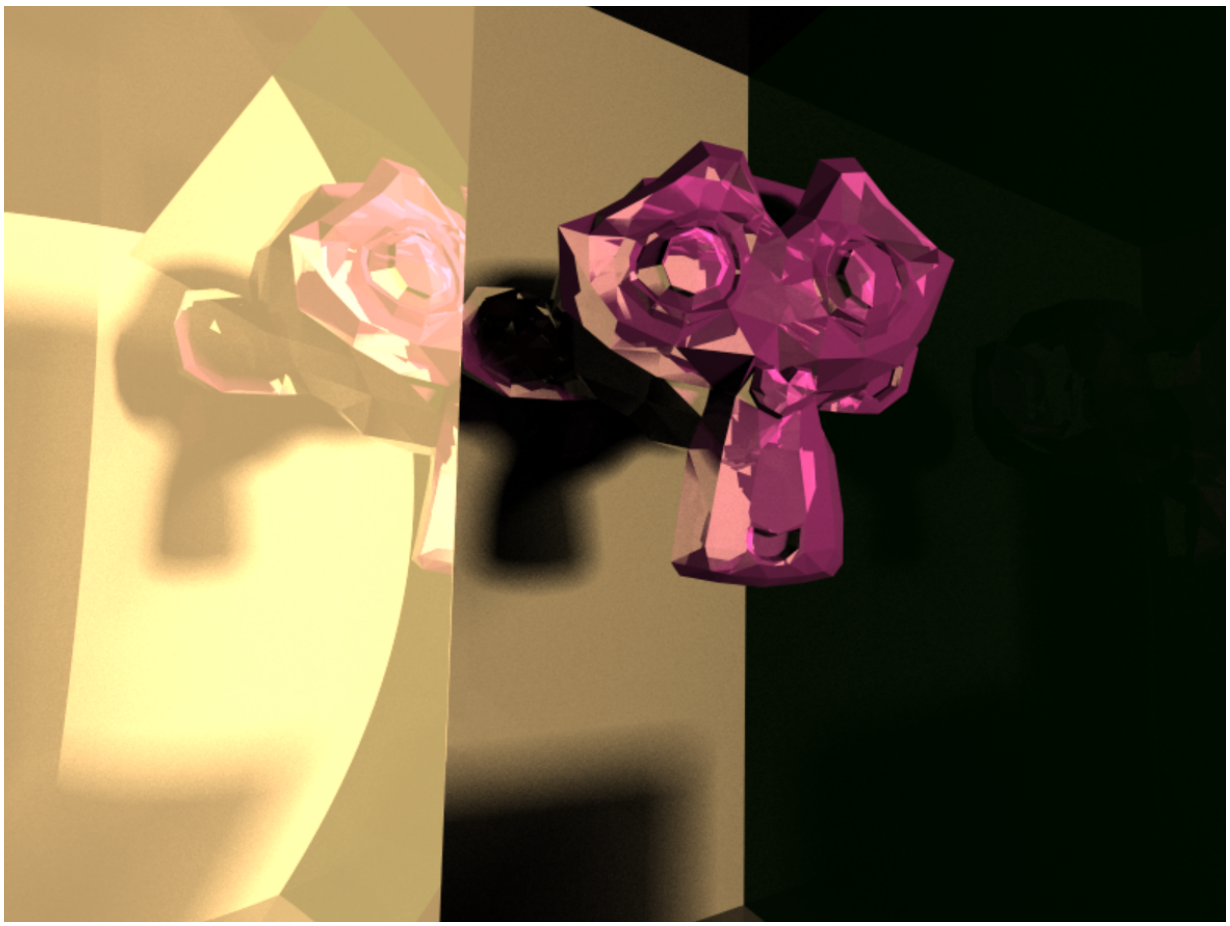
# Adaptive Assignment for Real-Time Raytracing

**Paul Aluri** [paluri] and **Jacob Slone** [jslone]



**Carnegie Mellon University**
**15-418/618 Spring 2015**

**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

# Summary

We implemented a CUDA raytracer accelerated by a non-recursive bounding interval hierarchy (BIH) and experimented with different methods of adaptive assignment of rays to CUDA thread blocks to minimize SIMD divergence.

# Background

Our project involved two major aspects to achieving a real-time raytracer capable of almost instantly producing high-fidelity renders from scenes with thousands of vertices.

The first component was building a data structure that would allow for amortized logarithmic search time when computing intersections. We implemented this data structure for both the objects and faces in our scenes. This optimization significantly increased the framerate of our renderer, especially in scenes with higher numbers of vertices/faces (since the penalty of linear rather than logarithmic lookups grows with the number of vertices/faces).

