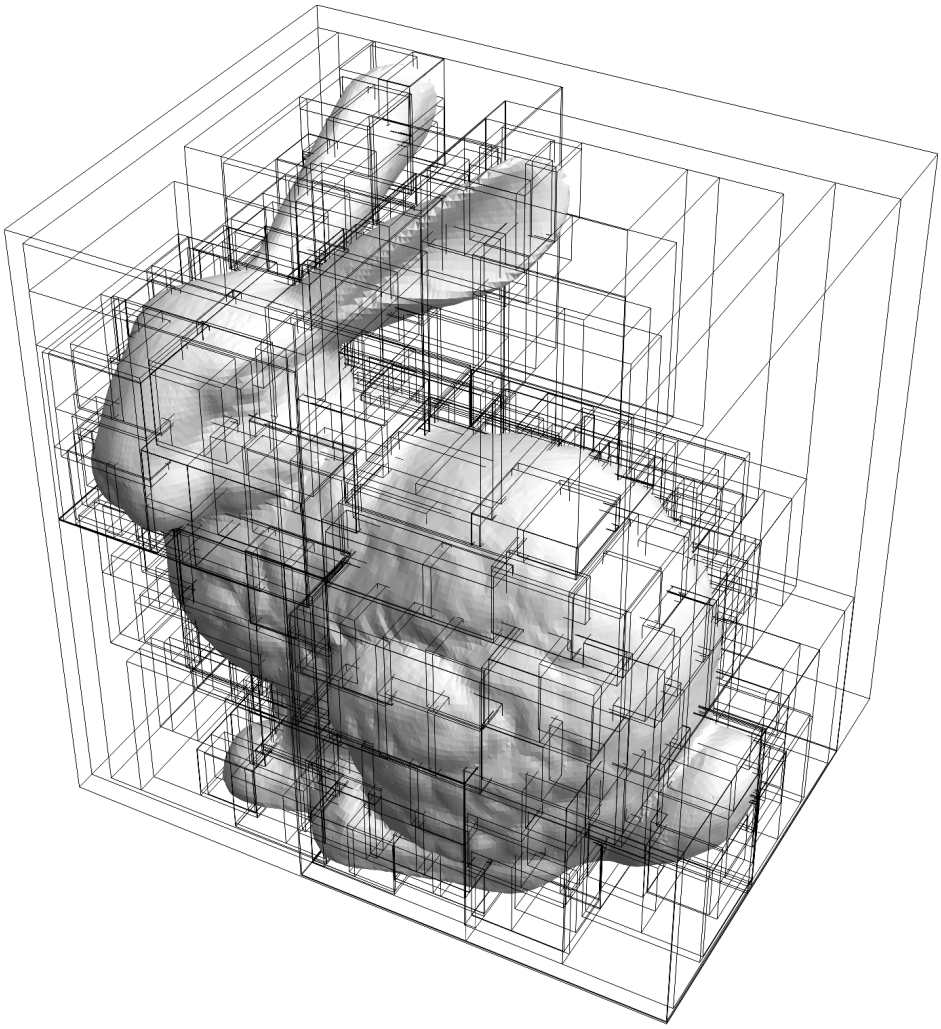


Ray Tracer Version 1

Alec Shedelbower

October 14, 2018

HARDWARE

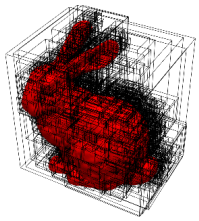
Description

I'm running my ray tracer on a PC I built with a **GTX 1080 Ti** GPU. The PC is affectionately known as PANDA.

LANGUAGE

Description

I am using a combination of Mathematica (Wolfram Language) and CUDA (C++). The interface for my ray tracer exists in a Mathematica notebook. My Mathematica code then calls custom CUDA functions I wrote using CUDALink (https://reference.wolfram.com/language/CUDALink/tutorial/Overview.html).



