CS 184: Computer Graphics and Imaging, Spring 2023

Project 3-1: Path Tracer

YOUR NAME(S)

Website URL: Click Here.

Overview

YOUR RESPONSE GOES HERE

Part 1: Ray Generation and Scene Intersection (20 Points)

Walk through the ray generation and primitive intersection parts of the rendering pipeline.

To generate a ray, a camera coordinate and normalized direction vector are necessary. Three mappings are required for the sampling of every pixel via raycasting, and they are as follows:

1. Pixel coordinate frame to normalized coordinate frame via axis scaling.
2. Normalized coordinate frane to image frame via scaling and translation perpendicular to camera z-axis.
3. Camera-to-world transformation on direction vector using the c2w constant. (The origin of the ray is also translated to match the camera origin.)

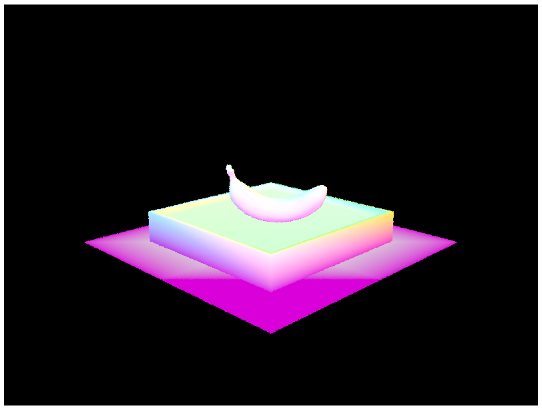
The goal of the first two transformations is to map the pixel plane onto the image plane to draw a ray from the camera origin through the field of view for each pixel. Multiple sampling for a single pixel can be done by random sampling. Once the ray direction is determined, the third transformation allows for the proper ray propagation.

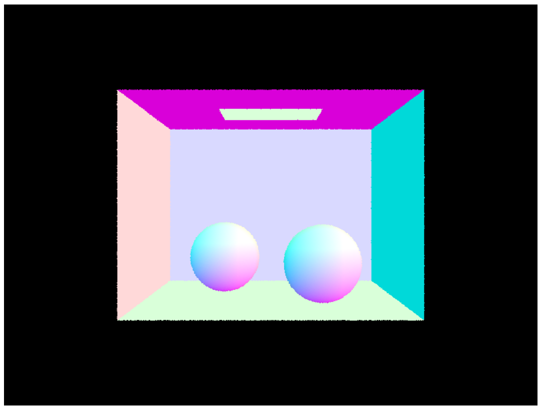
Explain the triangle intersection algorithm you implemented in your own words.

Ray-triangle intersection was implemented using the Moller-Trumbore algorithm. Basically, this algorithm effectively combines barycentric coordinates and the implicit definition of a plane to determine the intersection point's residence in the triangle by making sure that the coordinates are within [0, 1] and verifying that the intersection t-value is within the bounds of the scene.

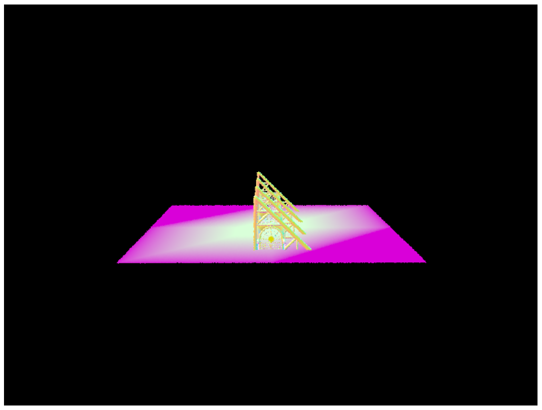
The sphere check was implemented according to the slide provided in the spec. Using the quadratic formula to solve an equation give the intersection t-values. Ensuring the discriminant was positive allowed for filtering imaginary results. Checking if either of the roots fell within the bounds and selecting the minimum root results in the intersection point.

