Project 3-1: PathTracer-1

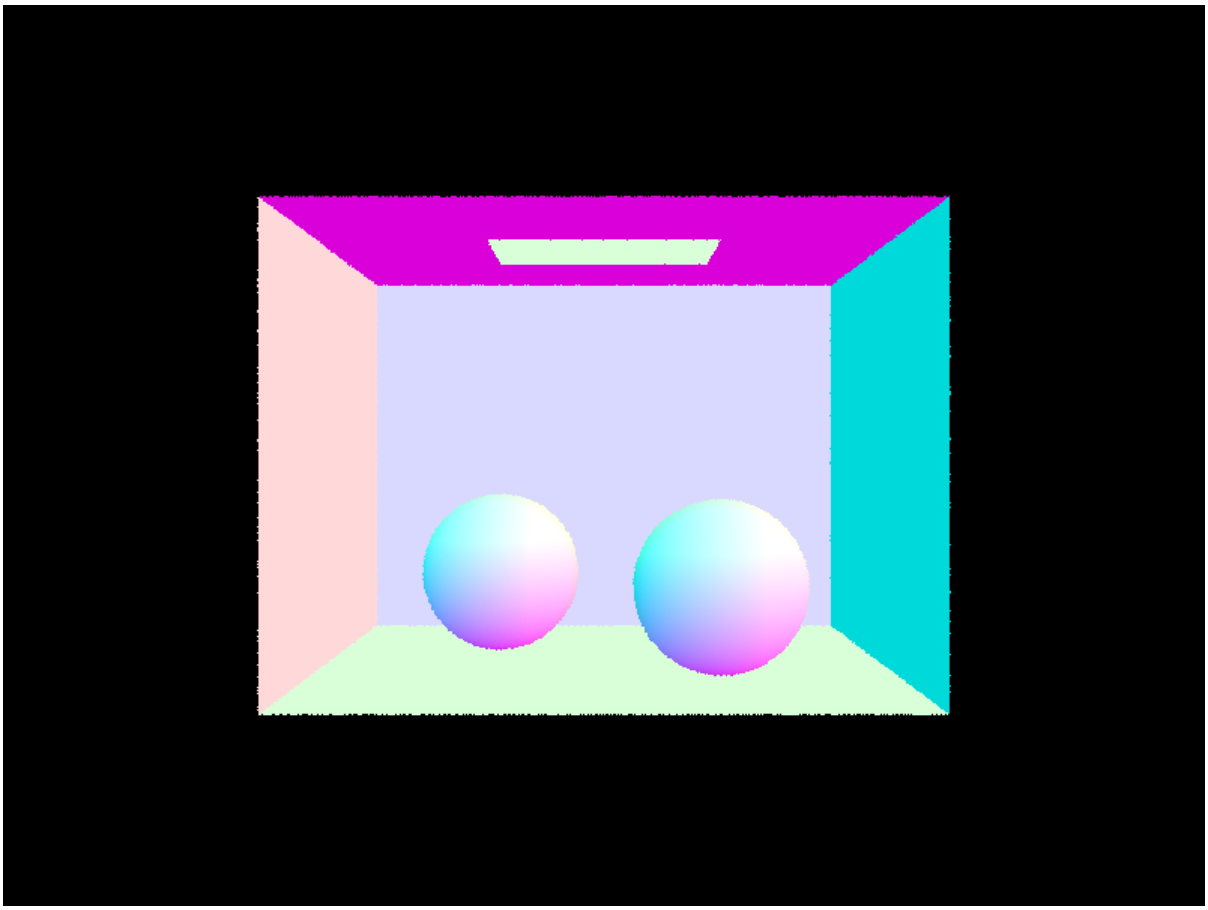
Part 1. Ray Generation and Scene Intersection

1. Walk through the ray generation and primitive intersection parts of the rendering pipeline.
    - Firstly, I generated camera rays. `Camera::generate_ray` is called with parameters x and y, which are coordinates in normalized image space. Within this function, I used the formula in spec and convert the inputs to a point in the camera space, using linear interpolation. The product of the c2w matrix and this point defines the direction of the resulting ray in the world space.
    - Secondly, I generated pixel samples. For any given pixel located (x, y) in the unnormalized image space, I generated ns_aa rays that intersect with the pixel at randomly different positions. I averaged the illumination values returned by these rays to derive the final illumination value for each pixel.
    - Thirdly, I implemented Ray-Triangle Intersection function. I implemented the Möller-Trumbore Algorithm to find b1, b2, t values for intersection detection. If t value is with in the valid range of the ray ([Ray::t_min, Ray::t_max]), I can determine that an intersection has occurred. In such a case I set the Ray::t_max to t so that the ray is "blocked" by the triangle.
    - Finally, I implemented Ray-Sphere Intersection function. I derived b1, b2, t values by solving a quadratic equation. Vieta's formulas give me two t values, which corresponds to the entering point and the leaving point of the sphere. However, I only care about the entering point of intersection since no more ray is needed beyond that. Setting the Ray::t_max to the smaller t value so that the ray is "blocked" by the sphere.

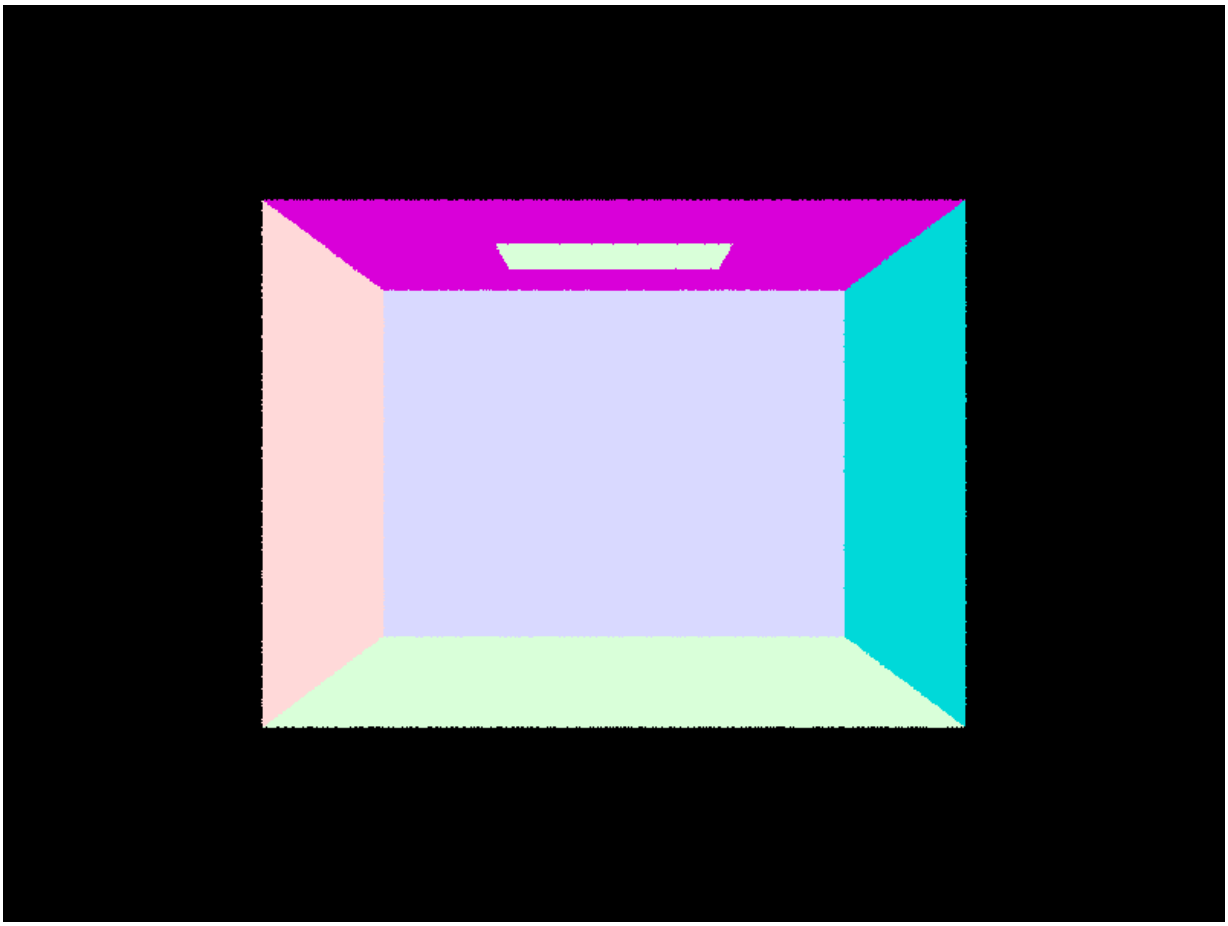
Explain the triangle intersection algorithm you implemented in your own words.

