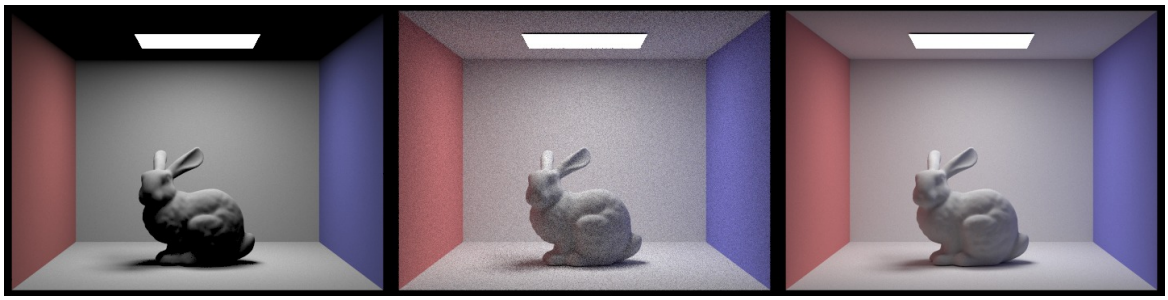
CS 184: Computer Graphics and Imaging

Project 3-1: Pathtracer

Jonathan Kim and Steven Christopher



Overview

In this project, we implemented the core routines of a physically-based renderer using a pathtracing algorithm. This assignment reinforces many of the ideas covered in class recently, including ray-scene intersection, acceleration structures, and physically based lighting and materials. Specifically, we had to implement the generation of camera rays and their intersections between spheres and triangles. Additionally, we implemented a bounding volume hierarchy structure that allows to simplify complex geometrical structures. We also used different sampling methods to implement different illuminations and lightings of the different scenes provided to us.

Part 1:Ray Generation and Scene Intersection

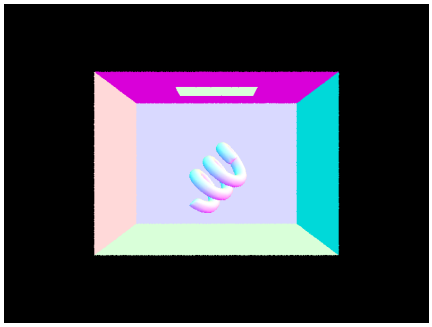
This portion of the project required us to implement a function that takes the normalized image coordinates (x, y)(x,y) as input and outputs a Ray in the world space, a function that takes pixel coordinates (x,y)(x,y) as input and updates the corresponding pixel in sampleBuffer with a Vector3D representing the integral of radiance over this pixel, and

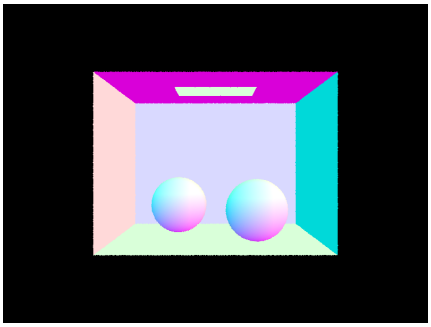
functions that test whether an intersection exists between a ray and a triangle/sphere.

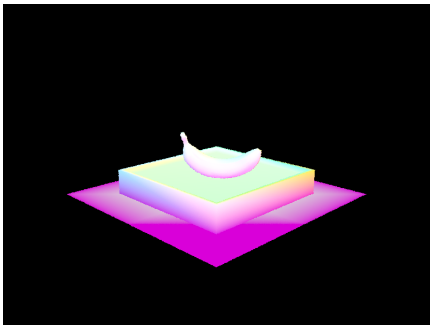
First, we generate rays from our camera by using a sampled coordinate (x,y) on the camera sensor plane at Z = -1. To do this, we use the fact that (0, 0) in the image space maps to (-tan(0.5 * hFov), -tan(0.5 * vFov), -1) in the camera space, and additionally (1, 1) in the image space maps to (tan(0.5 * hFov), tan(0.5 * vFov), -1) in the camera space. Using this, we can linearly interpolate to get the point at which this ray originating at the camera origin (0, 0, 0) will intersect this camera sensor plane and pass this in as the direction of the ray-after first transforming the direction vector from camera to world space and normalizing.