ACCELERATION STRUCTURE

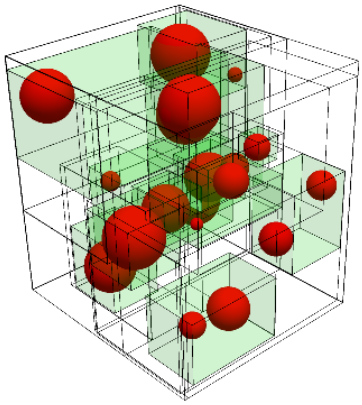
Description

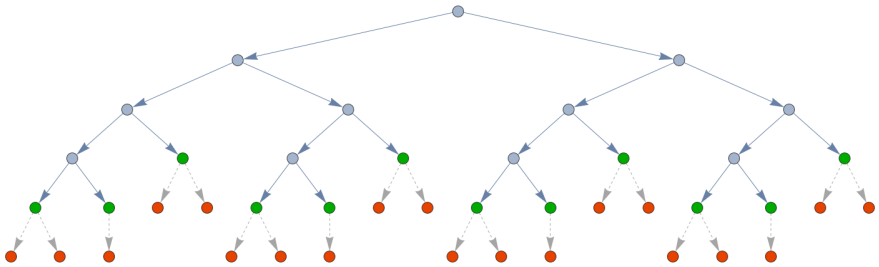
My ray tracer uses a Bounding Volume Hierarchy for its acceleration structure. It implements the Split-Equal method for subdividing the primitives. The maximum depth and maximum number of primitives per leaf node can be easily adjusted. For most of my renders I've been using a maximum of 3 primitives per leaf node.

Visualization

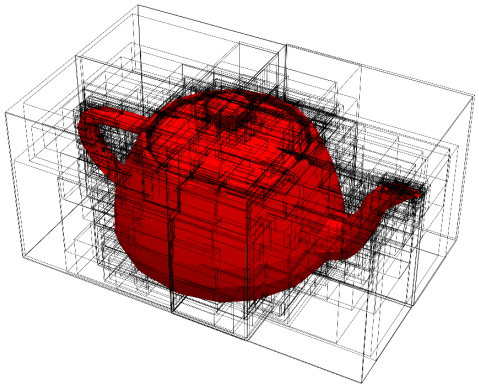
While I was testing my BVH, I made a lot of functions to visualize it. I thought I would include them here as well. Note that the graphics in the two examples below were **not** made with my ray tracer, but with built-in Mathematica graphics functions.

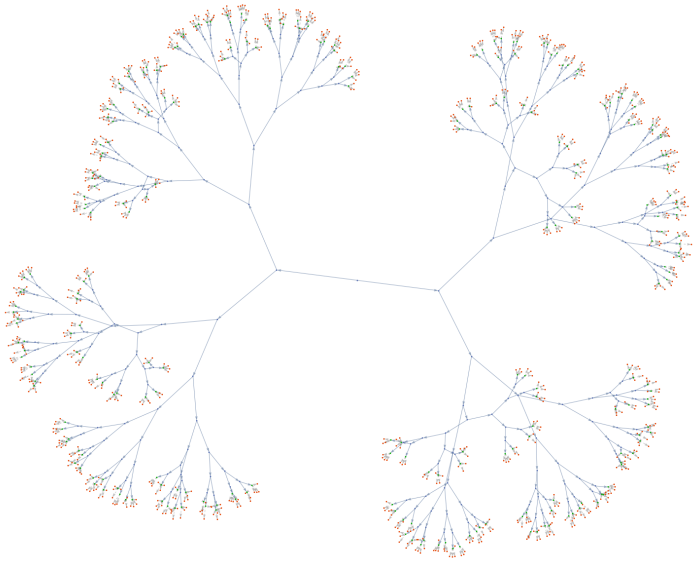
Example Small (20 Random Spheres)





Example Medium (Utah Teapot)

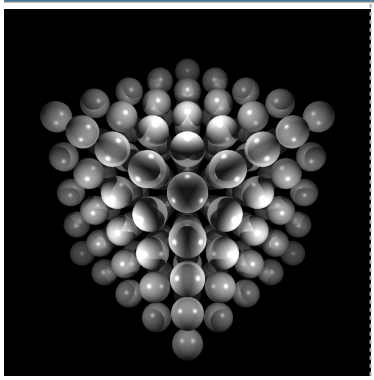
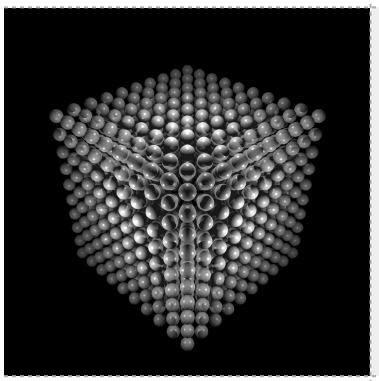
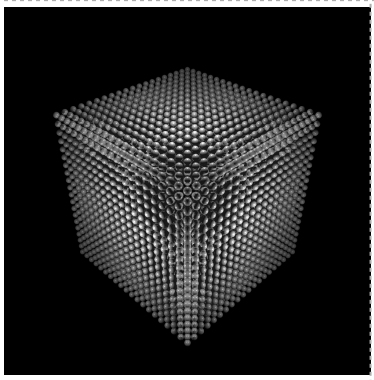