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


banana.dae


CBspheres.dae

cow.dae


building.dae

Part 2: Bounding Volume Hierarchy (20 Points)

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

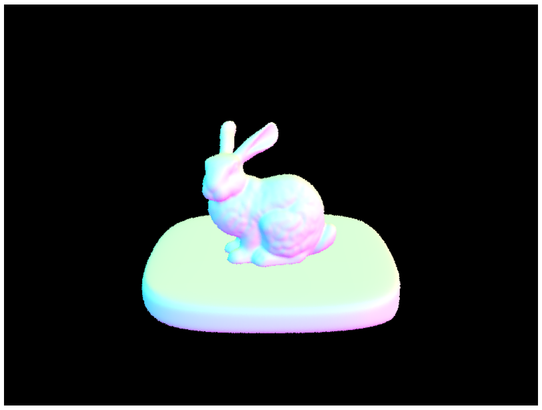
I did a bit of research on surface area heuristic for axis splitting and chose to implement this. The general implementation is as follows:

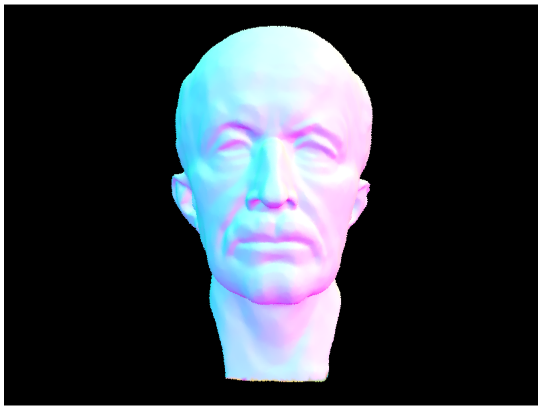
For any node, compute the mean 3D position of the centroids of each primitive within the node's bounding box. This results in three possible points along which to split the selected axis. Then, compute the cost of splitting each axis using a simple cost function of surface area of bounding box multiplied by the primitive count in each child node. Next, select the lowest cost axis and split at the previously found candidate point. This method of axis splitting reduces the chances that a ray will hit a bounding box without hitting a primitive.

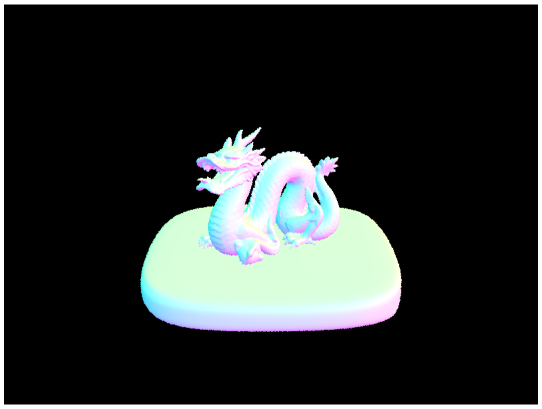
The reason for my selecting this heuristic over others (specifically, selecting the mean over the median centroid) is effectively as such:

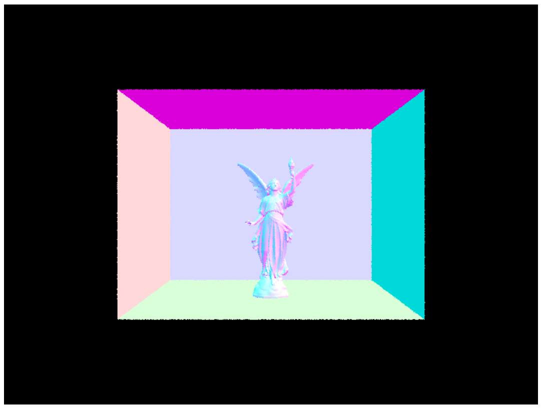
Firstly, this method doesn't require prior information about the distribution of the primitives. This method assumes a relatively even distribution of primitives over the scene. Under this assumption, the mean centroid and the median centroid are roughly equivalent, and thus over many renders the tree will remain mostly balanced.
Second, using the median over the mean would require sorting the primitives along each axis, which is cost-heavy, as a sort is required at each step in the recursion. For such a cost increase, the trade-off is a marginally more optimal tree.

