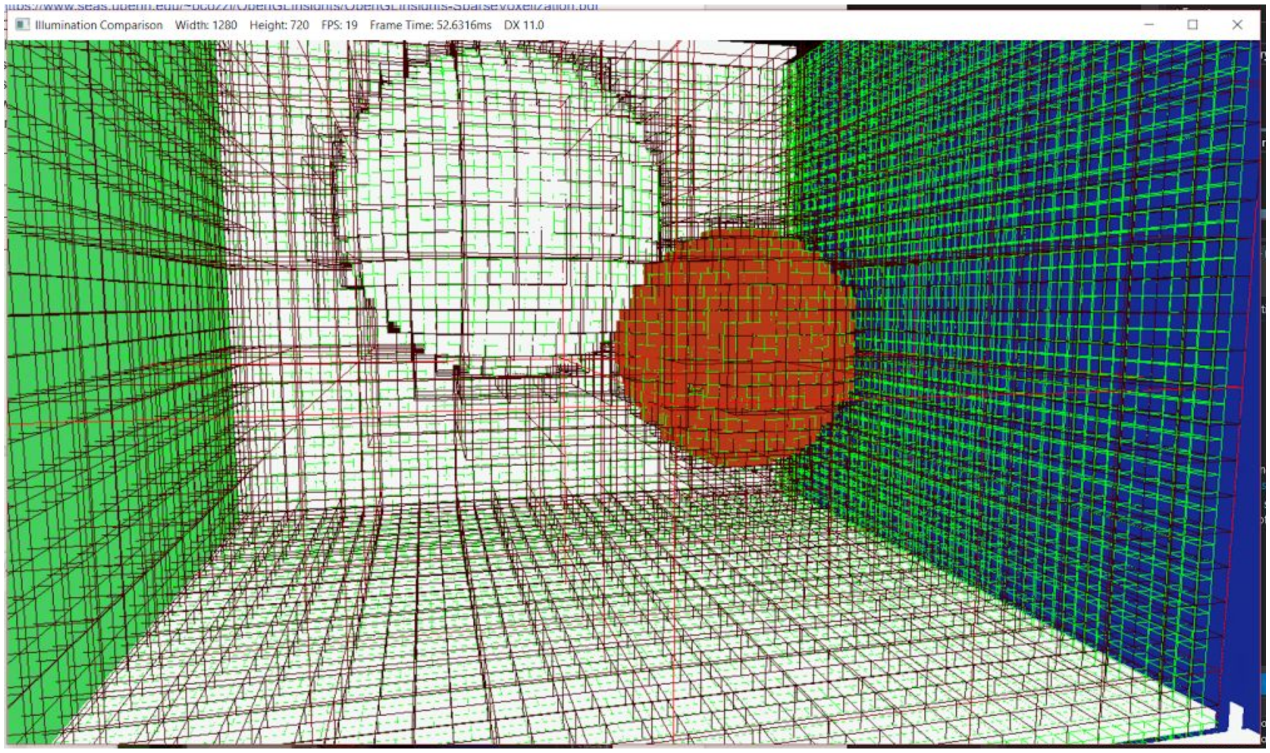

Illumination Comparison Width: 1280 Height: 720 FPS: 19 Frame Time: 52.6316ms DX 11.0

Figure 2: Cornell Scene rendered in octree and voxel debug. Where you can see the final voxels in green and the octree outlines in red