For the primitive intersection parts, involving triangles and spheres, different algorithms were used for each primitive. For the sphere intersection, a different algorithm is used to find the parameter t at which the ray intersects the sphere, involving use of the quadratic formula. Due to the quadratic formula, there are two possible solutions (t_0 and t_1), however we only choose the valid t which is the lesser of the two (taking the min only if both t_0 and t_1 are valid).

Specifically, for triangle-intersections we made use of the Moller-Trumbore intersection algorithm. This fast algorithm was implemented by computing the 5 components of the formula (E_1, E_2, S, S_1, S_2) using the points of the triangle and the origin and direction of the ray. These components (as well as the direction fo the ray) are used directly in the formula to get a vector containing [t, b_1, b_2] where t parameterizes the ray (only t >= 0 is valid, else the ray is going out in the opposite direction which we do not want), and b_1 and b_2 are the barycentric coordinates for the triangle. b_0, which is coordinate corresponding to p_0, can be calculated by (1 - b_1 - b_2) since these barycentric coordinates must sum up to 1 and each be in the range [0, 1]. In order to actually check that the above triangle intersection is valid, we must check that the barycentric coordinates are all in the range[0, 1], that t >= 0 and min_t <= t <= max_t (min_t and max_t are the bounds for which intersections are valid, and are initially set to the bounds of the camera, nClip and fClip). In the case of a valid intersection, max_t is updated accordingly so that future intersections do not allow intersections farther than the previous intersections which were found. Furthermore, we populate an Intersection object to keep track of the point of intersection, the surface normal to this point, and the primitive (this triangle) and its bsdf.


Spring Dae File


Sphere Dae File


Banana Dae File

Part 2: Bounding Volume Hierarchy

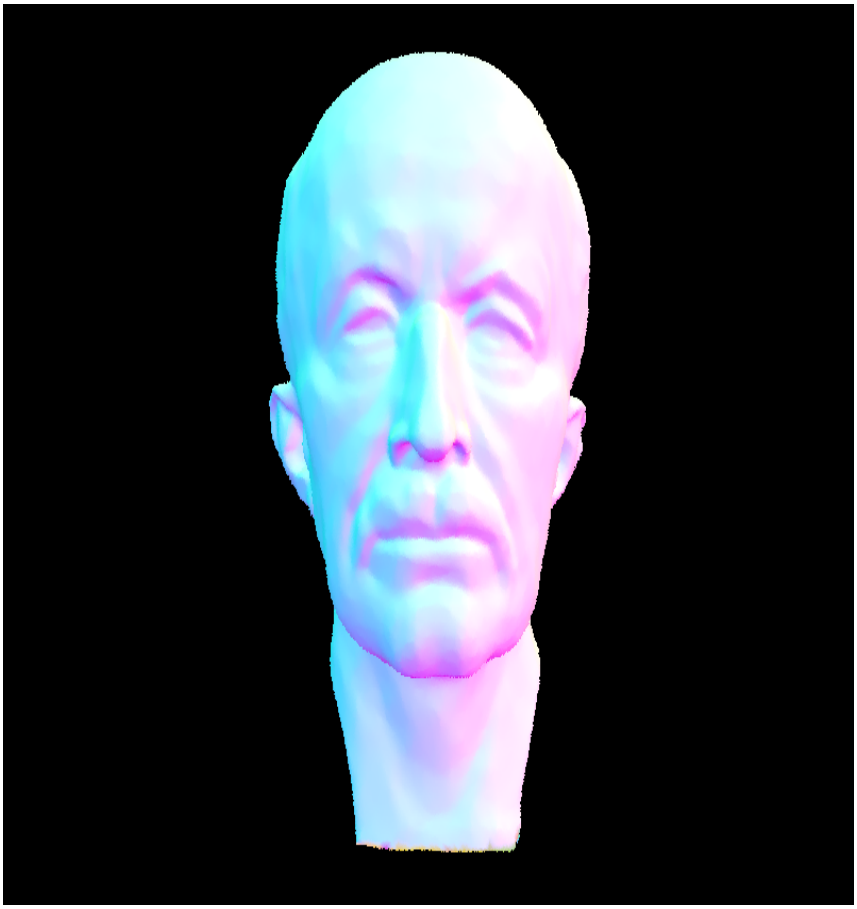
This portion of the project required us to create a BVH tree based on a list of primitives given to us, and create and test for intersections between an input ray and the different primitive nodes of the BVH.

