CS 184 Project 3-1 ~ Ray Tracing

About the project

In this project I constructed a rendering pipeline for a ray tracing program that can produce photo realistic images of three-dimensional environments. I did this by generating a virtual camera (of sorts) which shoots out rays into a virtual three-dimensional space does a series of bounces and calculates the amount of red, green and blue light that would hit the virtual sensor at the location that is equivelent to the pixel that is being rasterized.

Part 1 - Ray Generation and Scene Intersection

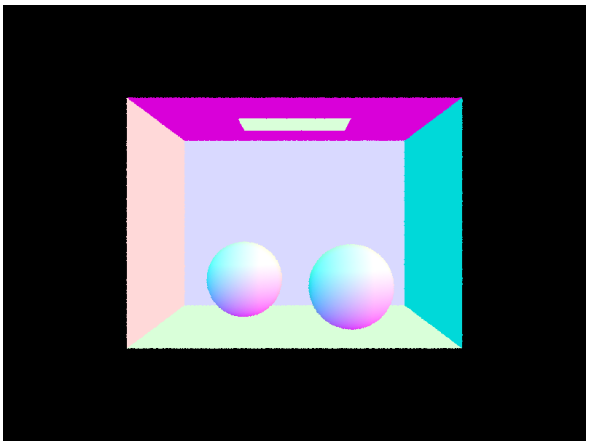
In this first part of the project, the first task that had to be completed is the process of initializing a ray to be used to aid rasterization. To do this the program takes the x and y co-ordinates which describe the location of the pixel that is being rasterized in the image space and calculate the ray that would pass through the camera origin and the location of the sensor in the camera space that corresponds with the pixel in the image space that is being rasterized. The length of the ray is restricted by its min_t and max_t variable attributes. This describes the range of values of t that the ray exists in.

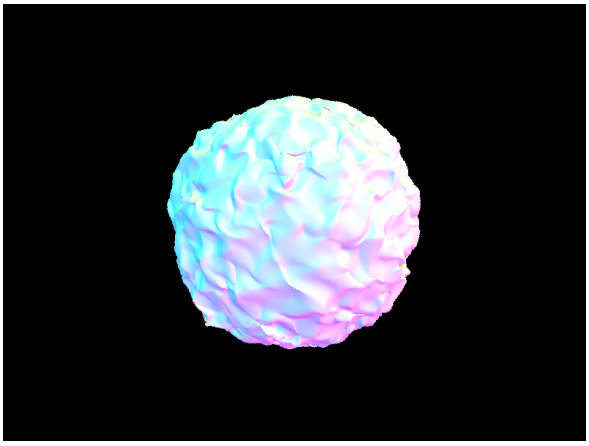
To render shapes in the 3D space I needed to implement some ray-object intersection functions for ray triangle intersection and ray-sphere intersection. These functions return true if the ray intersects the object (sphere or triangle) and if the location of this intersection is within the accepted t value range. It also returns the data involved with the intersection including the normal, bsdf, t value and the memory location of the object that is being intersected. If the intersection is valid, the ray's max_t variable will be set to the (smallest) t value of the intersection such that for any future test in the bvh intersection algorithm, if the object is further away from the camera origin, the function will not return true.

When implementing the ray-triangle intersection test, I used the Möller-Trumbore algorithm but removed the test that uses the determinant to return

CS 184 Project 3-1 ~ Ray Tracing

When implementing the ray-sphere intersection test, I found the locations of where the ray intersects the boundary of the sphere using the quadratic formula to find the values of t when the ray is exactly the distance of the radius away from the center of the triangle.





Part 2: Bounding Volume Hierarchy

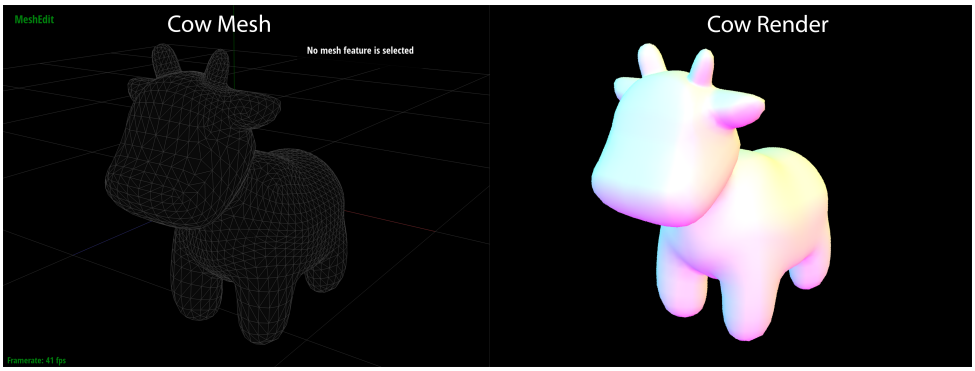
In part two I sped up the process of performing an intersection function on a three-dimensional space containing a large number of objects.