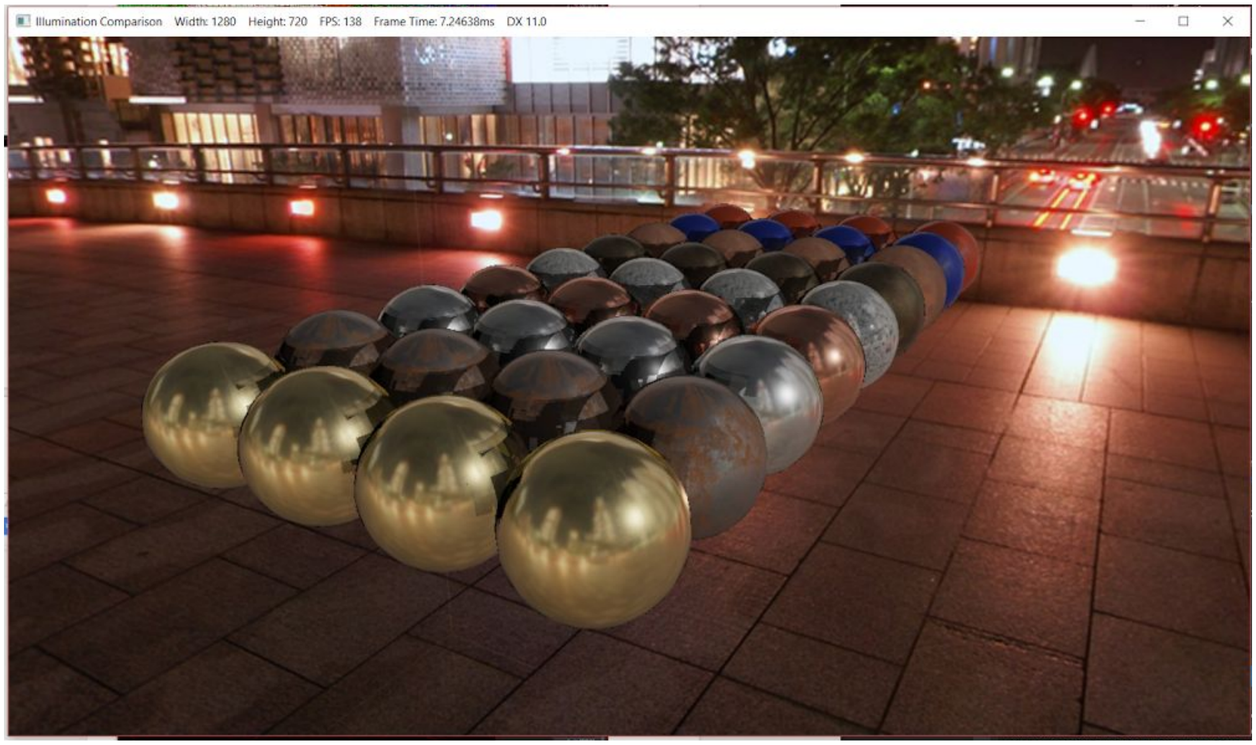

Illumination Comparison Width: 1280 Height: 720 FPS: 138 Frame Time: 7.24638ms DX 11.0

Figure 3: PBR Demo scene final rendered image

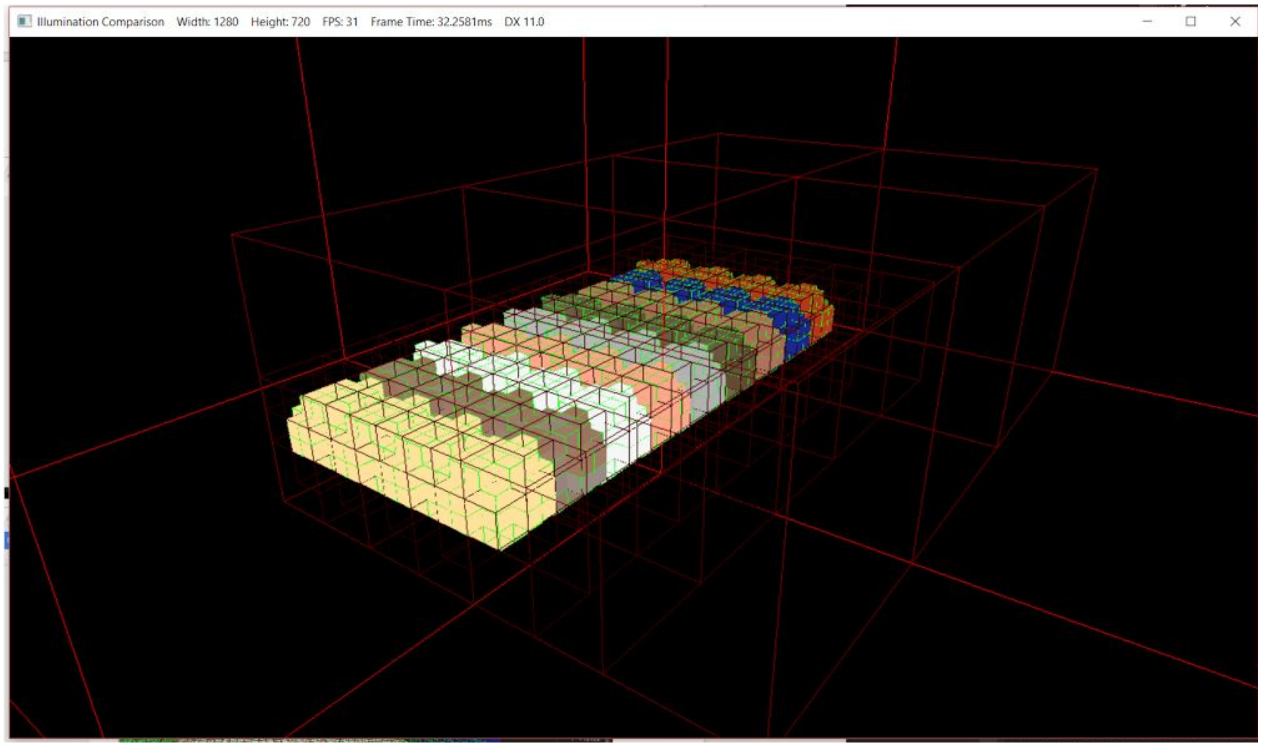
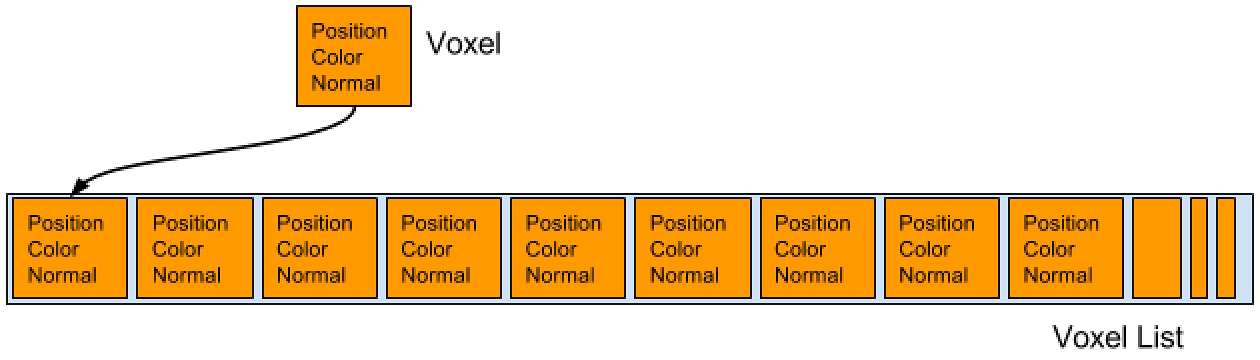