- As I described above, I implemented the Möller-Trumbore Algorithm in part 1 task 3. According to Slide 22 in Lecture 9, I computed s1, s2, s, d, e1, e2 with the formula on the right and passed them to the optimized function. I got the barycentric coordinates b1, b2 and the intersecting time t for the output. If the intersection occurs on the valid part of the ray and within the bound of triangle (t > 0 && r.min_t <= t && t <= r.max_t, 0 <= b1 && b1 <= 1 && 0 <= b2 && b2 <= 1 && b1 + b2 <= 1, respectively), then I can conclude that there is really an intersection on the triangle.

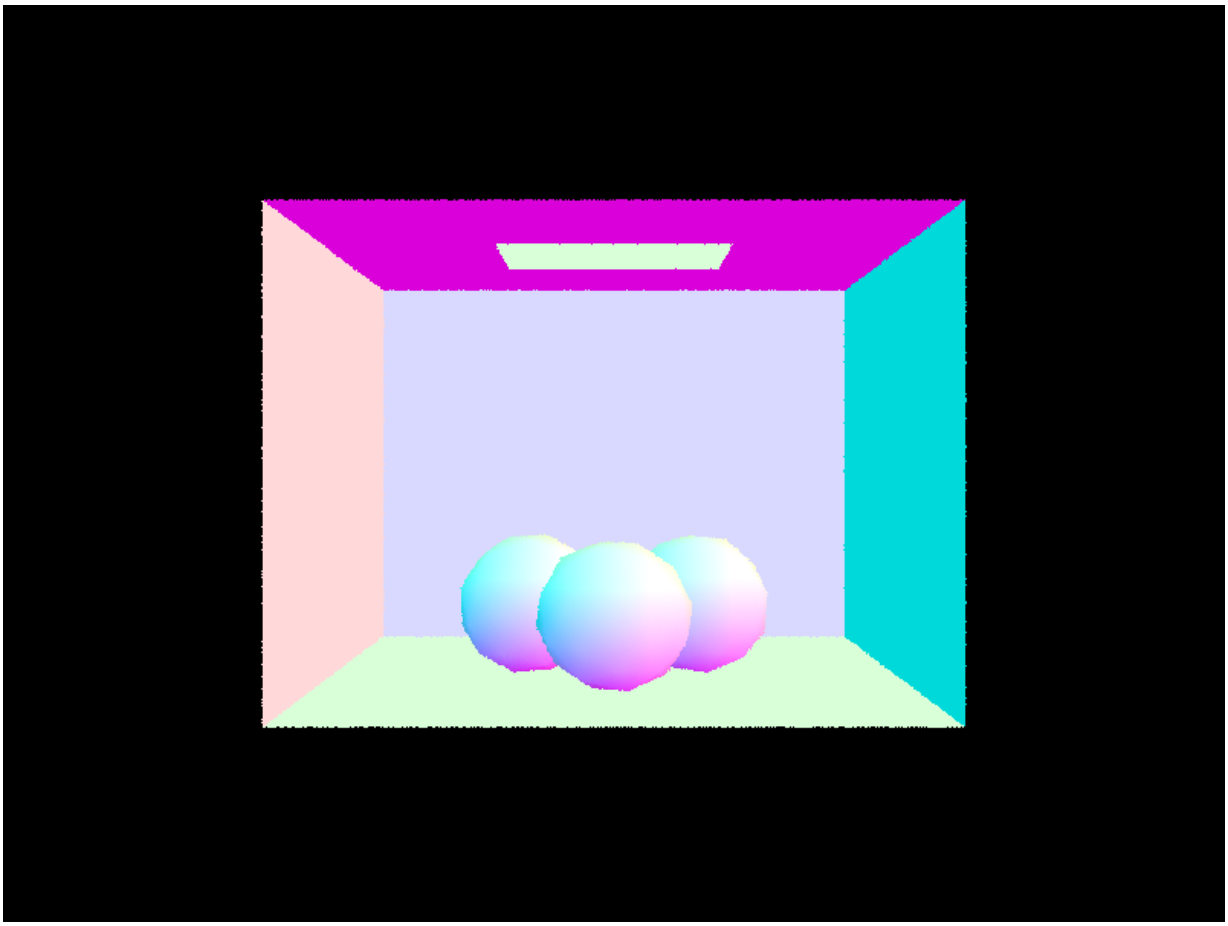
Show images with normal shading for a few small .dae files.



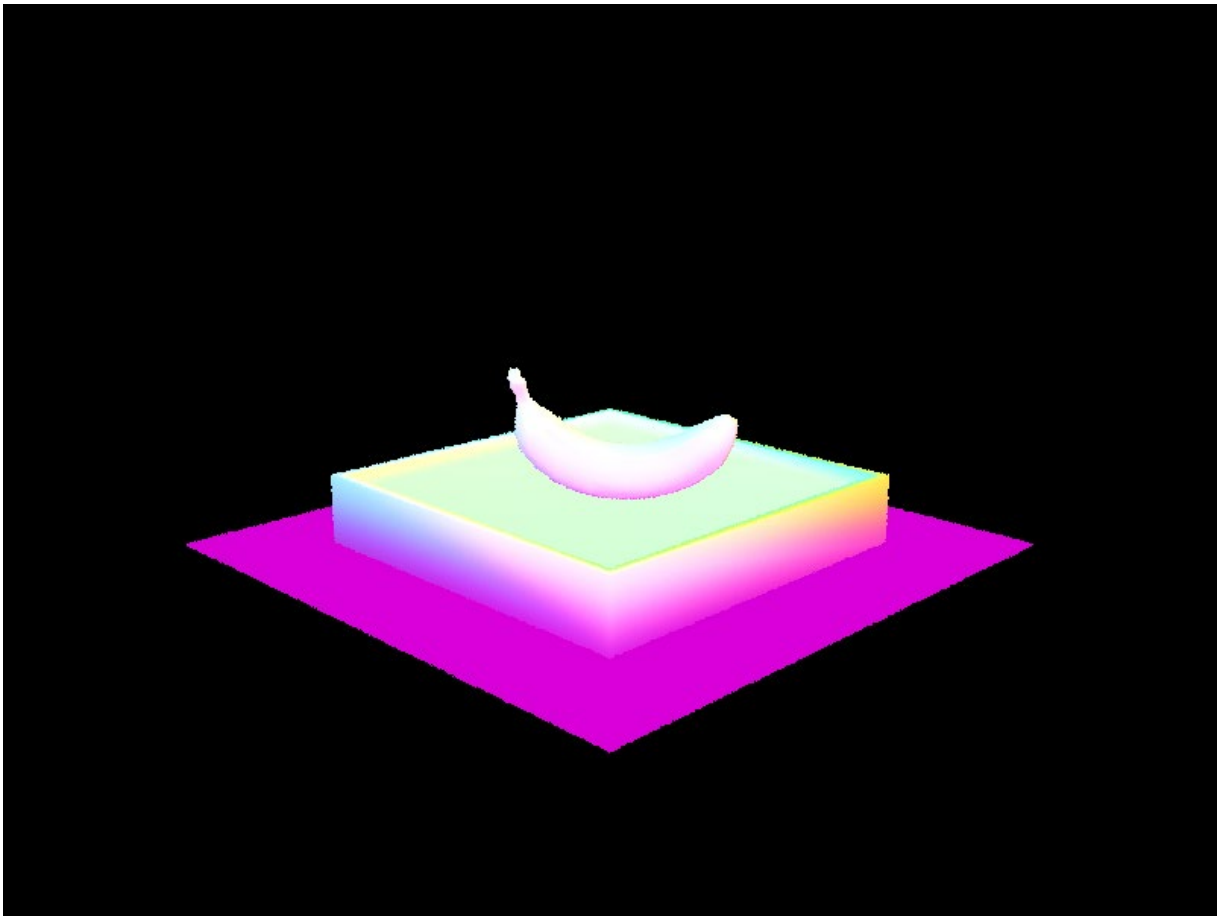
./pathtracer -r 800 600 -f CBspheres.png ../dae/sky/CBspheres_lambertian.dae