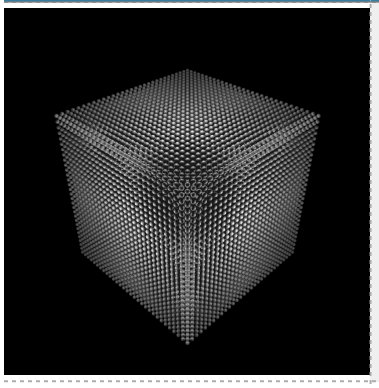
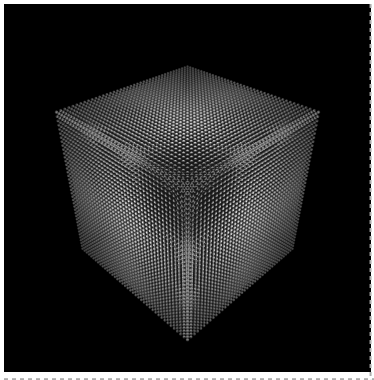
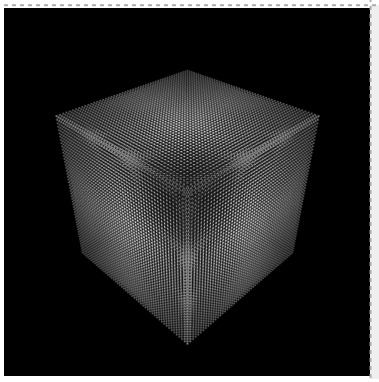


SPHERE BENCHMARKS

Scene Description

Each scene below consists of a large amount of spheres organized in a cube shape. The scene is lit using only 3 white point lights. The output images are each 1000x1000 pixels with antialiasing of 4 samples per pixel (total of 4 million rays per image).

Output Image	Spheres	BVH Construction Time	Ray Time	Trace Time	Total Time
	125	0.01	0.70	3.18	3.89
	1,000	0.01	0.71	3.40	4.18
	8,000	0.48	0.74	3.86	5.08

Output Image	Spheres	BVH Construction Time	Ray Time	Trace Time	Total Time
	27,000	2.59	0.71	4.88	8.18
	64,000	11.28	0.69	6.21	18.18
	125,000	43.75	0.72	9.53	54.01

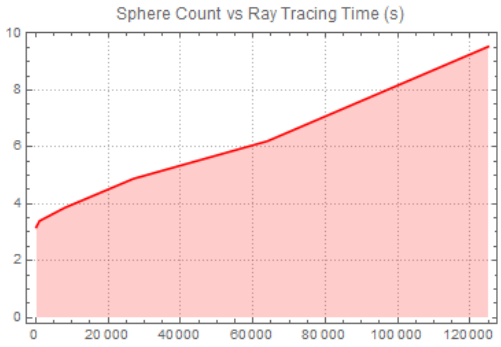
Benchmark Analysis

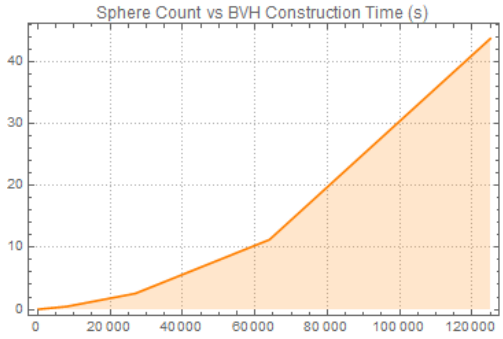
My rendering process can be split up into 3 distinct parts.

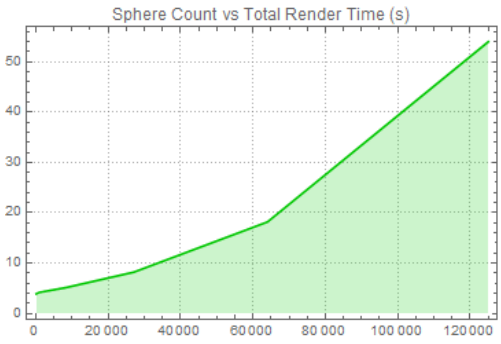
1. Construct the BVH
2. Create the rays to cast *(constant for a given resolution and antialiasing)*
3. Trace the rays into the scene *(calculate ray-primitive intersections and shading)*

