# CS 184: Computer Graphics and Imaging, Spring 2023

# Project 3: Pathtracer 1

Jonathan Guo, Wentinn Liao, cs184jgwl

## Overview

In this project, we implemented some path and ray tracing algorithms. We computed intersections of rays with spheres and triangles and constructed bounding volume hierarchies (BVH) to speed up intersection computation. To speed up the rendering, we utilized importance sampling, russian roulette, and adaptive sampling.

## Part I: Ray Generation and Scene Intersection

To generate rays from the camera, first we took the x and the y values and translated them into the camera space x and y by multiplying by the width of the viewplane and translating appropriately. Next, we turned them into world space by multiplying by the c2w matrix. Finally, there was the normalization and adding of position that occurs normally.

The triangle intersection algorithm we implemented was based off slide 22 in lecture 9, the optimized version. But what it basically does is that we know that the point of intersection of the ray and the triangle is on the plane of the triangle, which means the dot product between the plane's normal and a vector on the plane to the point of intersection is zero. Once there, we can compute the barycentric coordinates of the point of intersection with respect to the triangle, and it is inside the triangle if and only if all of the coordinates are non-negative.

*dae/meshedit/teapot.dae.*

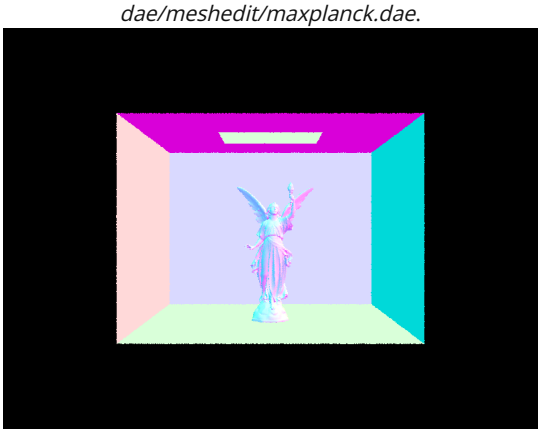
*dae/meshedit/cow.dae.*

## Part II: Bounding Volume Hierarchy

Our Bounding Volume Hierarchy construction algorithm took the longest side of the box and split the objects by the midpoint of that side. It terminated if the all of the primitives were on one side of the midpoint. Although this is not ideal, in practice it almost always does not occur until the very end when there are very few primitives left.