Illumination Comparison Width: 1280 Height: 720 FPS: 31 Frame Time: 32.2581ms DX 11.0

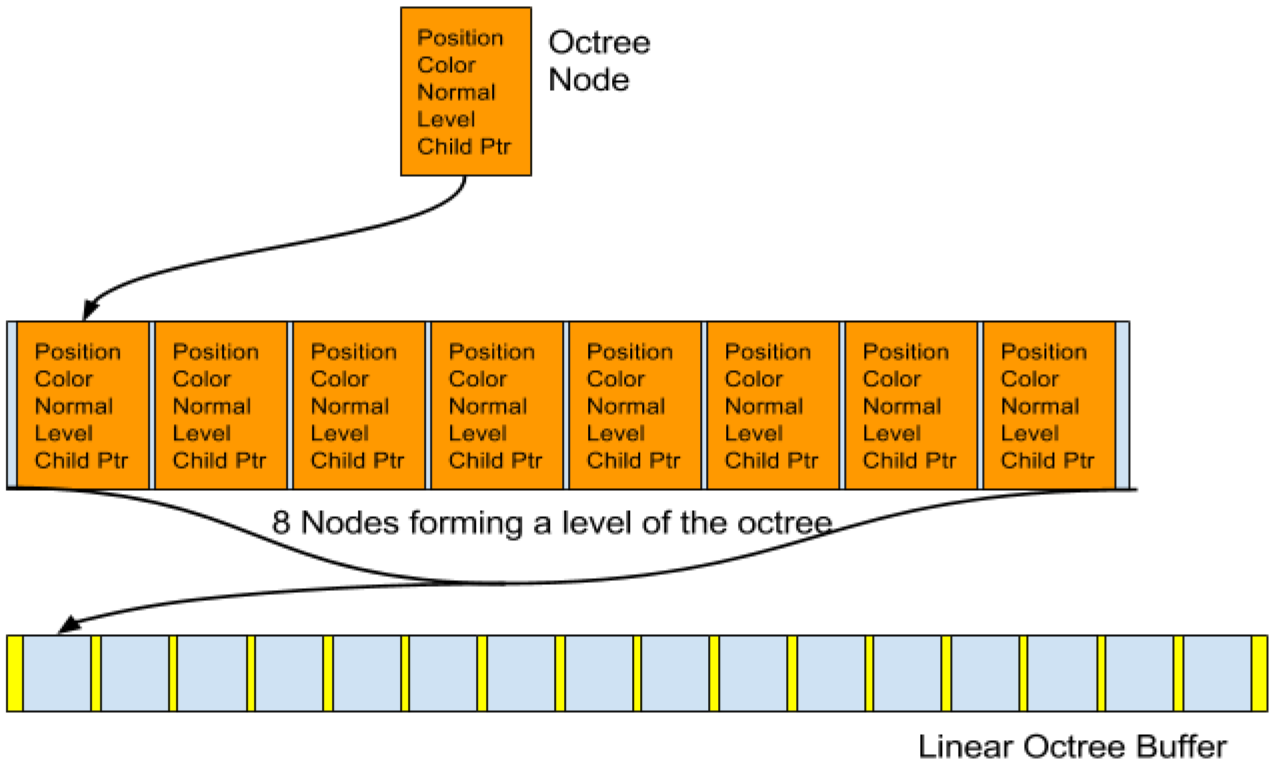
Figure 4: PBR Demo Scene rendered in octree and voxel debug. Where you can see the final voxels in green and the octree outlines in red