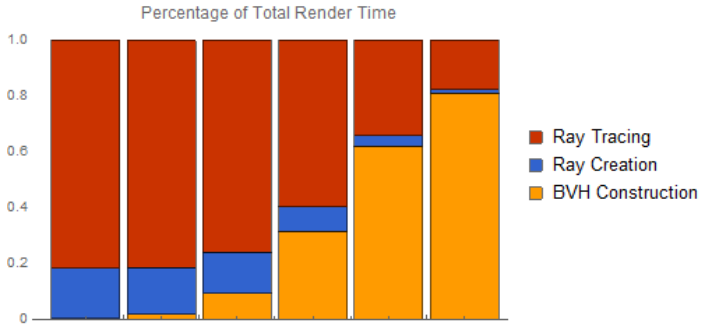
The actual ray tracing time is linear with the number of objects in the scene (shown in red plot below). Unfortunately, my algorithm for BVH construction appears to be quadratic (shown in orange plot below). Therefore, the total time to render a scene is quadratic (shown in green plot blow).

For small scenes, the time to construct the BVH is negligible, however after ~50,000 objects it surpasses the ray tracing time. The time percentages for each of the 3 processes is shown in the final chart below.









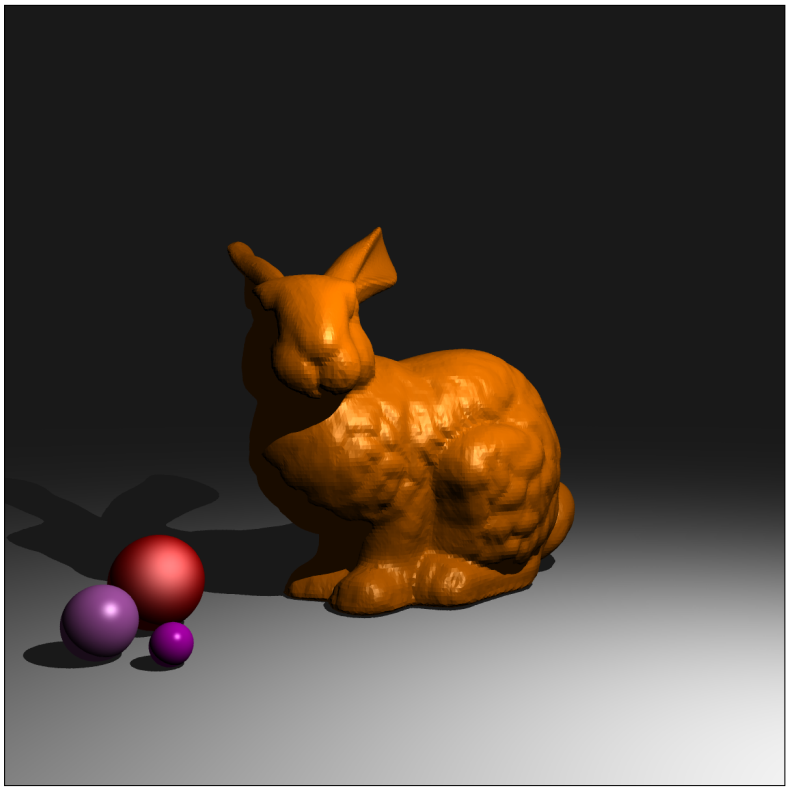
MESH RENDERING

The image shown to the right was rendered both with and without using the BVH acceleration structure.

The scene consists of 3 spheres, 2 triangles forming a plane, and a mesh made of 69,451 triangles. There is a single point light in the scene.

The image is 1200x1200 pixels with an additional antialiasing of 4 samples per pixel. (Total of 5,760,000 rays)

The corresponding times are shown in the table below. Note that the times do not include the amount of time it took to construct the BVH.



	Time (seconds)	Time (minutes)
With BVH (10 layers deep)	7.83	0.13
Without BVH	2123.35	35.39

From above, we can see that the BVH makes our rendering time over **250 times faster**!

ENVIRONMENTAL REFLECTIONS

I went ahead and implemented a basic skybox and environmental reflections.

Skybox

The skybox uses a single image (taken from a 360 camera) to use as the background. I am currently using this approach instead of a cube map with 6 separate images because it was quicker to implement.

Rays that do not intersect with any scene objects are considered to hit the skybox. When this occurs, they are converted to a point on the unit sphere. Next, I project this point to a 2D (u,v) vector which I can then use to index into the skybox image.