Before any intersection tests take place, the BVH tree has to be constructed from the list of primitives in the virtual space. To do this I recursively split the list of primitives using recursion.

function terminates and a tree with small enough leaf nodes has been created. If this is false then we have to do at least another split. I first determine which axis to split by figuring out which or the x, y or z axes of the binding box are larger (the largest is split). I then sorted the primitives using the sort function from the c++ library. The comparator input into the function would vary based on which axis was being split such that the primitives are sorted with respect to that axis. Then the list can be split into its first and second half and passed down the recursive function to take the value of the left and right children. This splits the bounding boxes based on the median which I believe to be faster than splitting by the mean because it will result in a more balanced tree.

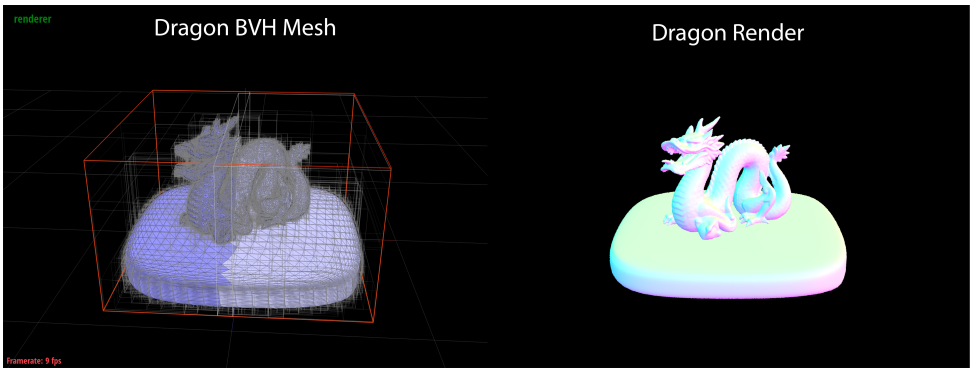
Render Time Comparison Results for Cow:

- Render time without BVH = 3.1832s
- Render Time with BVH = 0.4017s



Render Time Comparison Results for Dragon:

- Render time without BVH = 215.5702s
- Render Time with BVH = 0.6759s



CS 184 Project 3-1 ~ Ray Tracing

Part 3 - Direct Illumination

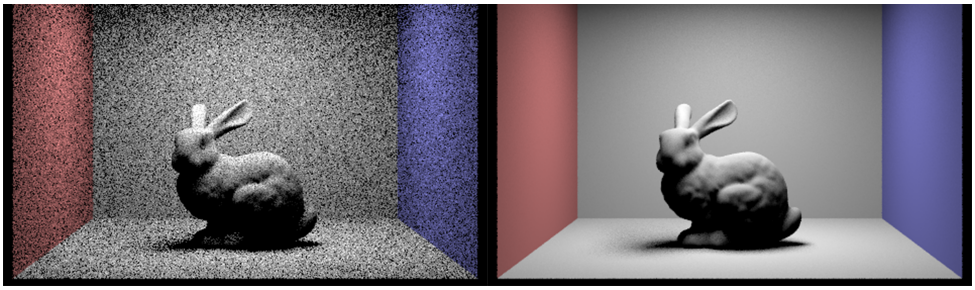
After implementing part 3 my program was able to render images with zero and one bounce illumination meaning that the only light that the program can detect from the image is light that either comes directly from a light source to the virtual sensor or light that is bounced one within the three-dimensional environment before hitting the virtual sensor.

The zero-bounce test sends the generated ray out into the 3d space and returns light if the ray intersects with any light sources and gives the value of the light being emitted from the light source.