*dae/meshedit/cow.dae.*

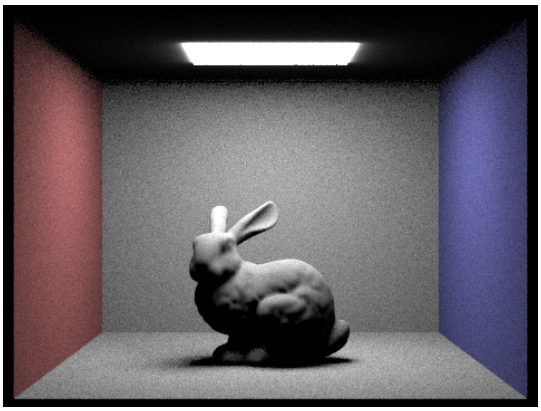
*dae/meshedit/maxplanck.dae.*



*dae/sky/CBlucy.dae.*

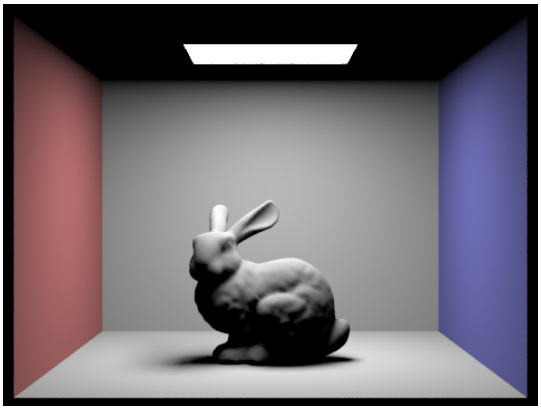
## Part III: Direct Illumination

The direct lighting function with uniform hemispheric sampling treats all samples identically. For each sample, we sample a random vector in a z-positive hemisphere. Because this is z-positive with respect to the object, this is the incoming direction in the object space, w_in_o which is translated to the incoming direction in the world space w_in_w using o2w. We then use the BVH data structure to compute the first intersection of the uniformly sampled ray with an object, denoted incoming_isect, and obtain its emitted radiance. We then multiply by the BSDF at the camera ray intersection with the outgoing ray to the camera and the incoming ray from the emitting object and cosine of the incoming angle and add to a cumulative sum of radiance. Finally, we divide by the uniform PDF (or multiply by 2 pi, and divide by the total number of samples.



Bunny Uniform Hemispheric Sampling at Rate 32.

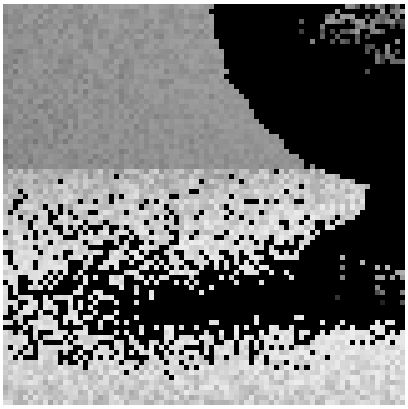
Importance lighting sampling iterates over each light for ns_area_light except for delta lights for which we only sample once. Calling sample_L from the light data structure gives the incoming direction in the world space, distance to the light, the PDF, and the radiance itself. The incoming ray is translated to object space and the ray is clipped to the time to the light so the BVH structure can be used to check for intersections with any other objects. If there is no intersection, then the same lighting equation is used, where the radiance is multiplied by the BSDF at the pixel intersection, the cosine of the incoming angle, and divided by the PDF and number of samples. The difference here is that we are keeping track of a sum where PDFs are non-uniform, and the number of samples across lights are different. We cannot make the same assumptions as before that allow us to optimize by doing multiplications at the end, so importance sampling does so properly during the loop.



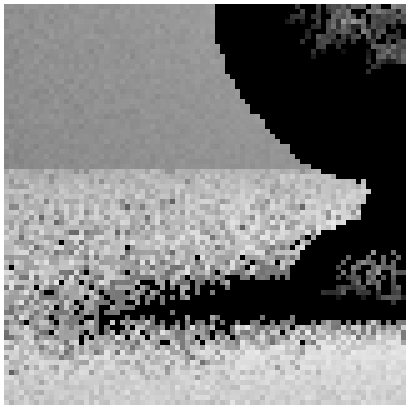
Bunny Importance Lighting at Rate 32.

Lighting sampling is a form of importance sampling in which we sample with higher probability the rays that will contribute the most to the integral. Intuitively in this case, the rays that have been determined to "contribute the most" are ones that point to a light source. Particularly in this case, we are choosing to sample from a close-to-uniform distribution in the solid angle of rays to each light source, and zero distribution elsewhere. In effect, rather than integrating over the entire hemisphere, we are summing integrations over only solid angle regions that correspond to light sources. Because the number of samples remains the same, we are redirecting all samples to regions we know have a higher chance of reflecting light, which significantly improves the quality of the rendering. In general, importance sampling might not explicitly exclude particular regions of the integral, but in can be interpreted to be, with higher probability, allocating more samples to regions of greater or explicit interest to better estimate their integrals.