The second component was experimenting with methods for reassigning rays in CUDA thread blocks in order to minimize the amount of SIMD divergence caused by rays bouncing off objects differently. The amount of computation that our raytracer performs is directly related to the maximum number of bounces we allow our rays to take (which is directly related to how visually realistic the rendered scenes are). Because we were aiming for higher quality renders, we allowed our rays to bounce up to 5 times. A consequence of this is that even nearby rays can vary drastically in what they reflect off and also in how many times they reflect. Raytracers are very data-parallel, with locality in thread blocks due to the fact that similar mesh/face data will be accessed by threads in the same thread block. However, given the possibility for highly divergent rays existing in a block, SIMD vector utilization can be poor and potential performance benefits can be lost.

To counteract this, we came up with a couple of strategies for assigning rays. Neither proved to be groundbreaking. The first strategy was to count the number of bounces each ray made, sort all rays by the number of bounces, and reassign them to thread blocks in that order, so that rays with similar number of bounces would be assigned together. The second strategy was more complicated. We came up with a heuristic which attempted to model the intersection lookups in our BIH by creating a bit vector that represented the path that the ray followed down the BIH tree. Similar to the first strategy, we then sorted the rays by their path vector, hoping that rays with similar paths would be blocked together. This second strategy performed much poorer than the case of not reassigning rays at all (by default thread blocks are assigned 2D chunks of contiguous pixels). The first, simpler strategy proved to be slightly better than the default assignment method, although not significantly.

**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

# Approach

### Code
This project required the use of software libraries and code bases.

1. 3D Asset Importing - Assimp Open Asset Import Library (http://assimp.sourceforge.net)
2. OpenGL Rendering - FreeGlut and GLEW
3. Pieces of raytracer code - Jacob's raytracer

We decided to use NVIDIA GPUs and Cuda to accomplish our goals. We targeted mid-range gaming GPUs, although performance could easily be tailored to various GPUs by scaling the quality of the raytracer up/down (e.g., change the max number of bounces that rays can take).

### Data Structures
We calculated the bounding interval hierarchy data structure on the CPU and then copied it over to the GPU, where it was accessed by rays in order to alleviate the amount of computation they had to perform when determining their intersections.

For the mesh, face, and vertex data, we loaded in the scene data using the Assimp library, reformulated it to meet our needs, and sent it to the GPU.

The literature was conflicted on the best heuristic to use for splitting in the BIH, and given more time we would have liked to test different heuristics (e.g., surface area), but we decided to use the standard heuristic of splitting in the middle of the longest axis of the current bounding box.

A major feat of our BIH intersection implementation was the ability to completely avoid recursion. This gave us substantially higher vector utilization than the naive implementation while traversing the tree, limiting the divergence to intersection tests on the leaves of our tree.

We were also able to leverage C++ templating to keep the BIH for our scene in the device's constant memory, allowing for substantially faster reads when traversing the scene.

### Mapping
Rays were mapped to Cuda threads.  Each ray has a pixel position, and simulates the path that light travels from a light source to the 3D point located at the ray's 2D pixel position.

### Iteration/Optimizations
It took us many more hours than we would have liked to get a functional raytracer working.  Although we began with code libraries for importing assets,pieces of raytracer code, and OpenGL rendering, it took a while to find some tricky bugs in our code and get a scene rendering, but it was definitely extremely satisfying once we got a scene rendering to incrementally make visual improvements on the raytracer.  The first part of our iteration involved just that—implementing and improving features of our raytracer to provide aesthetically appealing, realistic renders.

Once we were satisfied with the visual aspect of our raytracer, we moved onto the next portion of our iteration, implementing the bounding interval hierarchy for our objects and faces.  Once we got this data structure working, we saw immediate and drastic speedups for our renders, particularly scenes with high face-count meshes.

Finally, we began iterating with different heuristics for reassigning rays to Cuda thread blocks.  We tried both the ray bounce and BIH bit path optimizations.  In each of these methods, for each ray we kept track of the heuristic, sorted the rays by the heuristic value (using thrust) and then reassigned rays to blocks in this new order.