./pathtracer -r 800 600 -f CBempty.png ../dae/sky/CBempty.dae



./pathtracer -r 800 600 -f CBgems.png ../dae/sky/CBgems.dae



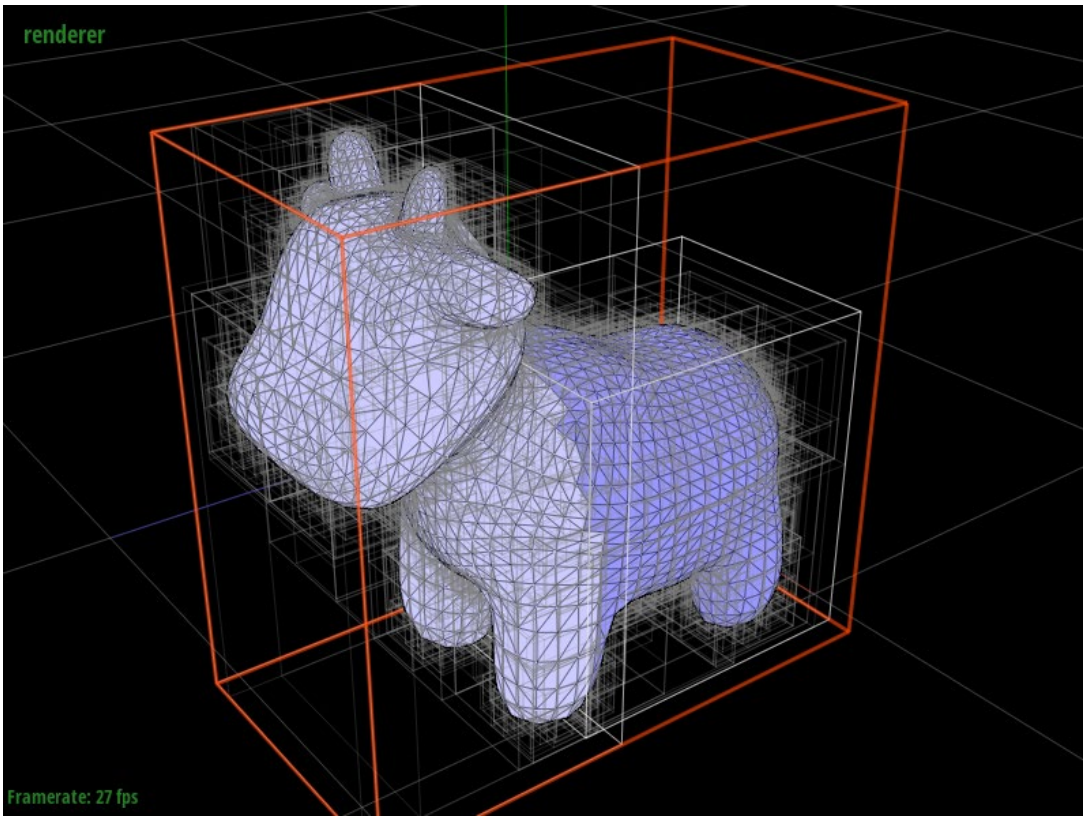
./pathtracer -r 800 600 -f banana.png ../dae/keenan/banana.dae

Part 2. Bounding Volume Hierarchy

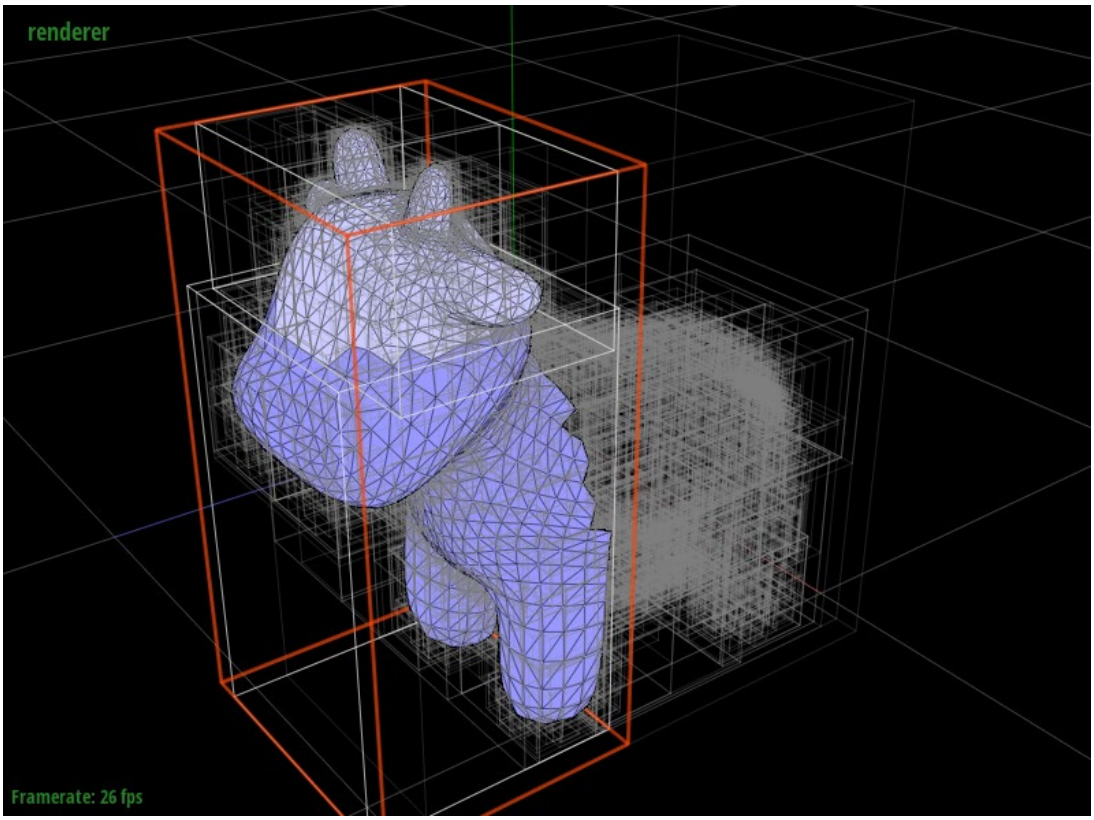
Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.

- My BVH construction algorithm is simple. First it ensures the number of primitives is large enough (larger than max_leaf_size) for splitting into two child BVHNodes. Then it calculates the centroid of every primitive. One of the three axis which spans the greatest range is selected, and the algorithm will soon divide the primitives into two groups along this axis. The reason I chose the greatest spanned axis is such group of primitives will become more cube-like instead of some skinny cuboid. To make sure the vector of primitives is correctly sorted and fall towards the same side of the dividing boundary, I sorted the list (vector, actually) of primitives using build-in std::sort() according to the selected axis coordinate of the primitives' centroids. Finally, the primitives are divided into two equal sized groups as described above, and they will be divided recursively until small enough to fit in leaves. Thus, I used the median heuristics instead of the mean recommended in the spec, so every leaf node contains approximately same amount of primitives under my algorithm.

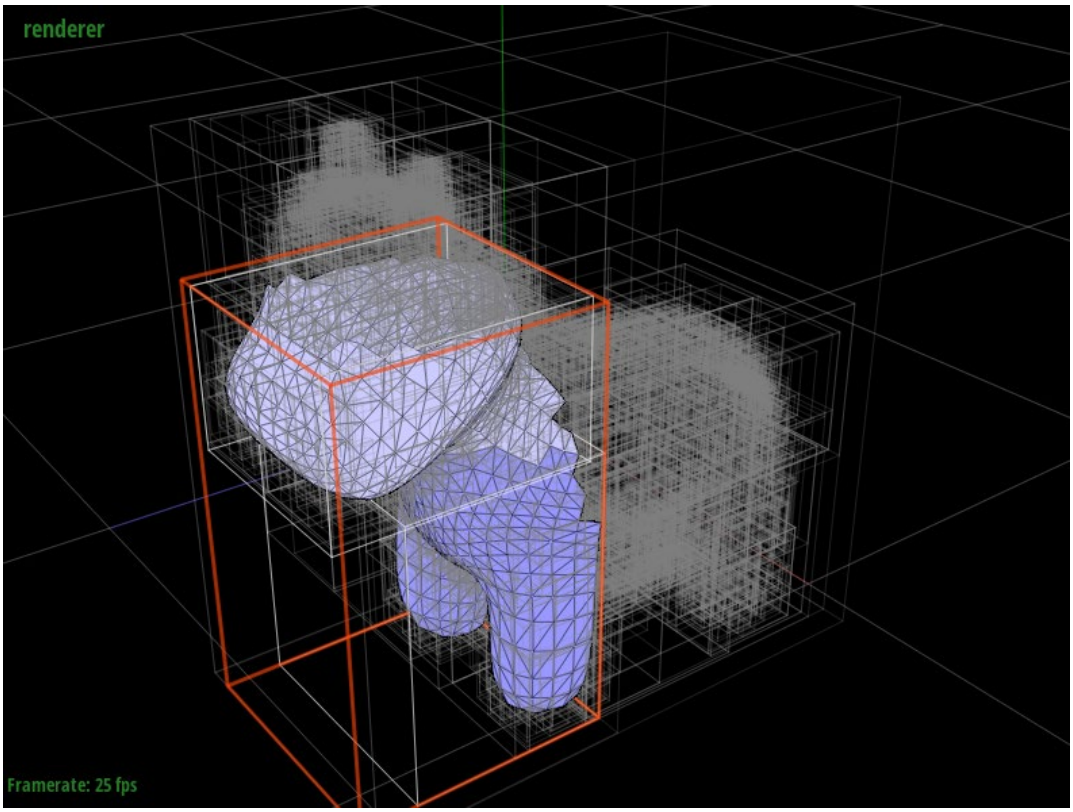
- Edge case: sometimes the algorithm encounters a set of primitives with exactly the same centroid coordinate, spanning on the selected axis. For example, the max_leaf_size is 2 and centroids of primitives are (1,2,3), (1,2,3), and (1,2,3). In such situation I decided to send each primitive into the currently lesser group, so they will not end up in the same group every time, in which an infinite loop occurs.
- Some previews after I finished 2-1:



The root node of BVH



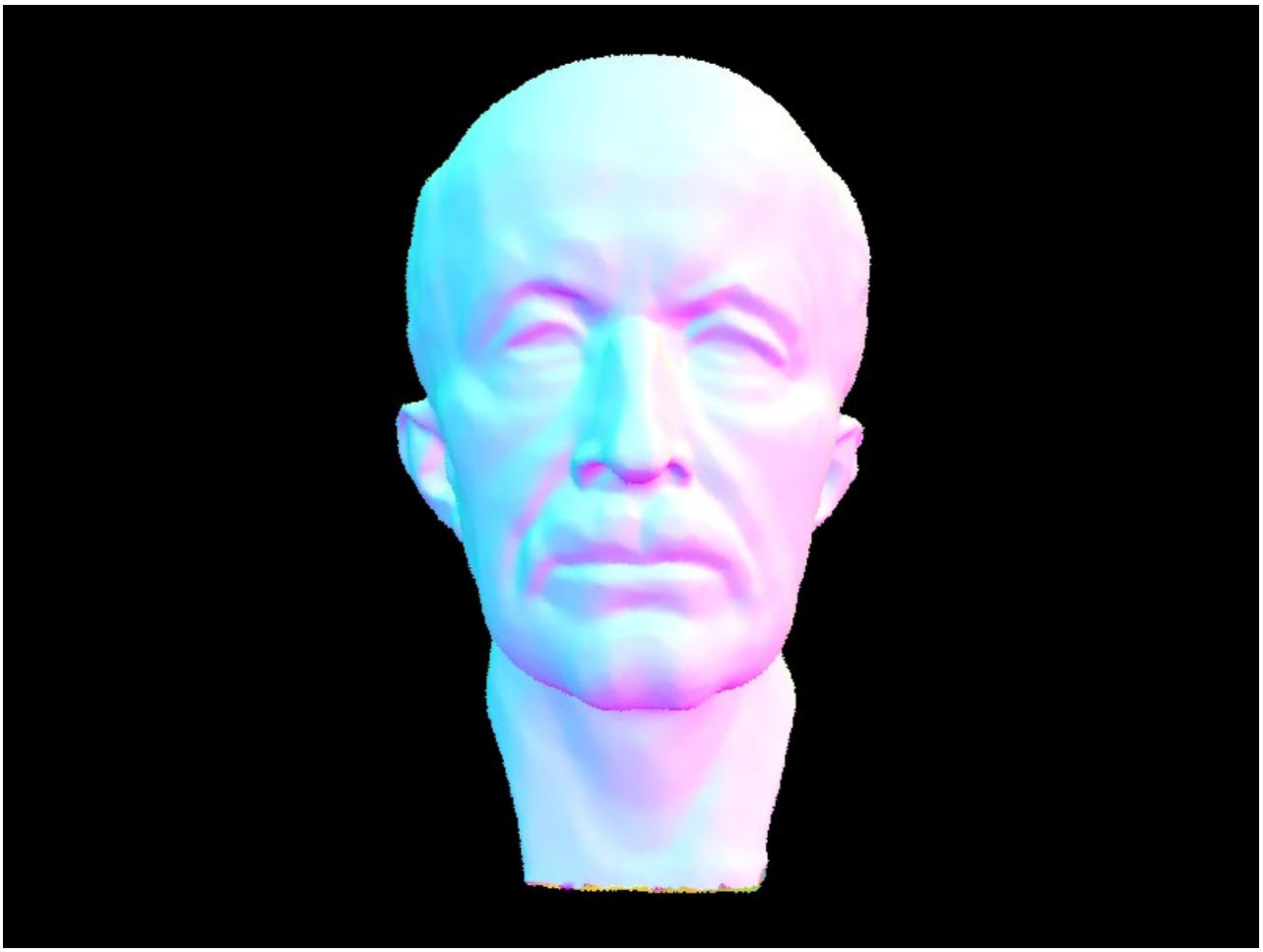
`root->left` node of BVH



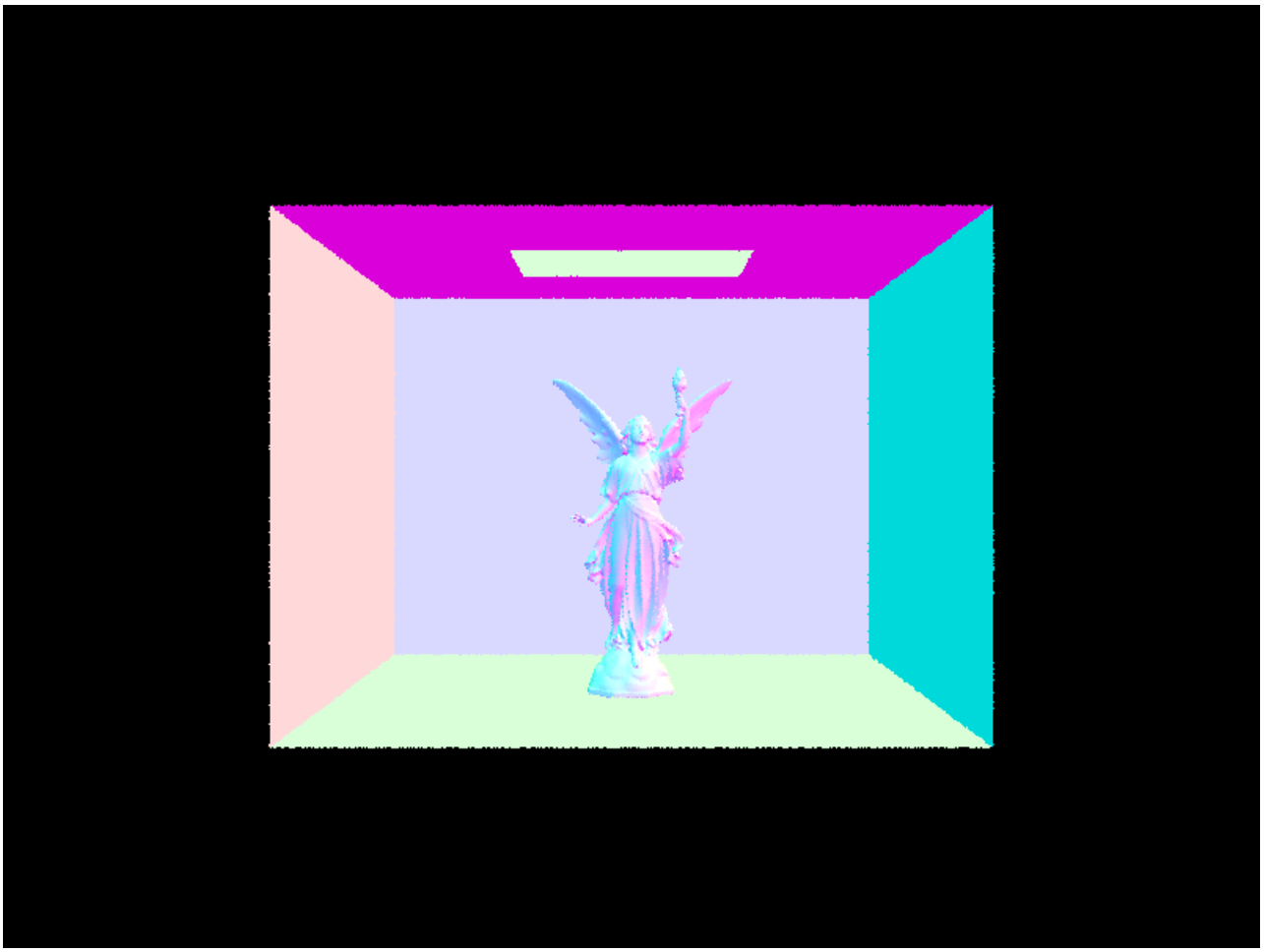
`root->left->left` node of BVH

Fully rendered preview

Show images with normal shading for a few large .dae files that you can only render with BVH acceleration.