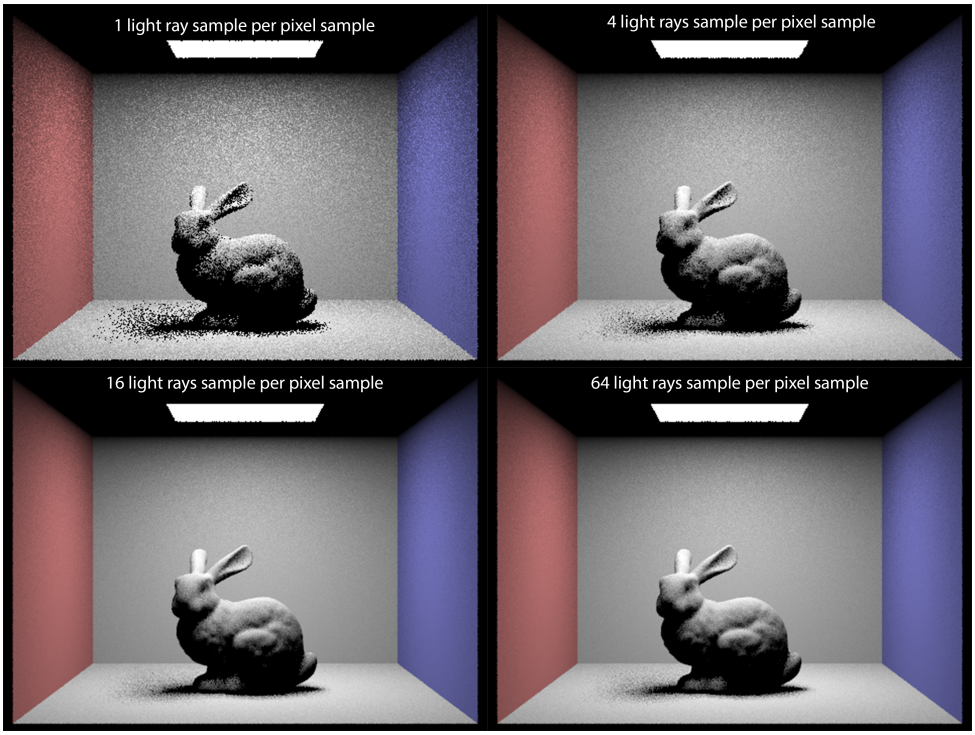
The one bounce test sends out a light ray into the three-dimensional space and does one bounce light ray bounce and gives the weighted sum of the light which is emitted from each of the two possible surfaces that it intersects. There are two methods that I implemented for determining the direction of the ray after the light bounce. These two methods are uniform hemisphere sampling and light importance sampling. Uniform hemisphere sampling takes its samples from the hemisphere radiating from the intersect point using a uniform probability distribution. Light importance sampling takes its samples from only directions that point towards a light source. I made sure to account for errors involved with the imprecision of a double when calculating the origin of each ray such that it would not intersect with the surface of intersection and also took this into account when calculating the distance from the point of intersection to the light that it is sampling.

Below is the comparison between uniform hemisphere sampling and importance sampling. The uniform hemisphere photo is noisier because it is less likely for a ray to intersect a light than it is in importance sampling so a lot of the pixels are darker than they should be due to the samples that didn't hit any light despite the location being sampled having no intersections between it and the light. This is what results in the noise shown in the image which uses uniform hemisphere sampling as the sample rate is only 16 samples per pixel. The image using importance sampling is less noisy because the sampled rays after the bounce are more likely to hit a light. As the sample rate per pixel increases, these photos will converge to become more similar.

CS 184 Project 3-1 ~ Ray Tracing



The below comparisons were captured using 1 sample per pixel and light importance sampling.



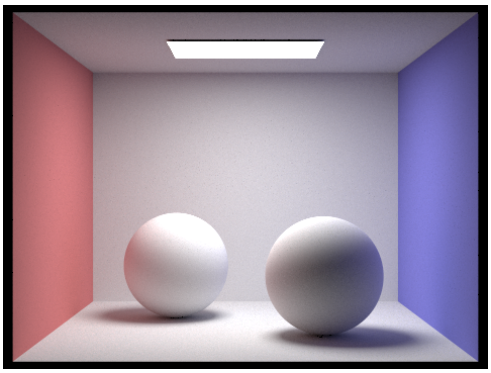
Part 4 - Global Illumination

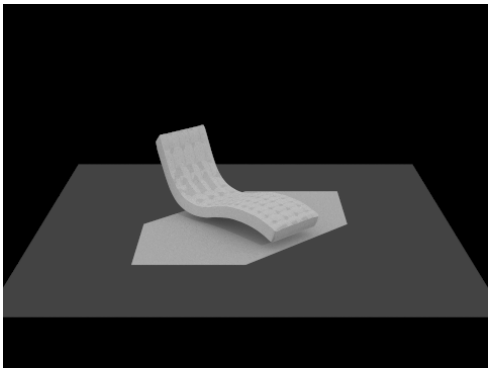
In this part of the project, I implemented a function that can calculate the indirect lighting in the photo. I did this by implementing the at_least_one_bounce_radiance() function which summates the estimate of the radiance that is emitted from each surface that the ray hits along an arbitrarily large number of bounces through the three dimensional virtual