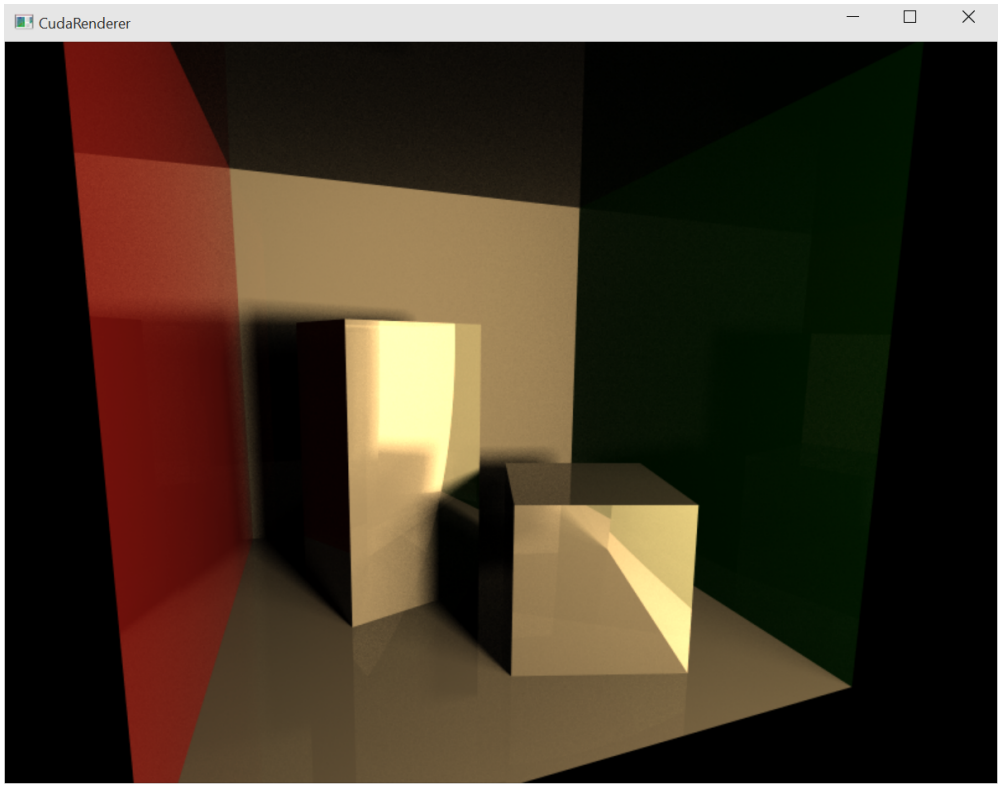
# Results

### Test Scenes
These test scenes were chosen because of their increasing complexity in terms of number of vertices as well as their affinity for rays bouncing several times on several different objects.

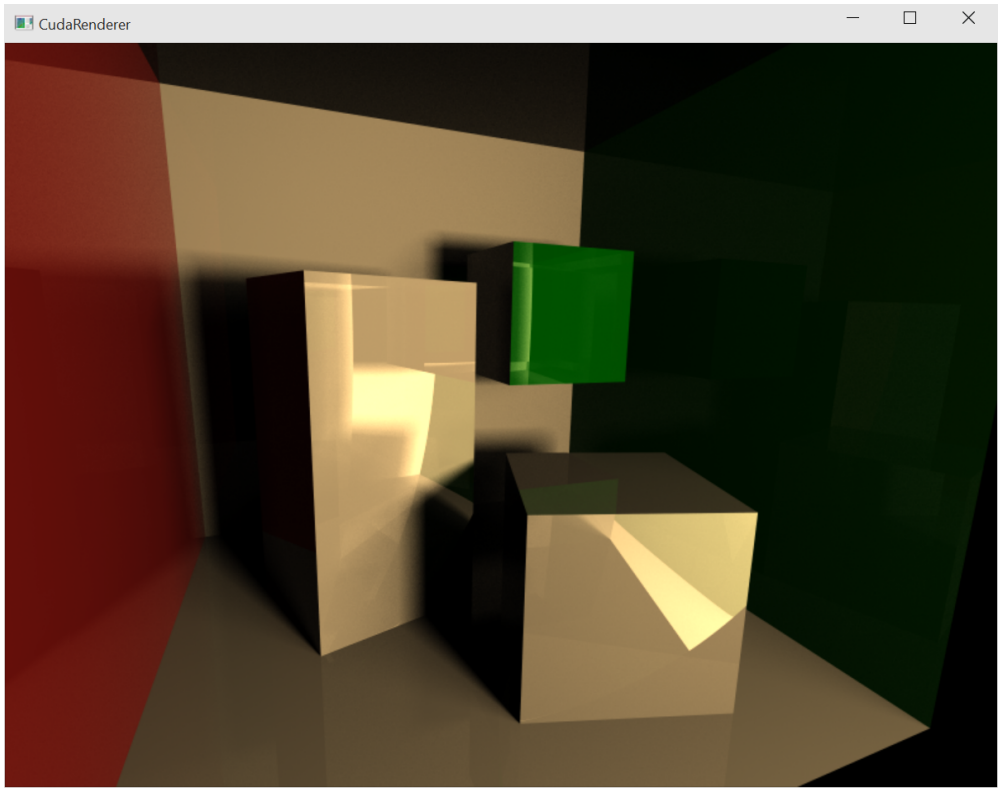
### Hardware
We used an NVIDIA 760 GTX GPU to gather our data.