After the completion of Part 1, the rendering of more complex scenes took a long time due to the many triangles needed to render the scene. This is due to fact that we checked if every single ray intersected an object. Therefore, to speed up the process we used bounding volume trees.

We constructed the BVH tree by first going through all the primitives and expanding their bounding boxes. During this process we calculated the average centroid after the bounding box was expanded. Then we checked if we were creating a leaf node by comparing the amount of primitives we have and the max_leaf_size. If it wasn't a leaf node and the creation of a left and right node were required, we found which axis we were going to compare the centroid on by finding the greatest extent between the x, y, z values of our bounding box's extent. Then we used the std::partition function to sort our primitives based on if they were greater or less than our average centroid calculation. We also made sure that there was at least one primitive when constructing our left and right nodes.


cow.dae


CBlucy.dae

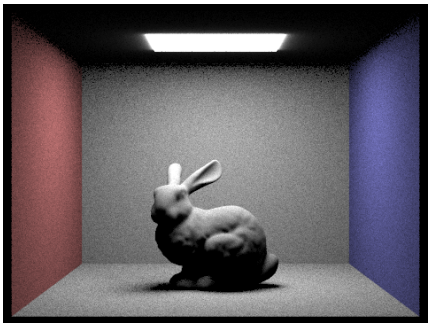


maxplanck.dae

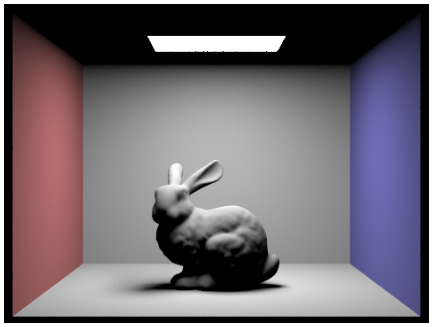
After implementing the BVH tree, we were able to reduce render times by a large fraction. For example, before BVH utilization, our cow.dae file took around 46 seconds to render. However, after implementing the BVH, we were able to render the file in 3 seconds. This was also the same for the maxplanck and CBlucy files as they took even longer than cow.dae to render due to the number of triangles. With BVH, they took a 7-8 seconds each. By checking for bounding box intersections before ray intersections, we can go through less nodes by getting rid of the ones that will never intersect. This can be reduced even further with a good heuristic going from O(n) to O(log(n)) due to the nature of trees.

Part 3: Direct Illumination

This portion of the project required us to implement two direct lighting estimation methods. We had to sample uniformly in a hemisphere or use importance sampling around each intersection. The estimation of the irradiance utilized the Monte Carlo integrations that we learned in class.



CBbunny.dae with Hemisphere Sampling

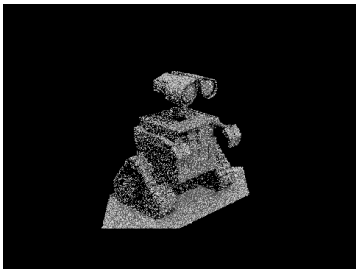


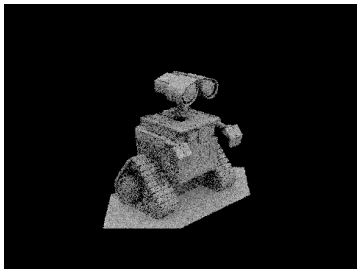
CBbunny.dae with Importance Sampling

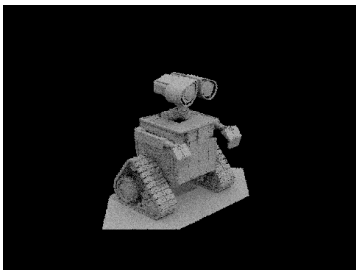
To implement estimate_direct_lighting_hemisphere, we first iterated num_samples amount of times which was given to us as the number of lights in the scene multiplied by the area of the lights. We then got a sample using the hemisphereSampler. We also created a ray by using hit_p as the origin and found the direction of the ray by converting the sample into world coordinates. If the sampled ray intersected with my BVH, we used the Monte Carlo estimate to get the material's emitted light. We multiplied the BSDF by the light emission, and a cosine term. We also divided by the pdf and finally added it to the L_out value. We then returned the sum of all the samples averaged.