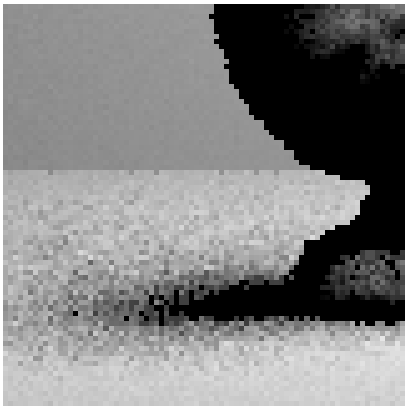
Shown below are images of a corner of the bunny in which soft shadows are sampled at a single ray per pixel, and at sampling rates of 1, 4, 16, and 64 respectively. We can see that the lower sampling rates are significantly more noisy. This is expected because the concept of soft shadow is centered on pixels at similar positions varying in shading not due to differences in distance or angles, but due to varying amounts of occlusion in lighting. With lower samples, the variation in occlusion is much less fine grained since averaging over fewer samples produces a higher variance estimate. For example, at a sampling rate of 1, all pixels within the shadow of the bunny now have shading that is binary, determined by whether a single random ray hits the bunny or not, which gives it the noisy appearance. As the number of samples increases, the average becomes a much more consistent approximation of the exact integral of occlusion per pixel.



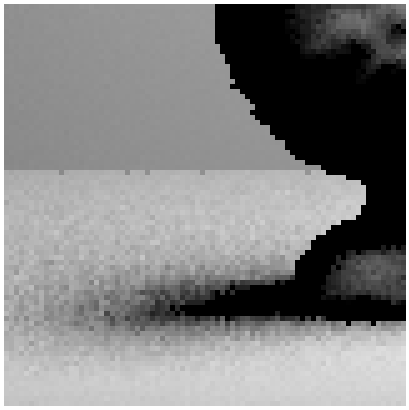
Soft shadow with sampling rate of 1.



Soft shadow with sampling rate of 4.



Soft shadow with sampling rate of 16.



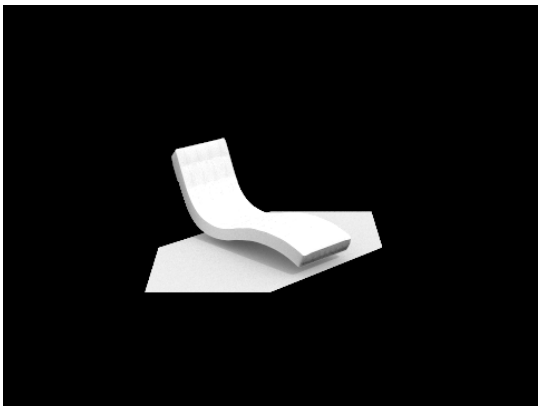
Soft shadow with sampling rate of 64.

## Part IV: Global Illumination

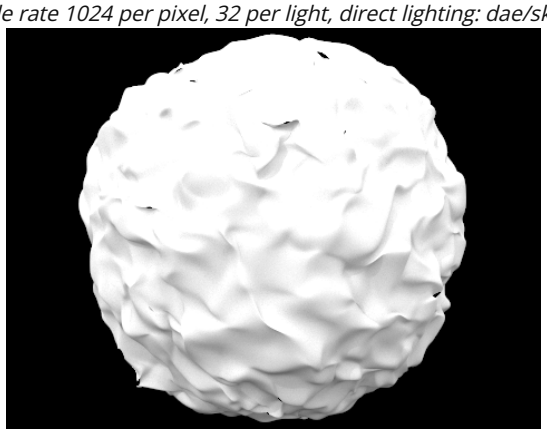
Our indirect lighting implementation occurs inside the **at_least_one_bounce_radiance** function, and first checks if the depth is greater than 1, and if not returns the one bounce radiance immediately. Indirect lighting is then computed by the remainder of the function. Using the desired outgoing direction, we sample the bsdf of the given intersection with **sample_f** to obtain the BSDF, PDF, and the sampled incoming ray, which is denoted by **next_d_in_o**, which occurs in the object space. This incoming direction is translated to world space with **o2w** and along with the hit point is used to construct the next ray. The new ray's depth is decremented once from the given ray.

With Russian Roulette implementation, we go on the conditions that the Bernoulli random variable comes up heads, or 1, and the newly constructed ray intersects at least one object. Because the coin flip is much faster, it is checked first, and the intersection is checked second with the BVH data structure. Finally, indirect lighting is calculated by recursively calling **at_least_one_bounce_radiance** on the new ray and inew intersection point, with the sampled BSDF and **next_d_in_o** as incoming angle, and the added term is divided by the PDF, and the continuation probability, in order to maintain the same expected outcome. The recursive call makes sense because all indirect lighting occurs from a combination of direct and indirect lighting (at least one bounce) at other places that reflect back to the point of interest.