./pathtracer -t 8 -r 800 600 -f maxplanck.png ../dae/meshedit/maxplanck.dae

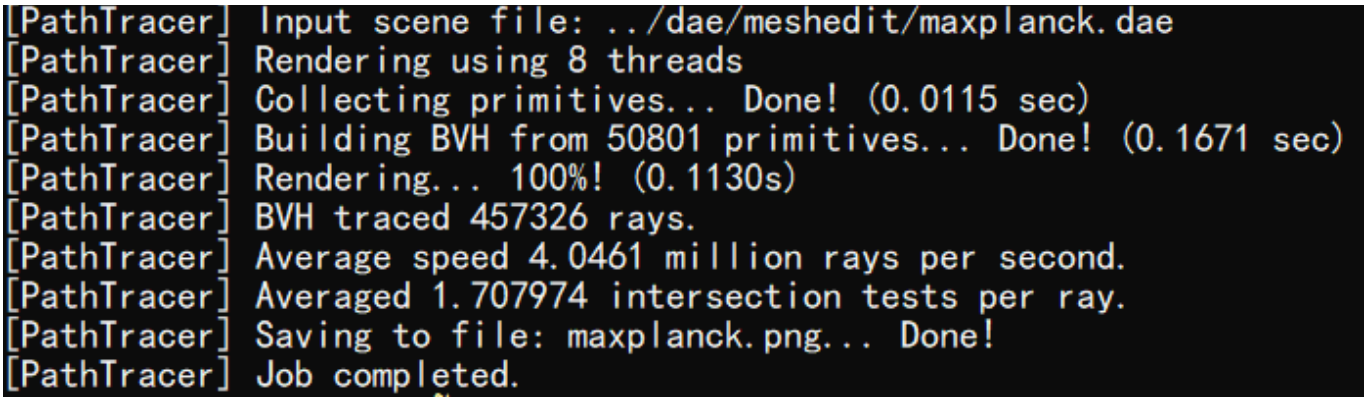


./pathtracer -t 8 -r 800 600 -f CBlucy.png ../dae/sky/CBlucy.dae

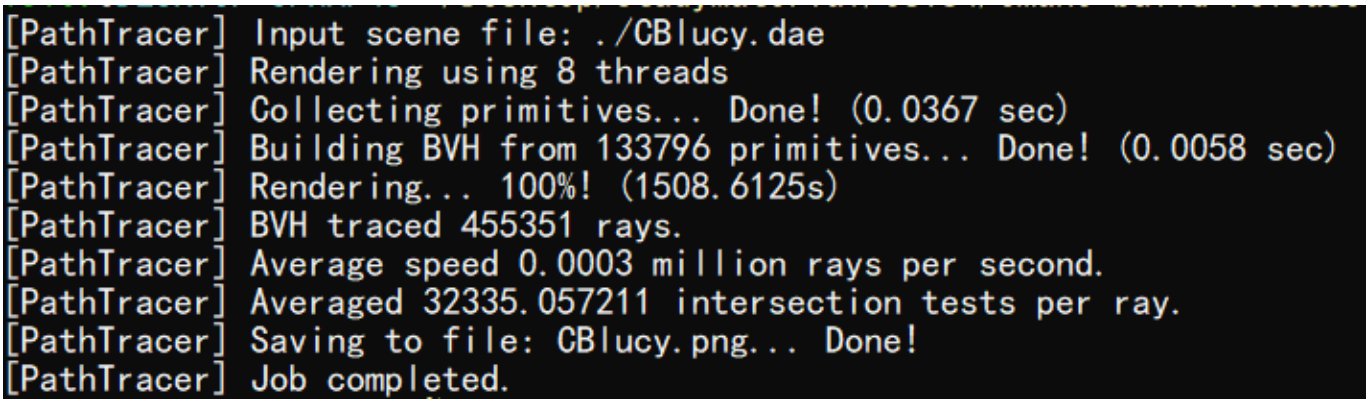
Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.

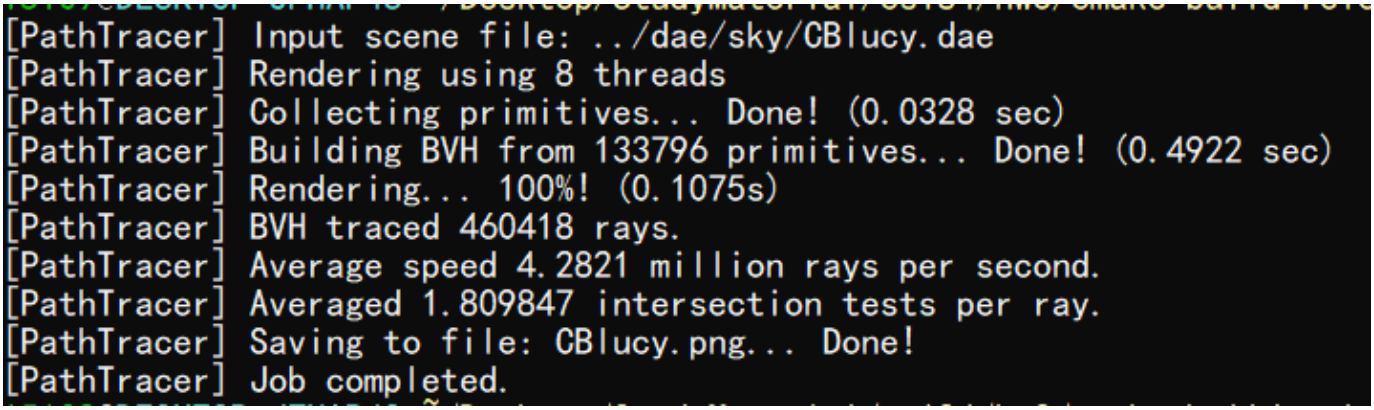
maxplanck.dae: without BVH acceleration: ~450 seconds -> with BVH acceleration: ~0.28 seconds (1600x speedup)

```
[PathTracer] Input scene file: ../dae/meshedit/maxplanck.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0181 sec)
[PathTracer] Building BVH from 50801 primitives... Done! (0.0019 sec)
[PathTracer] Rendering... 100%! (449.1164s)
[PathTracer] BVH traced 414593 rays.
[PathTracer] Average speed 0.0009 million rays per second.
[PathTracer] Averaged 10207.246673 intersection tests per ray.
[PathTracer] Saving to file: maxplanck.png... Done!
[PathTracer] Job completed.
```