**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

1. Base - Vertex Count: 60

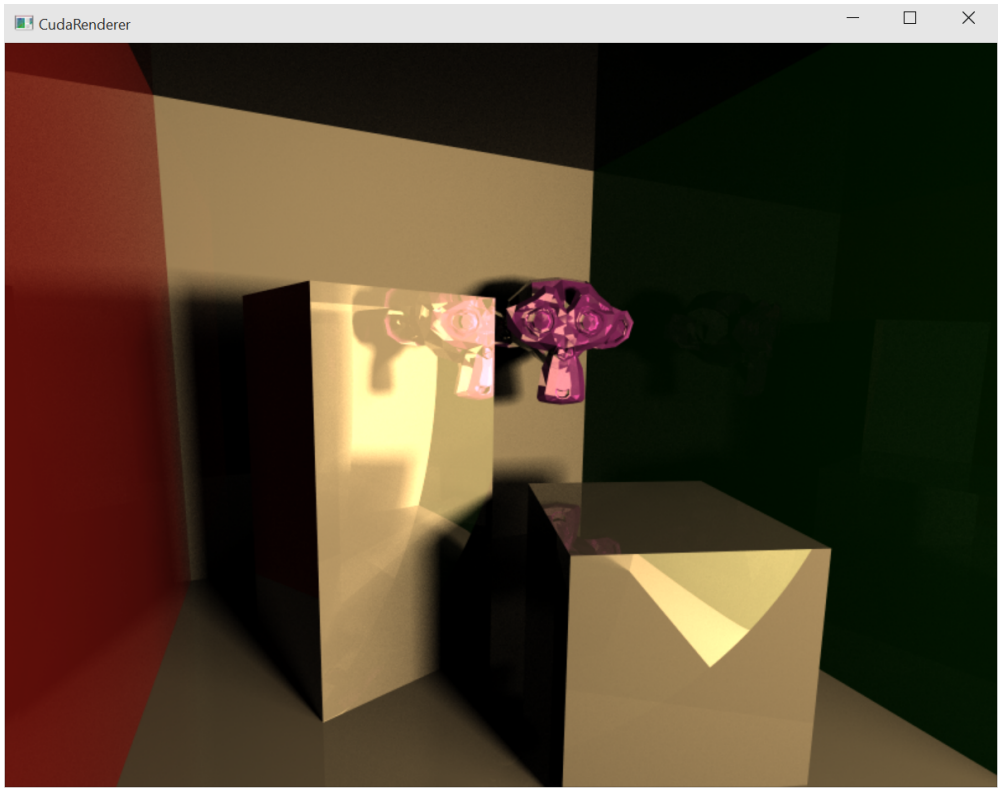



2. Cube - Vertex Count: 68

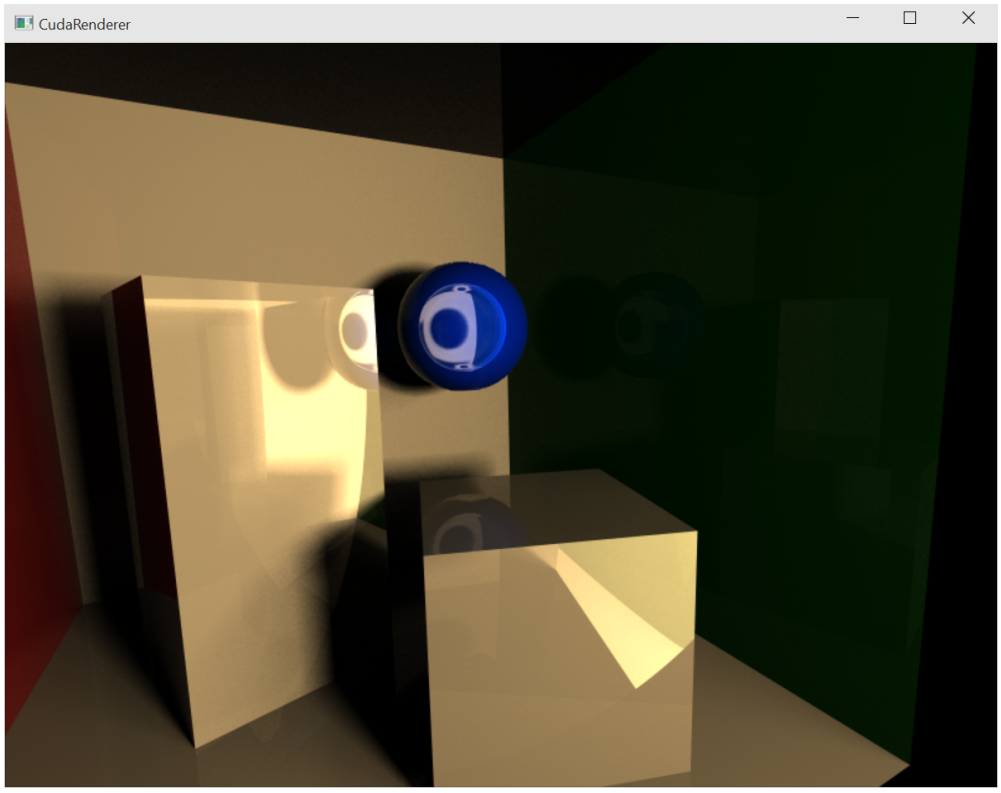


**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

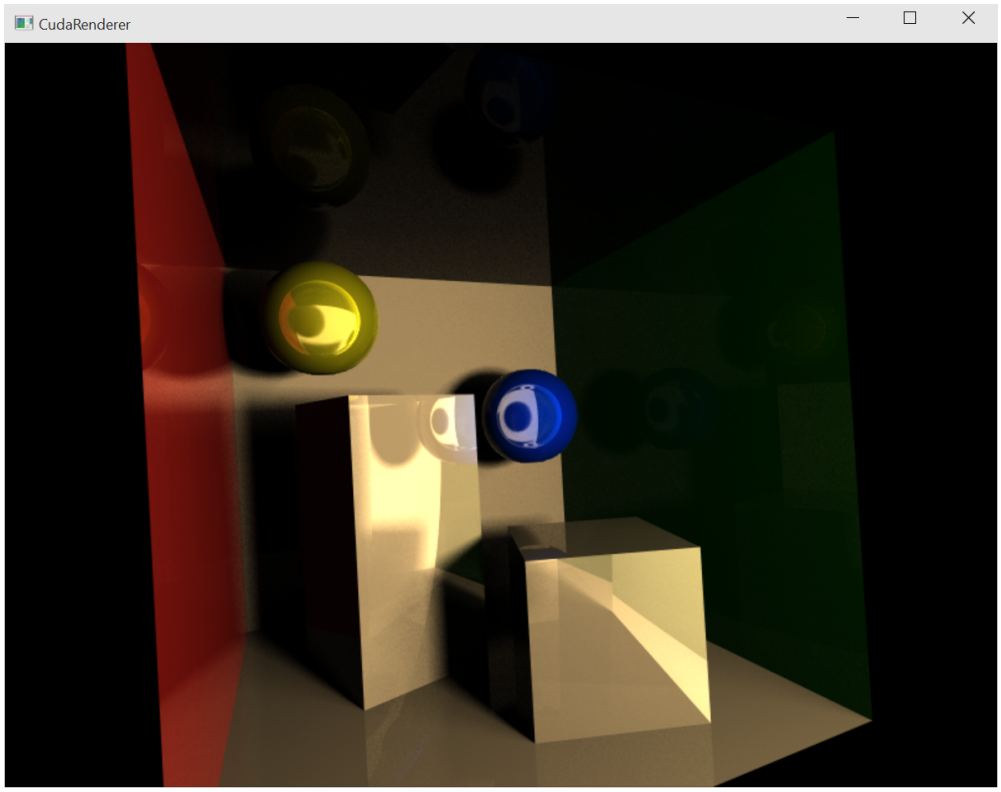
3. Monkey - Vertex Count: 567



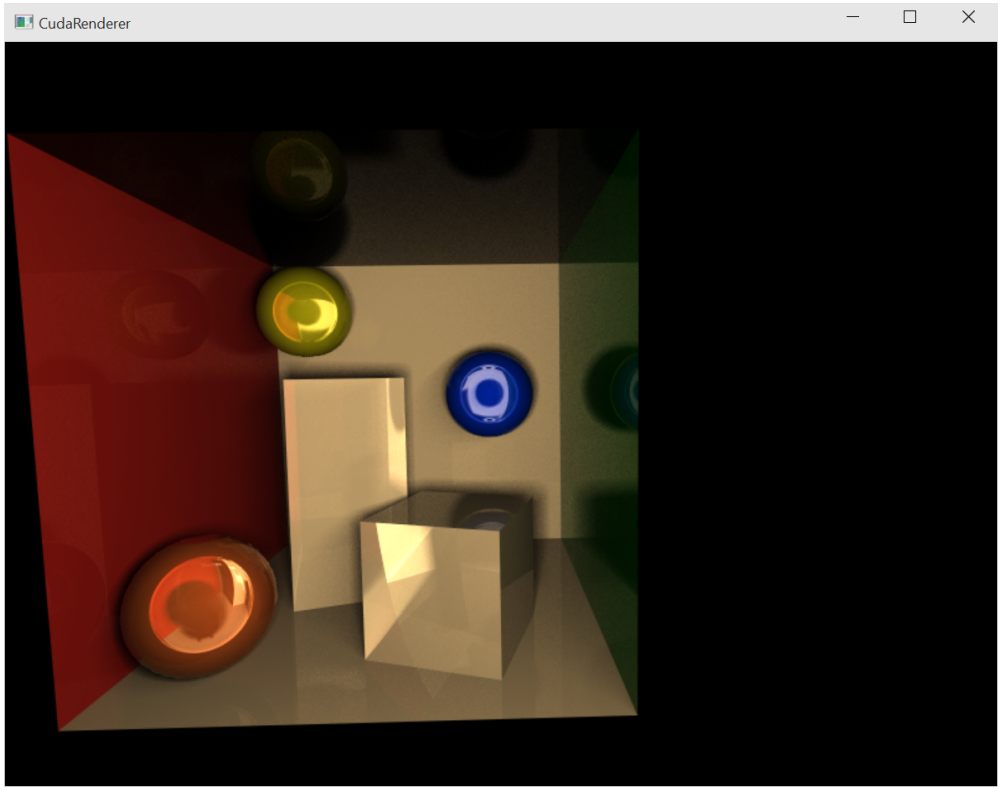

