Bounding Volume Hierarchy

Making our renderer go brrrrr

Before implementing rendering acceleration with a BVH, our pathtracer used a naive algorithm wherein every ray would be checked against every `Primitive` in the scene, leading to a very slow $O(nm)$ runtime with n primitives and m rays. To speed this up, we used a BVH constructed with the median heuristic. We sorted the list of `Primitive`s based on the longest axis of the bounding box, and then used the median as the split point. Another heuristic we implemented was the midpoint heuristic, but both produced approximately equal results from our testing, so we kept the simplicity of the median approach to avoid the edge case of creating an empty node. To construct a BVH, we performed the following algorithm:

1. Generate a `BVHNode` containing all of the primitives.
2. If the list of `Primitive`s passed is has less elements than `max_leaf_size`, return a `BVHNode` containing all of the `Primitive`s.
3. Otherwise, compute the size of the bounding box of the `BVHNode`, and choose the maximal axis.
4. Sort the list of `Primitive`s based on their position in the maximal axis, and split into two equal lists.
5. Set the `l` (left) member of the `BVHNode` to a `BVHNode` constructed from the first half of the list, and the `r` (right) member to be a `BVHNode` constructed from the second half of the list.

Comparing Rendering Times

Below, we compare output times of rendering the (really cute) Cow with and without BVH acceleration:

```
[PathTracer] Input scene file: .../dae/meshedit/cow.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0006 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0000 sec)
[PathTracer] Rendering... 100%! (21.9912s)
[PathTracer] BVH traced 477573 rays.
[PathTracer] Average speed 0.0217 million rays per second.
[PathTracer] Averaged 885.142177 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

```
[PathTracer] Input scene
[PathTracer] Rendering usi
[PathTracer] Collecting pr
[PathTracer] Building BVH
[PathTracer] Rendering...
[PathTracer] BVH traced 29
[PathTracer] Average speed
[PathTracer] Averaged 8.66
[PathTracer] Saving to fil
[PathTracer] Job completed
```

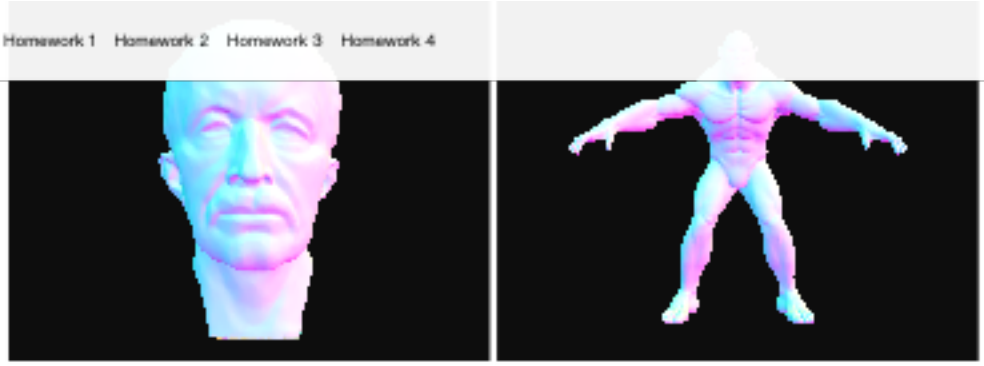
Notice how adding in the BVH reduced the render time from 21.9 seconds to 0.156 seconds — a dramatic speedup. (and really satisfying to run) Below is another example, using the teapot:

```
[PathTracer] Input scene file: .../dae/meshedit/cow.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0006 sec)
[PathTracer] Building BVH from 2464 primitives... Done! (0.0000 sec)
[PathTracer] Rendering... 100%! (21.9912s)
[PathTracer] BVH traced 477573 rays.
[PathTracer] Average speed 0.0217 million rays per second.
[PathTracer] Averaged 7.048304 intersection tests per ray.
[PathTracer] Saving to file: teapot.png... Done!
[PathTracer] Job completed.
```

```
[PathTracer] Input scene
[PathTracer] Rendering usi
[PathTracer] Collecting pr
[PathTracer] Building BVH
[PathTracer] Rendering...
[PathTracer] BVH traced 41
[PathTracer] Average speed
[PathTracer] Averaged 738.
[PathTracer] Saving to fil
[PathTracer] Job completed
```

In general, we found that the speedup is several orders of magnitude, likely due to the asymptotic speedup from $O(n)$ to $O(\log n)$ per ray. Additionally, this can be directly visualized by the average intersections per ray dropping to approximately 1% of the intersections computed per ray before implementing the BVH. Overall, we are very happy with the speedup presented by our implementation. More experimentation could be done on the various heuristics of generating BVHs to create a faster speedup.

Showcasing some speedily rendered complex models:



Relatively complicated .dae files generated with our BVH acceleration.