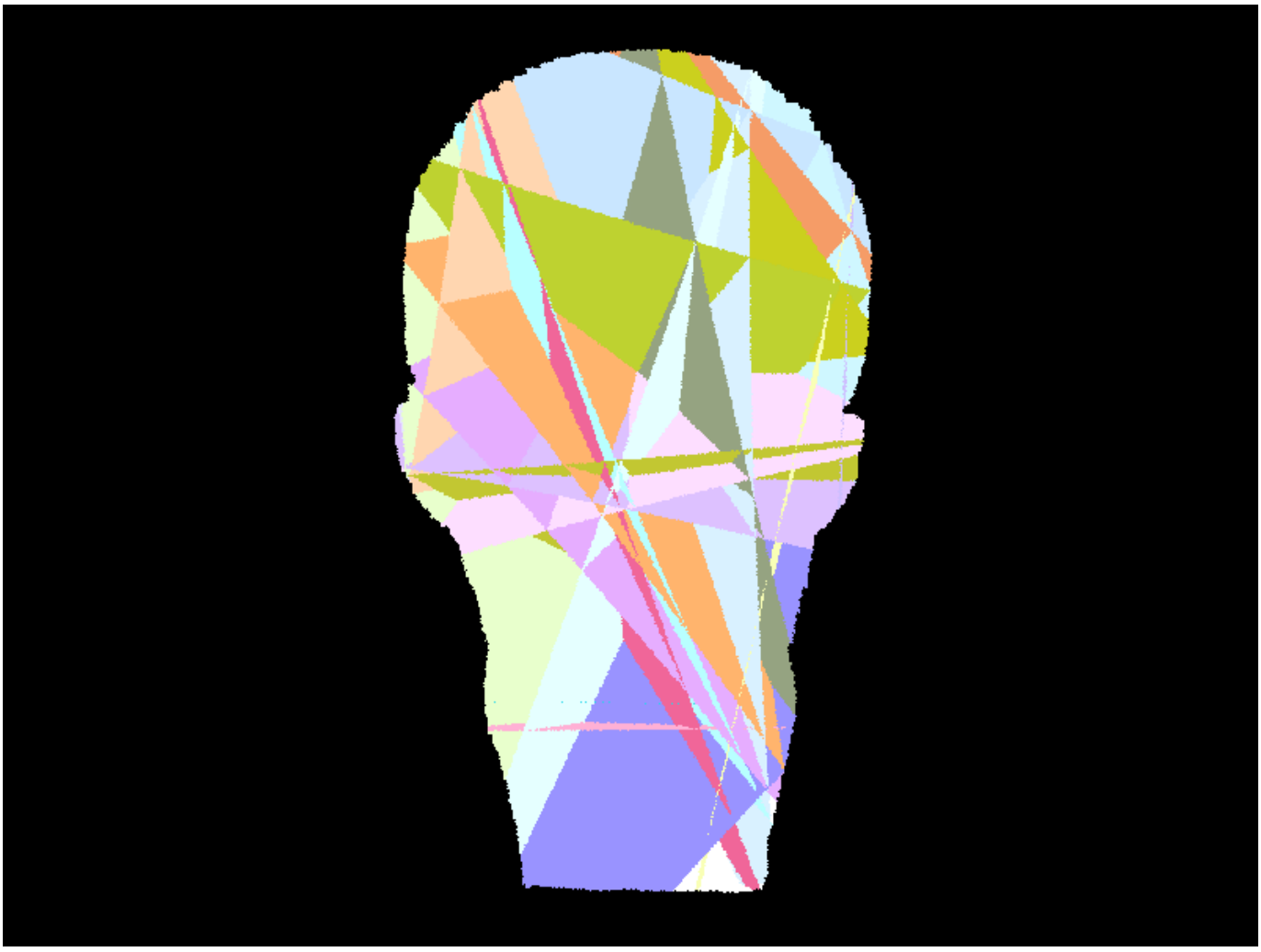


CBlucy.dae: without BVH acceleration: ~1500 seconds -> with BVH acceleration: ~0.6 seconds (2500x speedup)





Side note - fascinating buggy rendering:



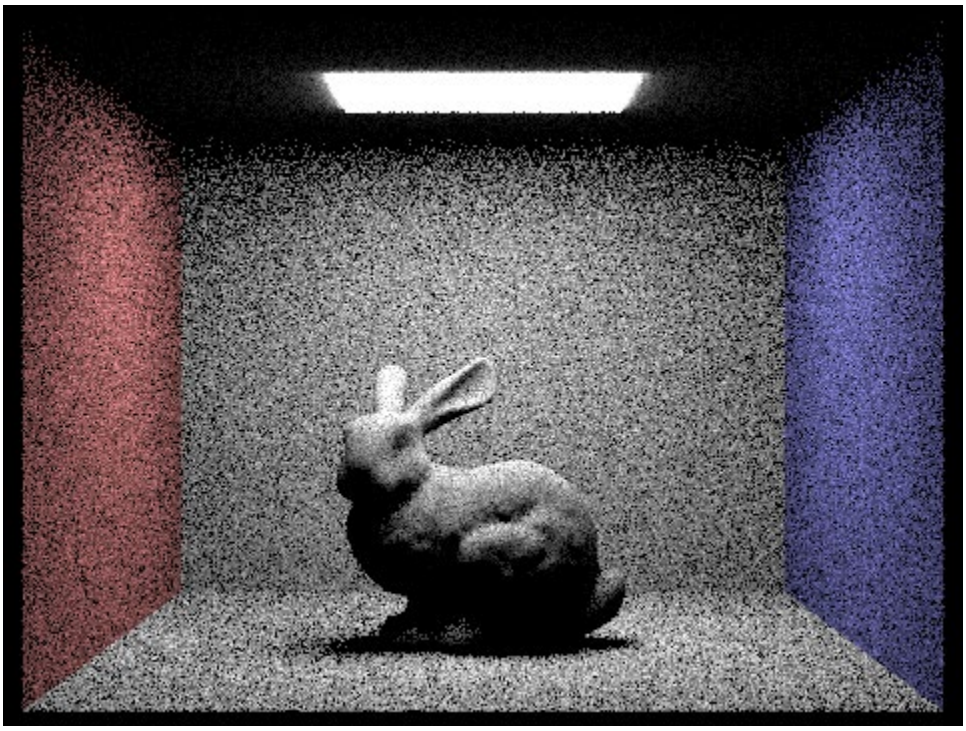
Task 3 Direct Illumination

Walk through both implementations of the direct lighting function.

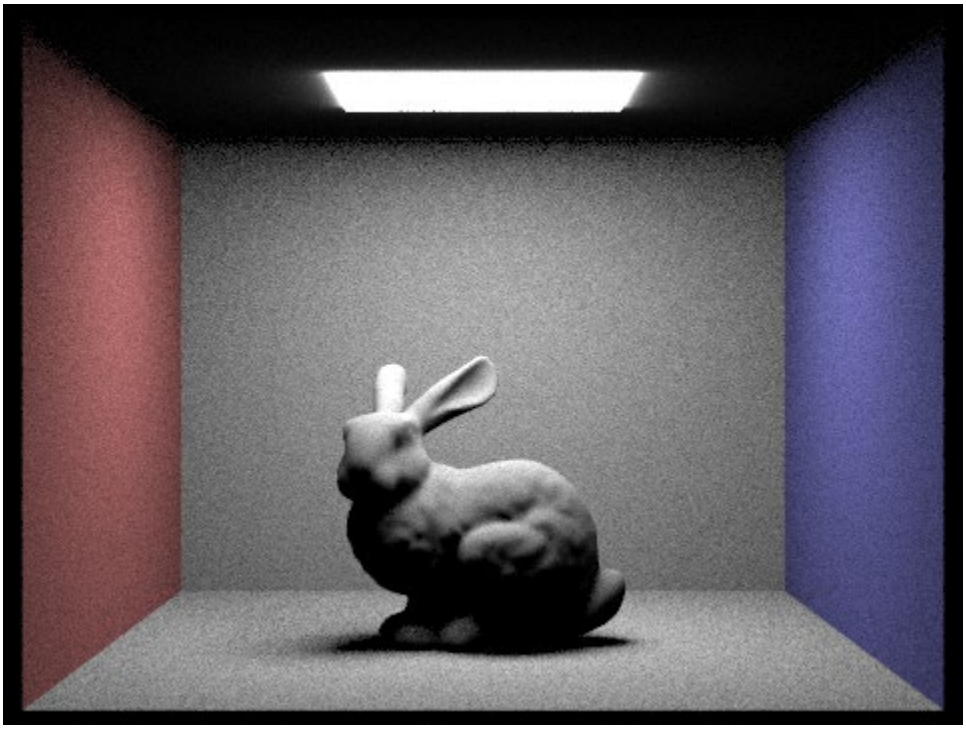
- Uniform hemisphere sampling takes `num_samples` samples of rays with random directions that pierce the hemisphere. The purpose is to determine the pixel's brightness. After sampling a ray, I reversely traced it to make sure it "goes into" a light source. `zero_bounce_radiance()` only takes rays directly from the light source. On the other hand, `one_bounce_radiance()` considers rays that only intersect exactly once and ignore rays that intersect more than once (which is implemented in indirect illumination section). I simply added them up to render the directly illuminated scene. When an intersection happens, I calculate the amount of light that emits from the intersection point towards the camera. By averaging all samples, I derived the brightness of a pixel.
- Importance sampling (light sampling) uses a shadow ray to determine the brightness of a given pixel instead of a light ray I just used in Uniform hemisphere sampling. Before sampling, I checked if the light source is a point or an area. The point light sources should only be sampled once, while the area light sources are sampled `ns_area_light`

times. After sampling a shadow ray, I traced this ray to see if it intersects with the scene. If so, the shadow ray is "blocked" and should be omitted. Otherwise, I count this ray as valid and add it to `L_curr_out`. After sampling enough rays, I averaged the value of `L_curr_out` and add it to `L_out`. I traversed all light sources to derive the final value of `L_out`, the brightness of a pixel with every light source's contribution. (Note: My importance sampling experienced higher noise running `./pathtracer -t 8 -s 1 -l 1 -m 1 -f bunny_1_1.png -r 480 360 ../dae/sky/CBbunny.dae` at the very beginning. Then I looked up the spec and piazza and subtracted a small value `EPS_F` from `distToLight`, and the noise was gone. The spec says the `EPS_F` acts like as small deviation that prevents shadow rays from intersecting with light itself, but I'm not sure why it makes the rendering noisy)