CS 184 Project 3-1 ~ Ray Tracing

In the at least one bounce function, the function calculates the radiance emitted from each intersection using the reflection equation as described in lecture and then use the specified brdf sampling method to find the direction of the of the incoming ray and thus, the direction to the next intersection that the ray is going to bounce off. It does this by calling itself recursively if the Russian roulette function returns true for the specified termination probability, the ray depth has not exceeded max_ray_depth and the intersect function called a ray leaving the current point of intersection in the direction of the incoming ray returns true. The only exception to this is if the ray has not done any bounces yet and it intersects with an object in the virtual three dimensional space.

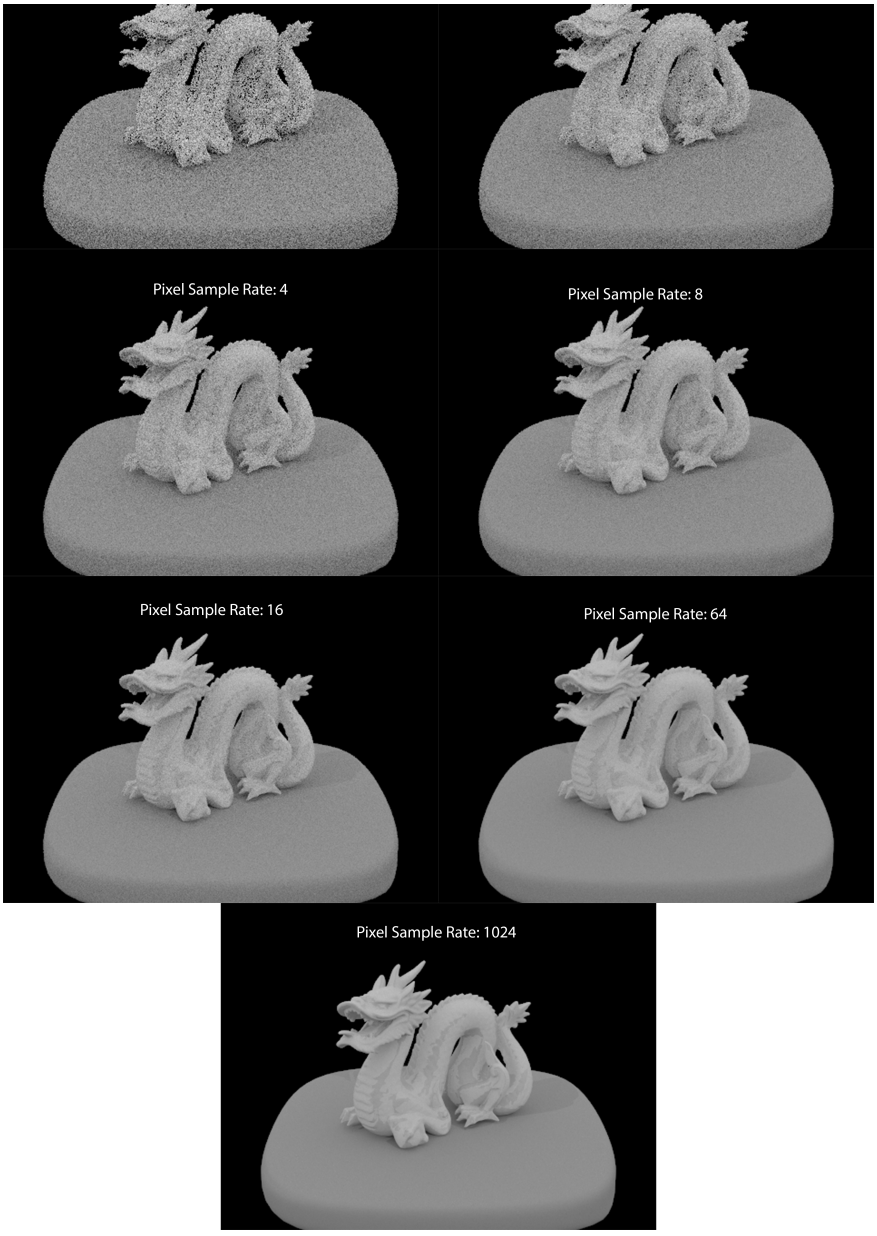
The following photos were rendered using global illumination with 1024 samples per pixel.