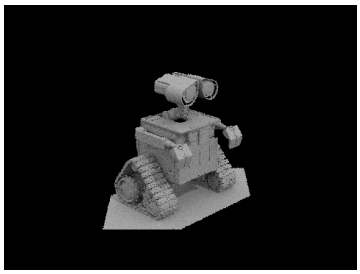
To implement importance sampling, we had to sample every light source in the scene by looping through all lights. If the light source was a delta light, we only needed to sample once since all the samples are the same. If not, we sampled ns_area_light amount of times. The sampler gives us the incoming radiance, the pdf, direction, and distance. So by using this information given to us, we can create a ray to test if it doesn't' intersect the BVH. We had to convert the direction vector from world to object coordinates. If there is no intersection, we summed up the irradiance by using the same calculation as in hemisphere sampling. We finally divided the sum by the number of samples we used.

In the pictures above, we can see the difference between hemisphere sasmpling and importance sampling when using the same amount of samples and light rays. There is more noise in the hemisphere sampling due to the fact that we take samples in all directions around a point, so only some rays will actually point towards a light source. This leads to the noisy spots along our scene. However, importace sampling will give more priority to samples that actually produce results. By only using samples from the light sources given to us, each ray will produce a value unlike hemisphere sampling. Therefore, our importance sampling image will be less noisy than that of the hemisphere sampling.


1 sample, 1 light ray

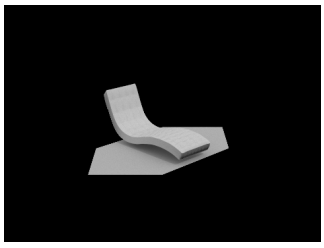

1 sample, 4 light rays

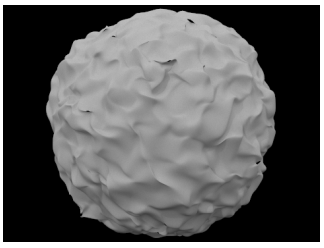

1 sample, 16 light rays

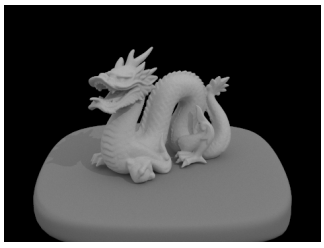

1 sample, 64 light rays

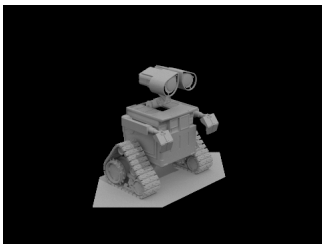
In the wall-e.dae renders, we tested only using importance sampling using only 1 sample per pixel and increasing the number of light rays. With fewer rays, we can see that there is a lot of noise in the soft shadows and with increasing amounts of light rays, we see the noise start to disappear. To decrease noise and make Wall-e happy, we should either increase the number of samples, increase the amount of light rays, or both.

Part 4: Global Illumination


Bench


Blob

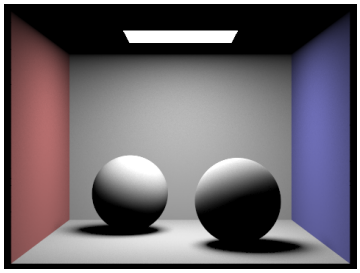

Dragon

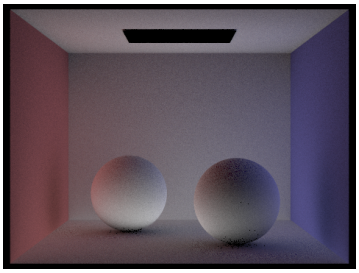

Wall-E

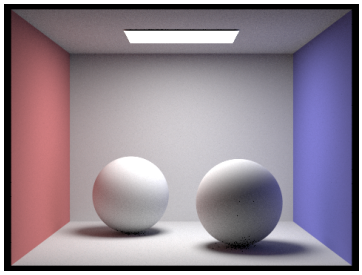
This portion of the project required us to implement global illumination by accounting for multiple bounces of light rather than just one bounce of light.