4. 1 Sphere - Vertex Count: 2046



5. 2 Spheres - Vertex Count: 4032

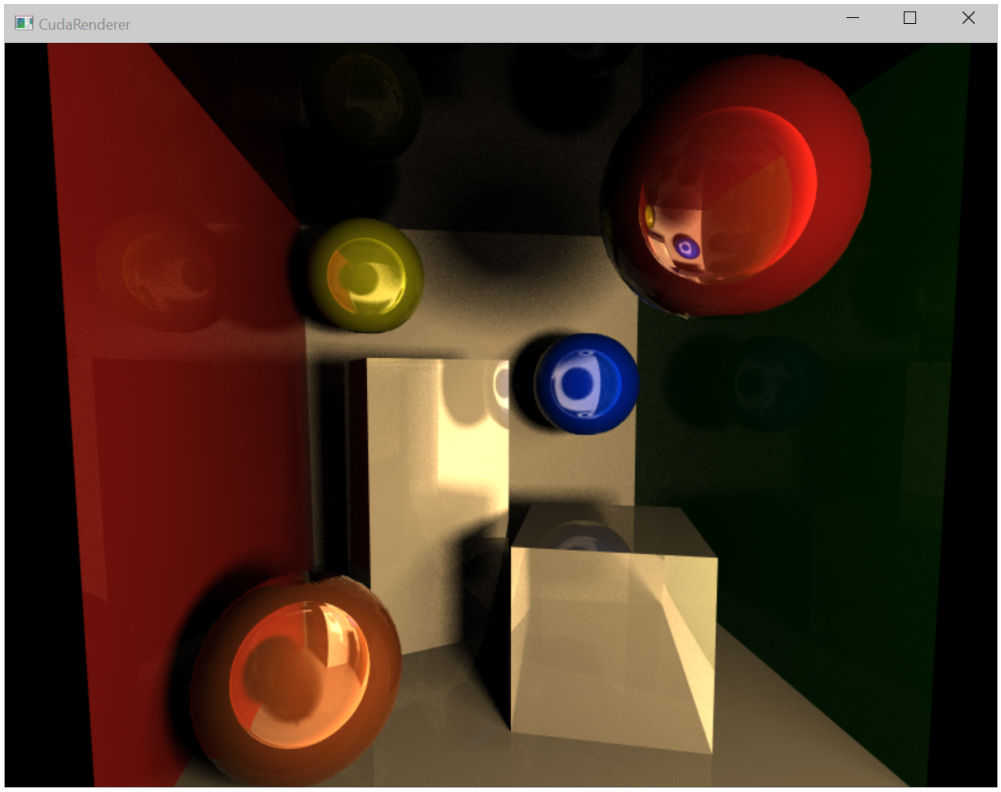



6. 3 Spheres - Vertex Count: 6018

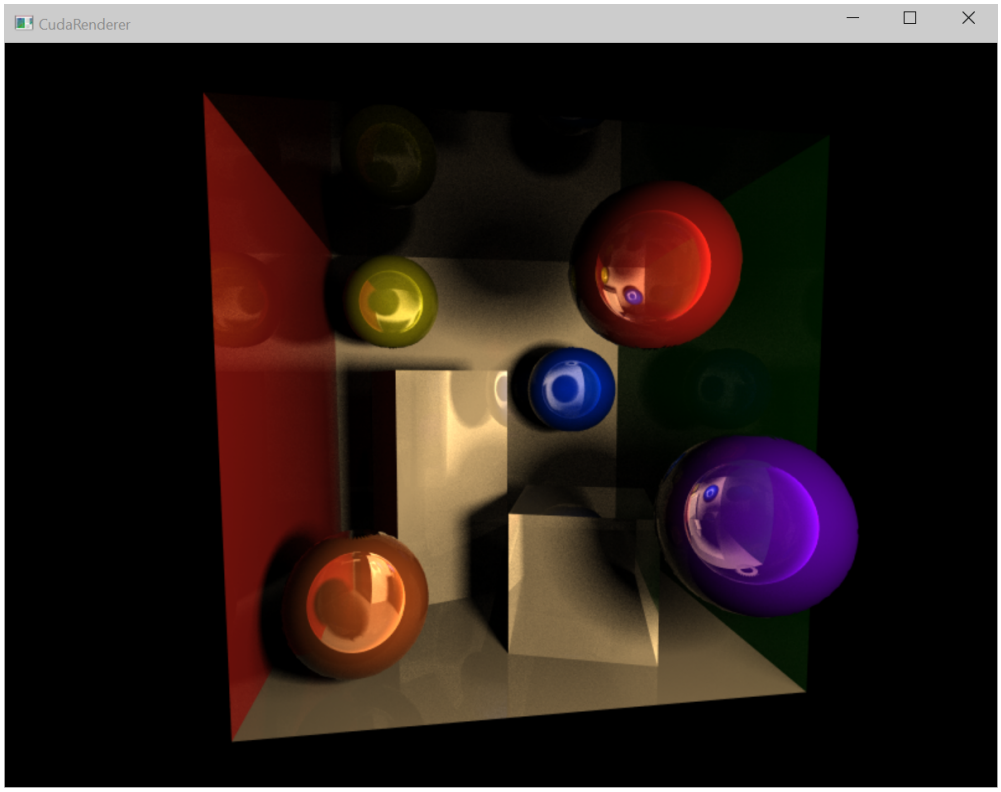


**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

7. 4 Spheres - Vertex Count: 8004




8. 5 Spheres - Vertex Count: 9900