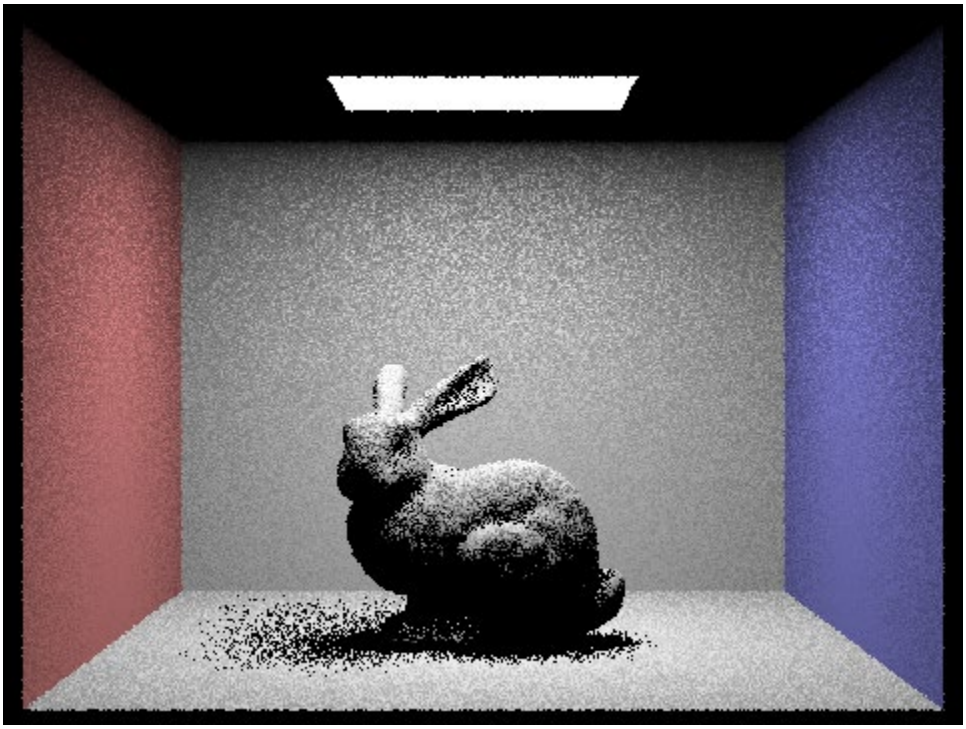
Show some images rendered with both implementations of the direct lighting function.



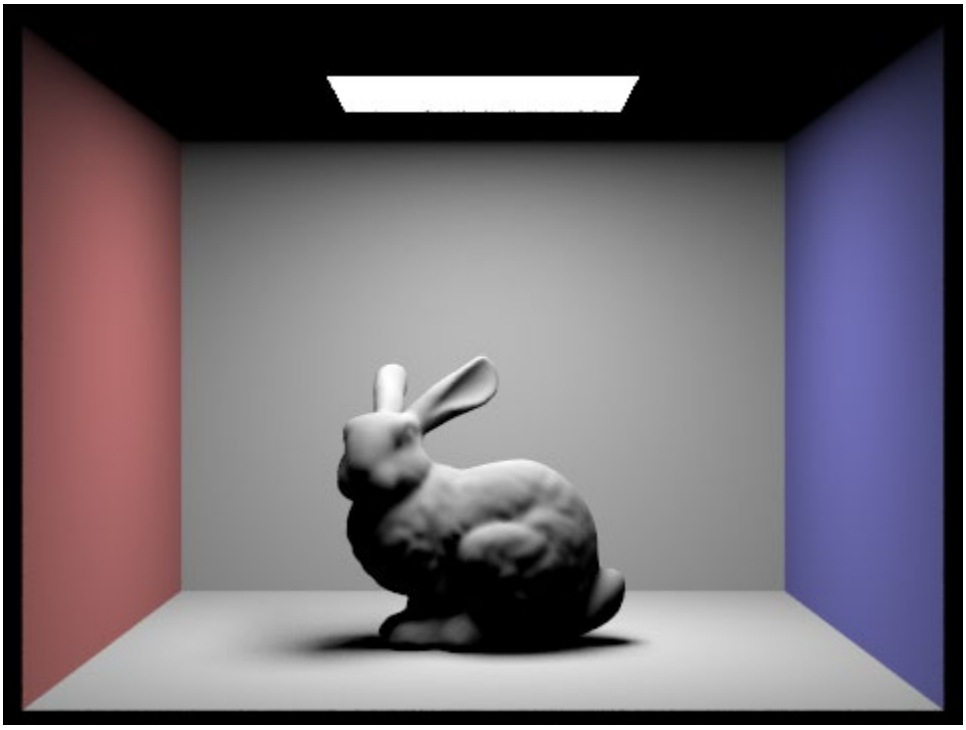
./pathtracer -t 8 -s 16 -l 8 -H -f CBbunny_H_16_8.png -r 480 360 ../dae/sky/CBbunny.dae



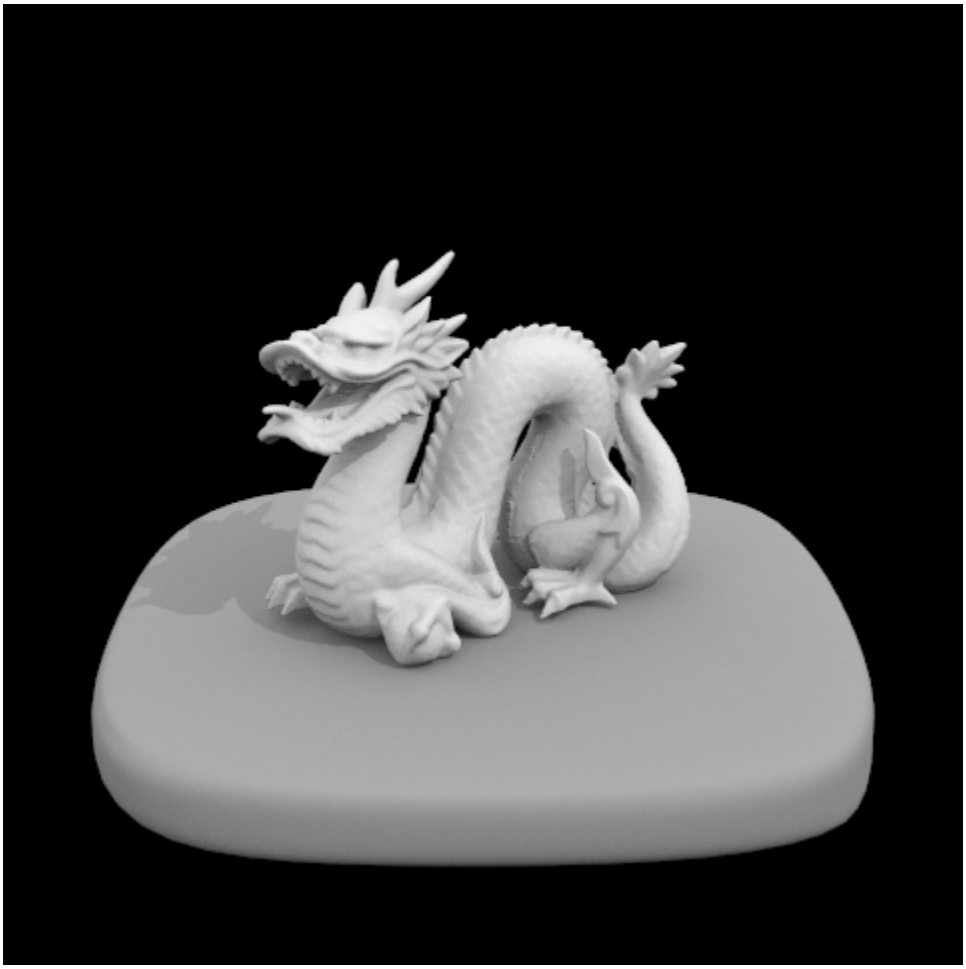
./pathtracer -t 8 -s 64 -l 32 -m 6 -H -f CBbunny_H_64_32.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 1 -m 1 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae

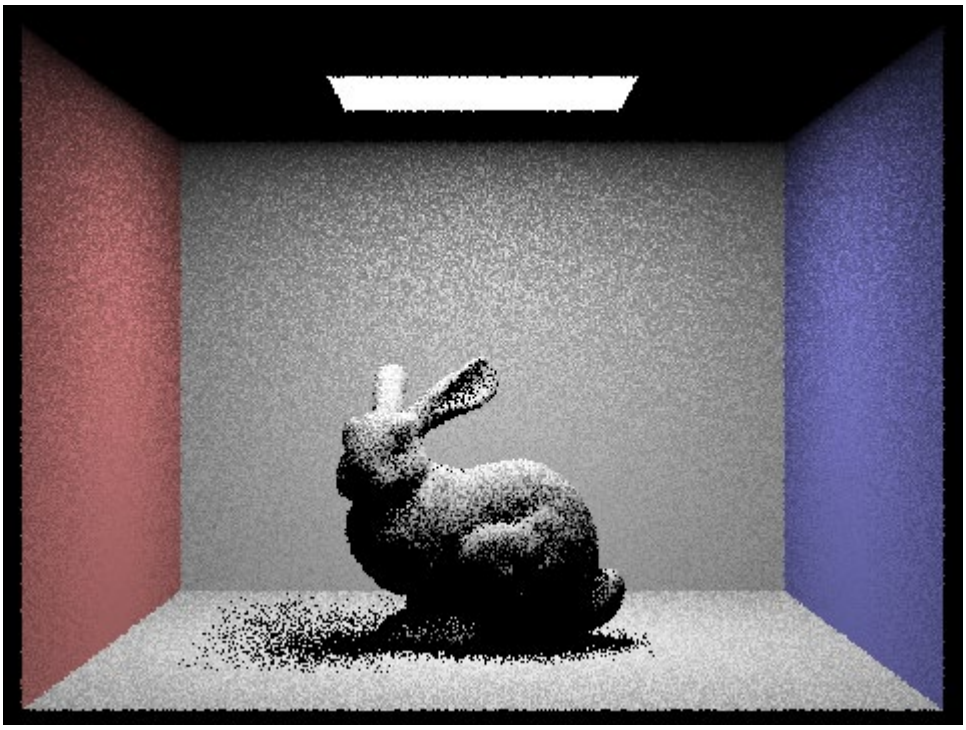


./pathtracer -t 8 -s 64 -l 32 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae