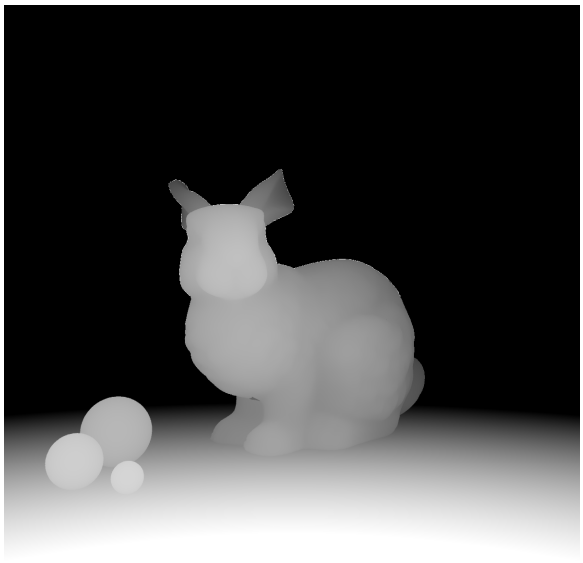
Reflections

If a material has a certain shininess exponent, I consider it to be a perfect mirror. In this case, instead of my usual rendering equation, I calculate the reflection vector of the ray off the surface and cast that against the skybox. There are currently no object-to-object reflections.

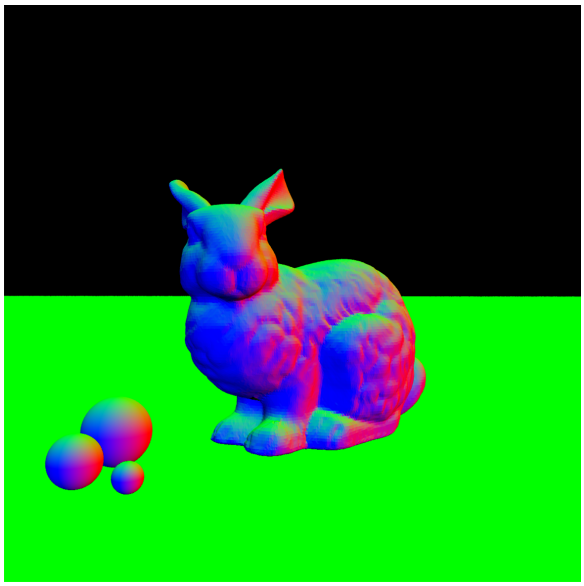
Static Examples

MISCELLANEOUS

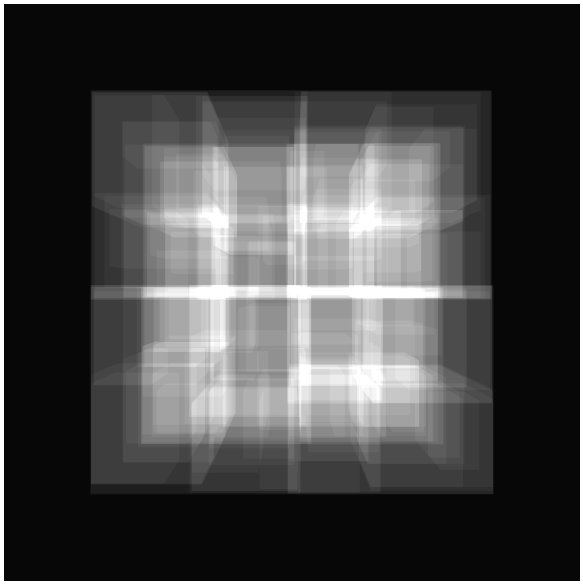
Here are some other graphics I produced while making the ray tracer, either for debugging purposes or because I thought they looked cool.



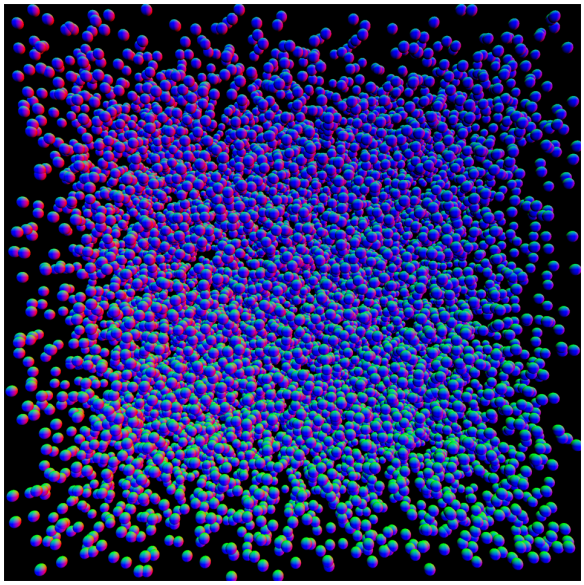
Depth Values



Normals



BVH Node Checks Per Ray
Lighter = More Nodes Checked



Random Spheres (Normals)