**Paul Aluri** [paluri]
**Jacob Slone** [jslone]
11 May 2015

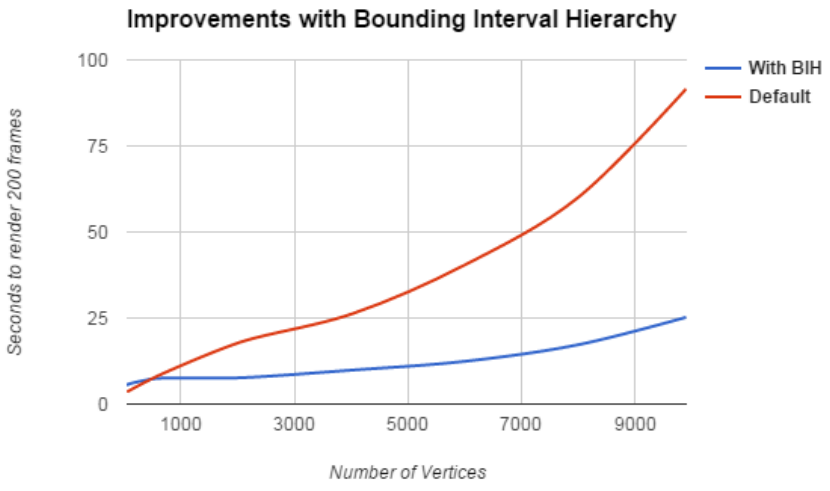
## Methodology

For our metric, we chose to measure the amount of seconds it took for each scene to render 200 frames. This can easily be converted to an average framerate over 200 frames, but we decided that using the time metric would be more easily understandable.