The hardware rasterizer was used to convert the triangle data into a discrete voxel representation of the world but what happens to that data and how is it to be stored. Currently when a triangle is converted into a voxel the pixel shader has the positional data of each particular voxel similar to how the pixel shader operates on pixel in a normal rasterization. With this in mind an atomic counter was used to count how many voxels were created from the rasterization process. Once the final count was found a buffer was created of the appropriate size then all the geometry was re rendered exactly the same way and stored in this buffer, the atomic counter can then be decremented to guarantee a unique value for each. This means that to create the voxel representation the geometry was rendered twice through the hardware rasterizer, once to count how many there will be then once to store the voxels in a buffer. This buffer will be referred to as the voxel list buffer or voxel list.



Create Octree

With the scene voxelized and all the voxels stored within the voxel list the sparse voxel octree can now be created. Desirably this should be done on the GPU with compute shaders for the increased parallel processing and not having to do CPU to GPU data transfers. Atomic operations can be used lock buffers for parallel processing on the gpu. This gpu octree is constructed linearly within a structured buffer. Where eight Octree Nodes form a level of the octree, then each Octree Node contains an offset pointer than indexes into the Linear Octree Buffer to the next eight Octree Nodes that form the next more refined level of the octree.



WHY

With a GPU octree representation of the octree the creation is a very quick and efficient process. This only needs to happen once per scene and can then be referenced for all lighting calculations giving improvements to AO and Global Illumination. The GPU octree allows for lighting improvements utilizing ray tracing to know more about the entire scene in an efficient way.

DIFFERENCE

The big difference between my implementation and others is the use of light injection into the octree. The idea was that standard ray tracing through the octree would be equivalent; however, the blocky nature of figure 1 would leads me to believe that the light info sampling from the octree would mitigate these aliased shadows caused by raytracing through the voxel octree.

BUT

The biggest drawback to SVO is the memory consumption required for high fidelity in the octree. With higher number of voxels representing the world space the resolution increases but this takes up much more space.

The marching through the octree can also be an issue once a proper octree is created. A marching algorithm needs to traverse the octree according to the particular level for each step since there is no knowledge of octree neighbors besides the current level. This traversal can be costly and a time consuming process. Along with traversal many algorithms and this included suffers from traversing through the thin voxels missing walls causing artifacting or light leaking.

In the majority of this project the results are very blocky and clear bounds of the voxels can be noticed. An antialiasing strategy might possibly be implemented where the adjacent voxel neighbors outside of the current are know and can be sampled. This would smooth out the rough edges.

A design choice for debug rendering the position of each voxel does not need to be stored inside the octree but can be derived through the traversal, it was simply easier to have the increased memory consumption for improved octree debug rendering.

Multiple cone tracing rays were attempted for a smoother render but the increased render time did not visually improve the results for the final scene so this idea was scrapped and could possibly be revisited.

Visual fidelity is believed to be reduced by NOT performing LIGHT INJECTION where light is sent into the octree to illuminate and bounce around the voxels where the light is then mip mapped up the octree and the final render will sample the octree for lighting information.

WHO?

Chris Cascioli for supervising

LINK TO PROJECT: https://github.com/EKnapik/IlluminationComparison

Previous Research:
http://simonstechblog.blogspot.com/2013/01/implementing-voxel-cone-tracing.html

http://developer.download.nvidia.com/assets/events/GDC15/GEFORCE/VXGI_Dynamic_Global_Illumination_GDC15.pdf

https://research.nvidia.com/sites/default/files/publications/GIVoxels-pg2011-authors.pdf