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


bunny.dae


maxplanck.dae


dragon.dae


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.

The improvements the BVH tree brings are immense. The meshedit/cow.dae image rendered in 0.2158 seconds using the BVH system, a 230x improvement over the 49.7292 seconds that the non-BVH systems. The sky/dragon.dae image rendered in 0.1683 seconds using the BVH system, a 7191x improvement over the 1210.2393 seconds (over 20 minutes) that the non-BVH system took. It seems that for images with more polygons, the improvement was multitudes more tangible. This is because the computational complexity of the non-BVH system is $O(n)$, while the computational complexity of the BVH system is $O(\log n)$.

As an aside, the CBlucy.dae image actually rendered (in 0.1288 seconds) using BVH, but it took well over an hour to produce nothing without using BVH. Not sure whether this is a standalone case or something replicable, but it is worth noting.

Part 3: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

Direct lighting considers only zero- and one-bounce light rays (rays that do not bounce or bounce only once in the object space before arriving at the image plane). Uniform hemisphere sampling and importance sampling are the two methods implemented, differing primarily in the way that they sample the ray directions for estimating the illumination with a Monte Carlo estimator. For both, the inverse light path of a ray starting at the camera and going into the scene are considered; upon intersecting an object, the radiance that the camera would observe at this intersection point depends on the BRDF of the surface material and the irradiance at that point.

In this Monte Carlo estimation, several ray directions are sampled in the object space, and new rays are propagated in these directions until they reach a light source. If they are obstructed, they are treated as shadow rays and thus do not contribute to the summation.

In the case of uniform hemisphere sampling, the direction of the rays is not guaranteed to point towards the light source, which results in many of the rays having the chance of non-contribution to the summation. On the other hand, importance sampling are guaranteed to point in the direction of the light source and are only discounted if there is an obstruction to that ray, and thus the irradiance is better characterized compared to hemisphere sampling.

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

Uniform Hemisphere Sampling

example1.dae

example2.dae

Light Sampling

example1.dae

example2.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.

1 Light Ray (example1.dae)

4 Light Rays (example1.dae)

16 Light Rays (example1.dae)

64 Light Rays (example1.dae)

YOUR EXPLANATION GOES HERE

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

YOUR RESPONSE GOES HERE

Part 4: Global Illumination (20 Points)

Walk through your implementation of the indirect lighting function.

YOUR RESPONSE GOES HERE

Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.

example1.dae

example2.dae

Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel. (You will have to edit PathTracer::at_least_one_bounce_radiance(...) in your code to generate these views.)

Only direct illumination (example1.dae)

Only indirect illumination (example1.dae)

YOUR EXPLANATION GOES HERE

For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100 (the -m flag). Use 1024 samples per pixel.

max_ray_depth = 0 (CBbunny.dae)

max_ray_depth = 1 (CBbunny.dae)

max_ray_depth = 2 (CBbunny.dae)

max_ray_depth = 3 (CBbunny.dae)

max_ray_depth = 100 (CBbunny.dae)

YOUR EXPLANATION GOES HERE

Pick one scene and compare rendered views with various sample-per-pixel rates, including at least 1, 2, 4, 8, 16, 64, and 1024. Use 4 light rays.

1 sample per pixel (example1.dae)

2 samples per pixel (example1.dae)

4 samples per pixel (example1.dae)

8 samples per pixel (example1.dae)

16 samples per pixel (example1.dae)

64 samples per pixel (example1.dae)

1024 samples per pixel (example1.dae)

YOUR EXPLANATION GOES HERE

Part 5: Adaptive Sampling (20 Points)

Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

YOUR RESPONSE GOES HERE

Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of the image you are rendering, and your noise-free rendered result. Use 1 sample per light and at least 5 for max ray depth.

Rendered image (example1.dae)

Sample rate image (example1.dae)

Rendered image (example2.dae)

Sample rate image (example2.dae)