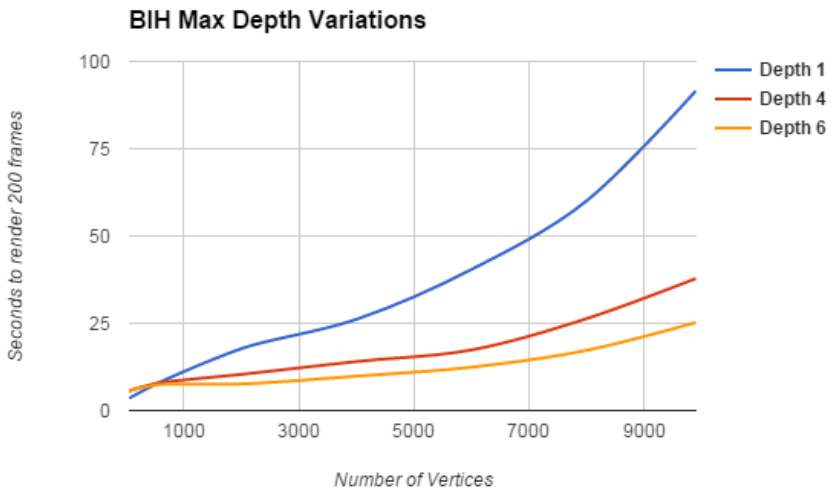
For each of the eight test scenes, we measured the time on seven different setups:

| Test Case | BIH | Reassignment |
|---|---|---|
| **1** | yes | no |
| **2** | yes | bit path |
| **3** | yes | bounce count |
| **4** | no | bit path |
| **5** | faces | bit path |
| **6** | objects | bit path |
| **7** | yes | bit path |