To implement global illumination, we had to fill out the at_least_one_bounce_radiance function. We did this by checking the max_ray_depth first. If it was 0, then we'd return a 0 vector. If it was one, we would just return the one_bounce_radiance as it is depth of 1. If it was

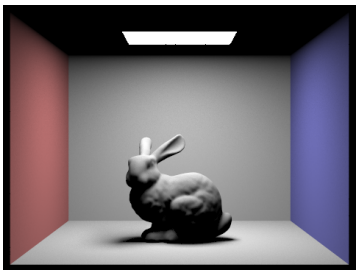
greater than one, we would need to use recursion and Russian Roulette to randomly terminate the sampling sequence. We then construct a ray from the current point and check if it intersects the BVH. We also make sure that the depth of the ray is equivalent to the input ray's depth - 1. If it doesn't hit, we stop, but in the case where is hits and pdf is greater than 0, we recurse using the ray we just found and summing up the resulting radiance like how we calculated in hemisphere and importance sampling. For each iteration, we had a probability of 0.7 of the sample being continued meaning that the sample wasn't continued 30% of the time.

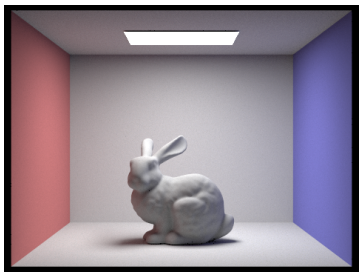

Direct Lighting Only

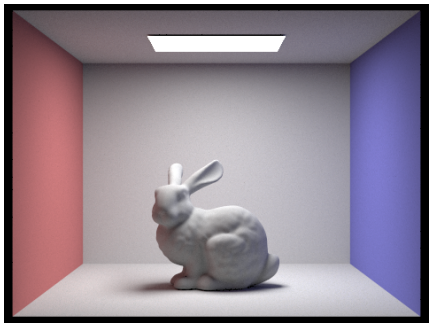

Indirect Lighting Only

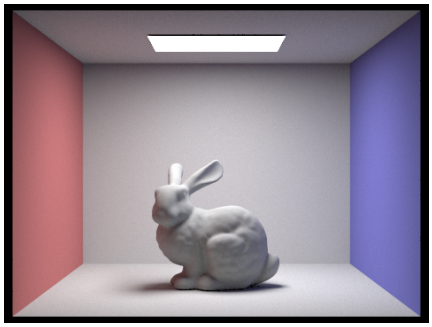

Both Direct and Indirect Lighting

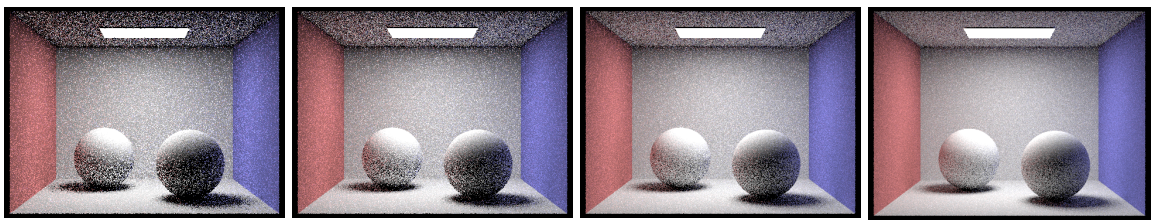

Max Ray Depth 0


Max Ray Depth 1

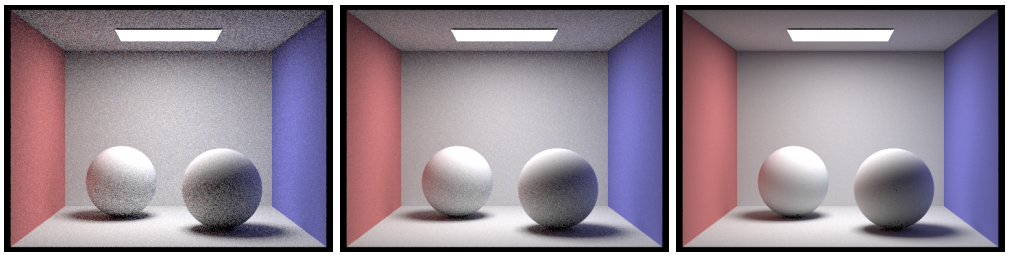

Max Ray Depth 2


Max Ray Depth 3


Max Ray Depth 100



1 Sample 2 Samples 4 Samples 8 Samples



16 Samples 64 Samples 1024 Samples

Part 5: Adaptive Sampling

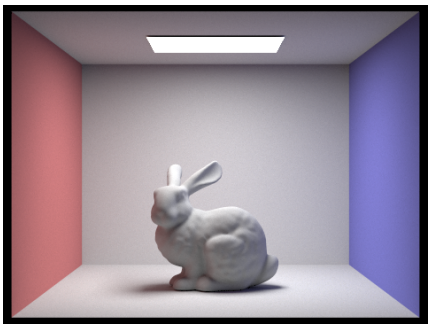
This portion of the project required us to implement enable adaptive sampling. It works for each pixel individually based on statistics, detecting whether the pixel has converged as we trace ray samples through it.

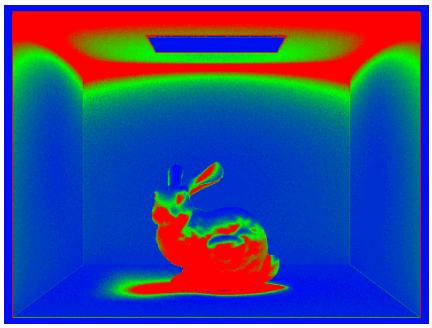
In order to implement adaptive sampling, changes had to be made to the method which raytraces each pixel. The additions necessary for adaptive sampling included z-testing by calculating the sample mean and variance of the illuminance and trying to find out when a pixel has converged through Monte Carlo Estimation.

First, how do we calculate sample mean and sample variance? We kept track of two variables: s_1 and s_2. Variable s_1 is the sum of the illuminance up to the amount of samples that have been taken, and variable s_2 is the sum of the illuminance squared up to the amount of samples that have been taken. To get the illuminance of a pixel, we use the illum() method of a Vector3D, specifically the one associated with our Monte Carlo Estimate. Then, we are able to compute the mean (mu) = (s_1 / number of samples) and variance (sigma^2) = (s_2 - (s_1)^2 / n) / (n - 1).