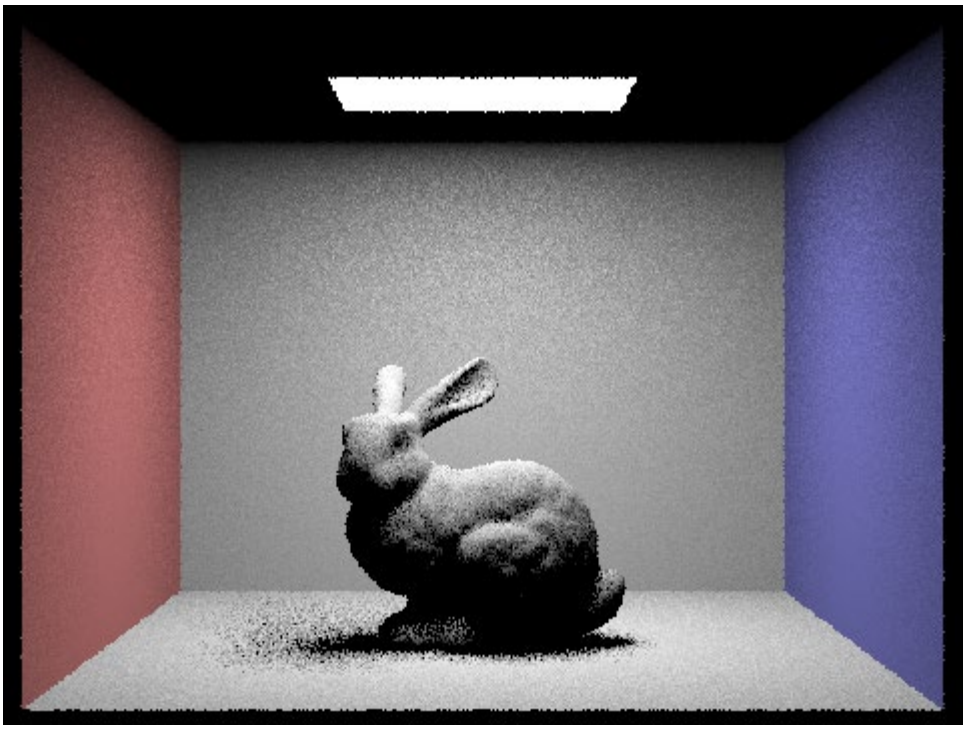


./pathtracer -t 8 -s 64 -l 32 -m 6 -f dragon_64_32.png -r 480 480 ../dae/sky/dragon.dae

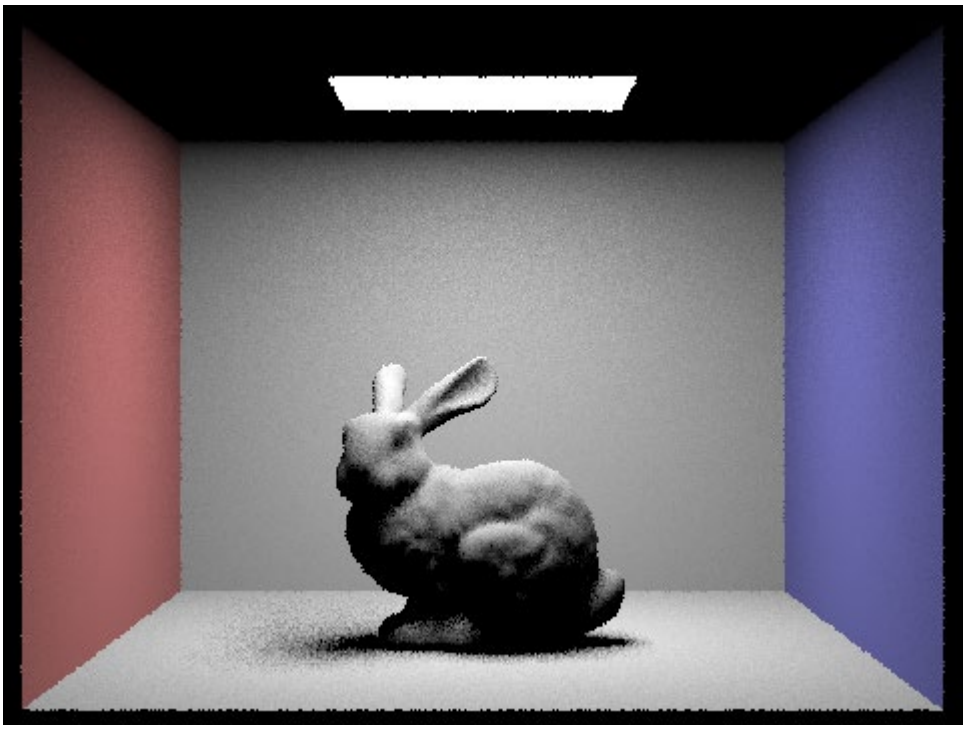
Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the -l flag) and with 1 sample per pixel (the -s flag) using light sampling, not uniform hemisphere sampling.



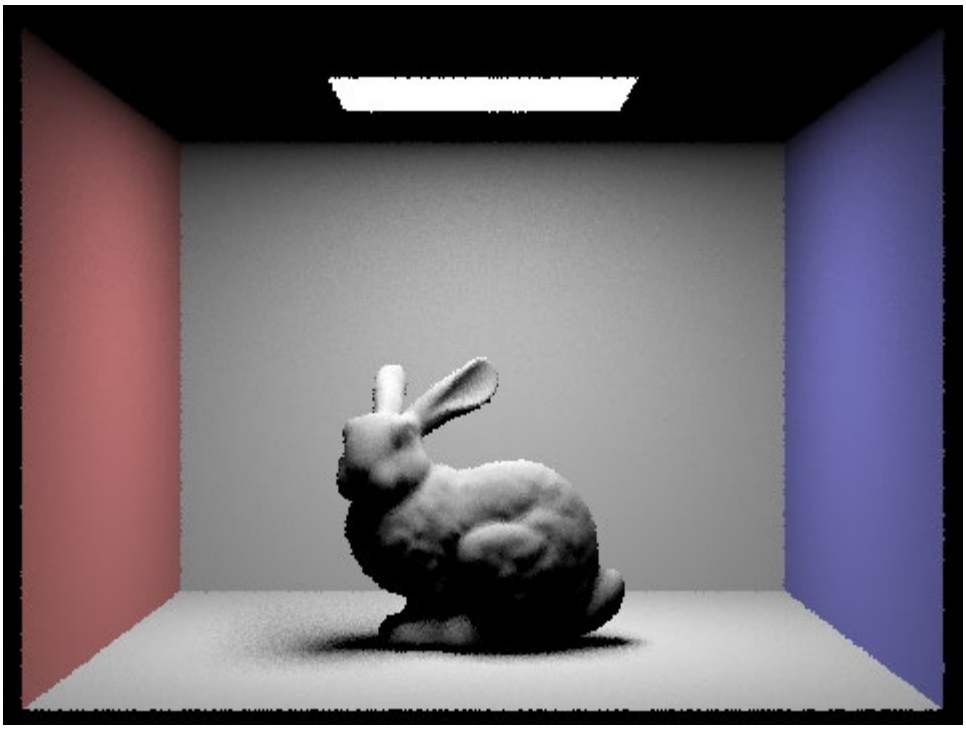
./pathtracer -t 8 -s 1 -l 1 -m 6 -f bunny_64_32_test.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 4 -m 6 -f bunny_64_32_test.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 16 -m 6 -f bunny_64_32_test.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 64 -m 6 -f bunny_64_32_test.png -r 480 360 ../dae/sky/CBbunny.dae

- The noise level decreases as the number of light rays increases.

Compare the results between uniform hemisphere sampling and lighting sampling in a one-paragraph analysis

The result of uniform hemisphere sampling shows soundly more noise than lighting sampling with the same number of rays and sampling frequency. Besides, the uniform hemisphere sampling (152s) spent more time rendering the same scene than lighting sampling (117s).