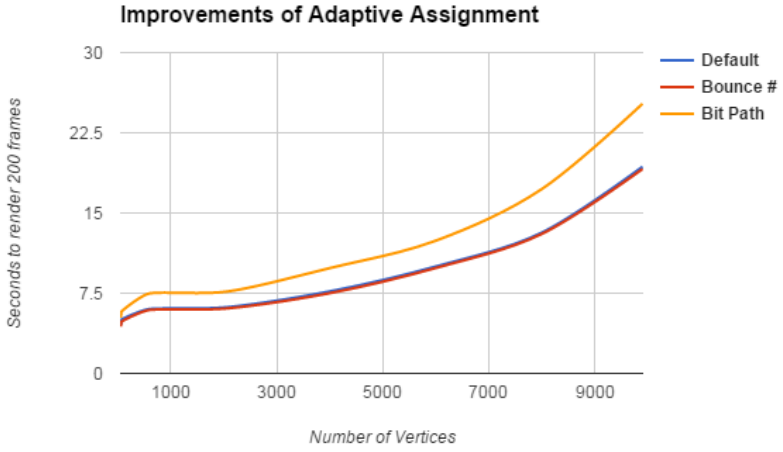
## Findings



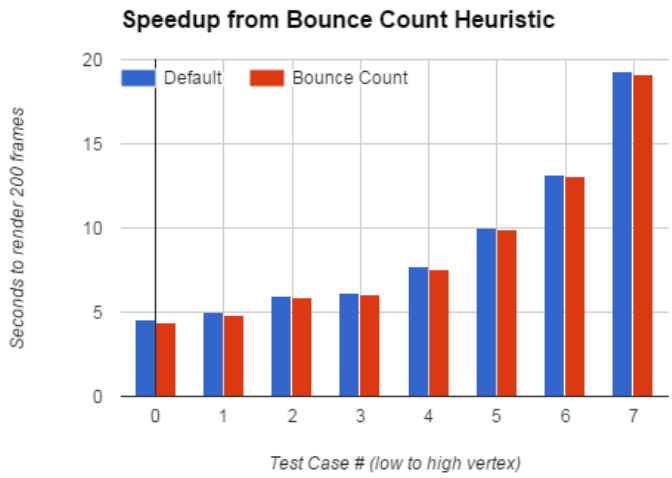

This above chart is not necessarily surprising, but it was definitely encouraging to see such a significant performance increase when reducing the complexity of our intersections from $O(n)$ to $O(\log n)$ time. Although the overhead of the data structure caused slowdowns with (unrealistically) low vertex-count scenes, it became exceedingly more efficient with higher numbers of vertices.




As part of our testing, we tried setting the max depth for our BIH to different depths (depth 1 being essentially equivalent to a linear lookup). Testing in this manner led to the conclusion that for our test cases max depth 6 was optimal, although this number should grow logarithmically with the number of vertices in order to achieve optimal results.




The above graph shows the different effects of three different assignment methods, the standard, bit path, and bounce count methods. The bit path resulted in noticeable slowdown from the default case. This is likely due to poor memory utilization, rather than divergence, but much more sophisticated testing would be necessary to conclude this. The bounce count heuristic actually proved to be slightly faster than the standard assignment, as seen more clearly in the bar graph below.




The speedup is not significant, but it does exist. And this fact alone provides encouragement for performing further research for the bounce count method as well as hope that reassigning rays in an adaptive manner is not only theoretically optimal, but also realistically achievable with current systems. We speculate that perhaps one clear approach to making the heuristic more system-friendly would be to optimize how memory is accessed, because given our setup, lots of memory locality is lost. We also speculate that it might be inherently difficult to achieve significant speedups due to adaptive assignment of rays because it is possible that the assumption that close pixels have similar rays is already a decent heuristic for assigning rays, making the standard assignment of pixel chunks to thread blocks a reasonable approach to combatting ray divergence.

# References

Kinkelin, Martin. 'GPU Volume Raycasting Using Bounding Interval Hierarchies'. *Masterpraktikum aus Computergraphik und Digitaler Bildverarbeitung* (2009).

Mukundan, R. *Advanced Methods In Computer Graphics*. London: Springer, 2012.

Pharr, Matt, and Greg Humphreys. *Physically Based Rendering*. San Francisco, Calif.: Morgan Kaufmann, 2010.

Wachter, Carsten, and Alexander Keller. 'Terminating Spatial Hierarchies By A Priori Bounding Memory'. *2007 IEEE Symposium on Interactive Ray Tracing* (2007): 10 May 2015.

Wald, I. 'Fast Construction Of SAH Bvhs On The Intel Many Integrated Core (MIC) Architecture'. *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012): 47-57. 10 May 2015.

# List of Work by Each Student

Equal work was performed by both students.