Once we have the mean and variance, we can calculate the error from the true mean (mu ± I) where I is the error / epsilon. To do this, we take advantage of z-testing, calculating a confidence interval (95%) of the true mean over the illuminances of the pixels. Here, I = 1.96 * sigma / sqrt(number of samples taken so far), such that sigma is the standard deviation and coefficient 1.96 comes from the z*-value for a 95% confidence interval. Specifically, if variable I <= maxTolerance * mu, where maxTolerance is the deviation from the true mean which we are accepting, then this means that the current pixel being processed has converged (since it lies within the 95% confidence interval).

However there's one issue: calculating the sample mean, variance, and variable I for EACH sample is EXTREMELY expensive. To deal with this, we have a constant samplesPerBatch, which signifies that we will only check if a pixel has converged if we are currently on the ith sample such that i is a multiple of samplesPerBatch. This is set to 32 by default, so we check this every 32 samples. Therefore, it is during this condition being true that we check if variable I is less than or equal to maxTolerance * mu, and if so, the respective pixel in the sampleBuffer is updated with the Monte Carlo Estimate over the total number of samples SO FAR, as well as the sampleCountBuffer being updated with the total number of samples so far.


Render


Sample Rate of Each Pixel

For this project, we first tried brainstorming ideas for each portion of the project and then broke down the responsibilities by parts. In terms of coding, we had to re-allocate who led certain parts. Specifically parts 3 and 4 were connected and therefore Jonathan took up leading those sections whereas initially Steven had done parts of 4. In the end, we both ended up figuring out the implementation together for all 5 sections. We debugged together whenever we encountered a problem because each person had more knowledge about the code they wrote. This allowed us to understand what each other's code did and the reasoning behind each decision. Regarding rendering pictures for the report, Steven did renders for parts 1, 5, and 4 (sample variation renders) while Jonathan did the renders for part 2, 3, and 4 (indirect/direct/both lighting renders). We can say with certainty that the bulk of our time was spent on figuring out how to best render our images. We took a lot of time trying to set up rendering on instructional machines but encountered too many problems and just went back to rendering solely on local machines. Therefore, we both split up render jobs as was explained in the breakdown. Additionally, renders took a long long time. We needed to allocate more time to do these parts of the write-up as they consumed most of the time used up during slip days.

https://cal-cs184-student.github.io/sp22-project-webpages-jkim1123/