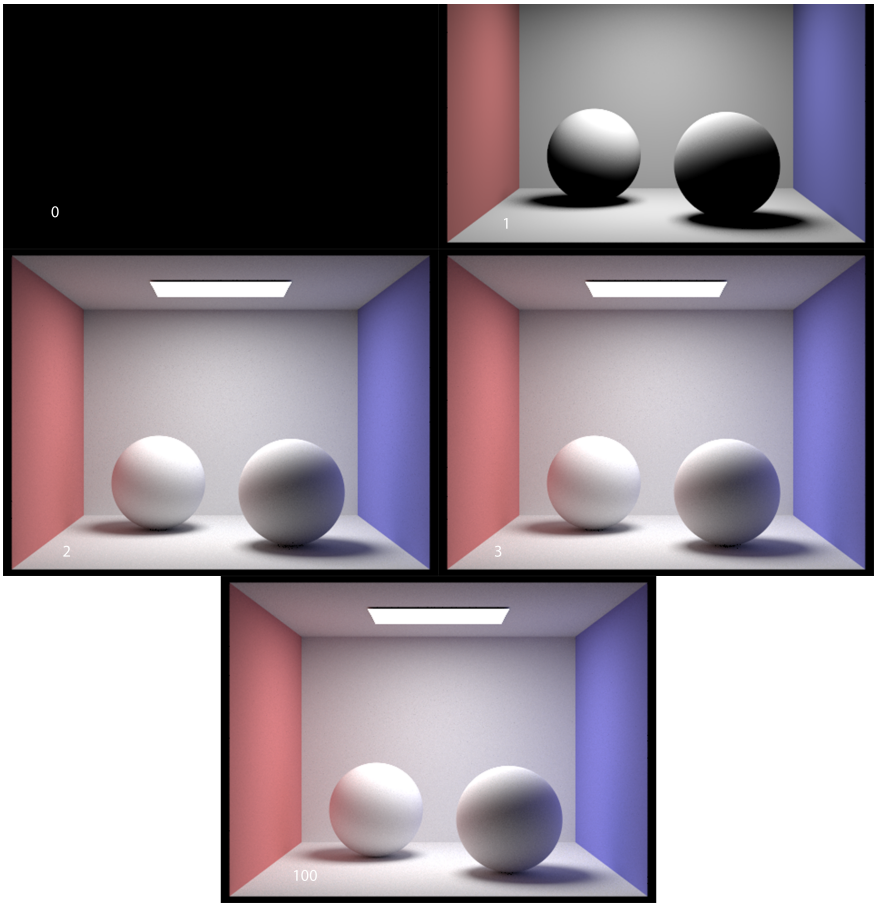
In the below photo the pixel sample rate is compared across a multitude of values. The max ray depth is 5 and the light ray sample rate is 4.

CS 184 Project 3-1 ~ Ray Tracing



The below photos have all have a sample rate of 1024 and the numbers on the photos represent the max ray depth used to calculate the global illumination.

CS 184 Project 3-1 ~ Ray Tracing



Part 5 - Adaptive Sampling

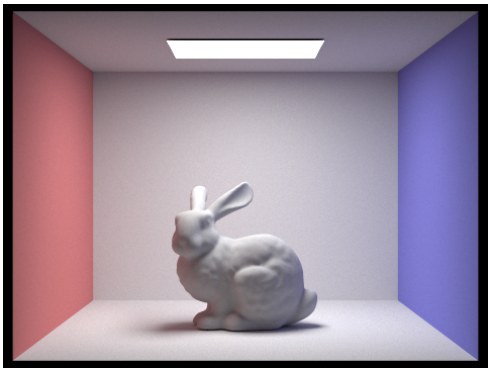
In Part 5 I implemented adaptive sampling such that my program can terminate sampling when it is (95%) confident that the color it has produced from the samples it has taken is very close to the desired color of the pixel. This allows the program to render images using higher sampling rates as it speeds up the process significantly at high sampling rates with low quality loss.

Data about the pixel's sample's expected value and and the expected value of the square of each sample's value means that at every step of the loop we can evaluate both the variance and mean of the sample data collected by the raytracing algorithm. Using these statistics, we can determine the pixel's

by the mean. The variable samples_per_batch described how often this is calculated as it is not computationally wise to recalculate ita after taking it every sample.

This process allowed my program to render very high-quality images at a lower cost than using a uniform sampling rate.

The following photo was rendered using 2048 samples per pixel with adaptive sampling.



This photo shows the number of samples taken across the image. Red presents the largest sampling rates and blue represents the lowest sampling rates.