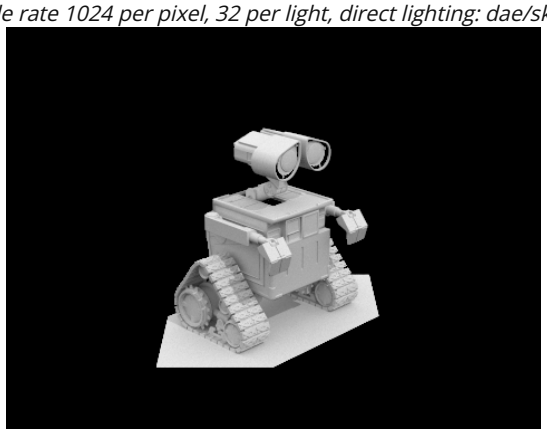
Also, below are some images rendered at 1024 samples per pixel and 32 samples per light using global illumination.



*Bench: sample rate 1024 per pixel, 32 per light, direct lighting: dae/sky/bench.dae.*

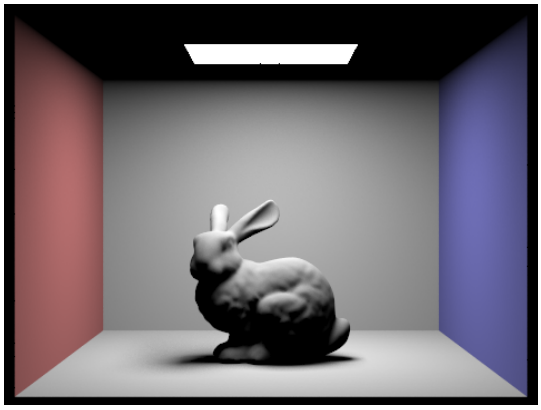


*Blob: sample rate 1024 per pixel, 32 per light, direct lighting: dae/sky/blob.dae.*

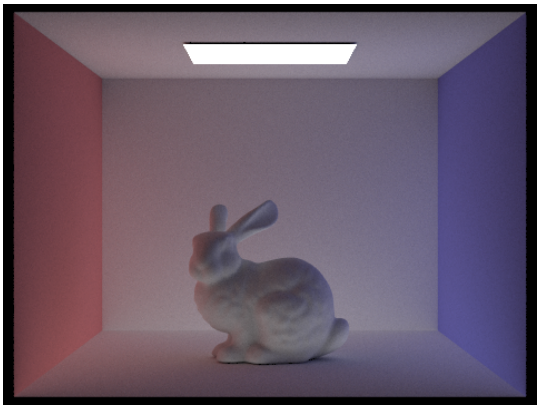


*Wall-E: sample rate 1024 per pixel, 32 per light, direct lighting: dae/sky/wall-e.dae.*

The images below show only direct and indirect lighting for the bunny scene. Direct lighting looks more normal than indirect lighting, since indirect lighting ignores the light that hits the pixel where the camera is pointing at. However, if you add the two, you should get the regular scene.



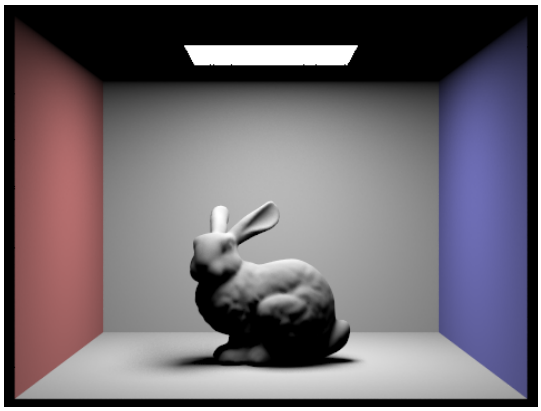
*Bunny: sample rate 1024 per pixel, 32 per light, direct lighting: dae/sky/CBbunny.dae.*



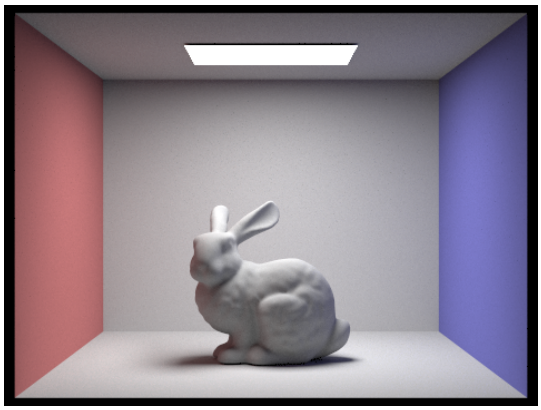
*Bunny: sample rate 1024 per pixel, 32 per light, indirect lighting: dae/sky/CBbunny.dae.*

Below, we compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100. With max ray depth 0, it is just zero bounce illumination, so the only things that are lit up are light sources themselves. With 1, it is the same as part 3, one bounce radiance. As our max ray depth increases, the scene gets brighter and brighter since we add more to our radiances.

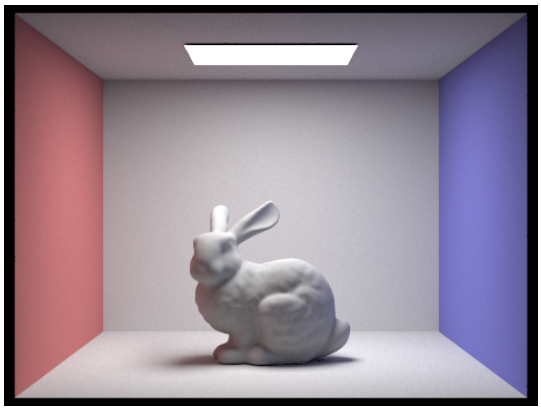
*Bunny: sample rate 1024 per pixel, 32 per light, max ray depth 0: dae/sky/CBbunny.dae.*



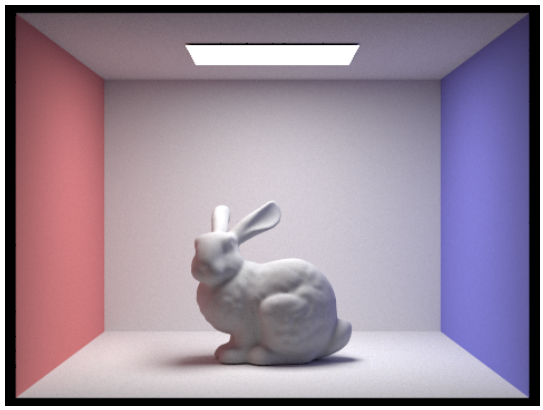
*Bunny: sample rate 1024 per pixel, 32 per light, max ray depth 1: dae/sky/CBbunny.dae.*



*Bunny: sample rate 1024 per pixel, 32 per light, max ray depth 2: dae/sky/CBbunny.dae.*

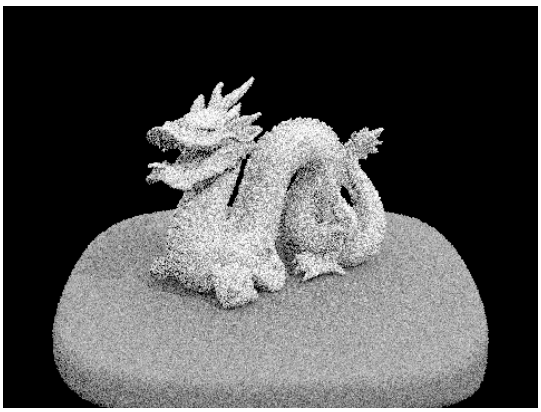


*Bunny: sample rate 1024 per pixel, 32 per light, max ray depth 3: dae/sky/CBbunny.dae.*

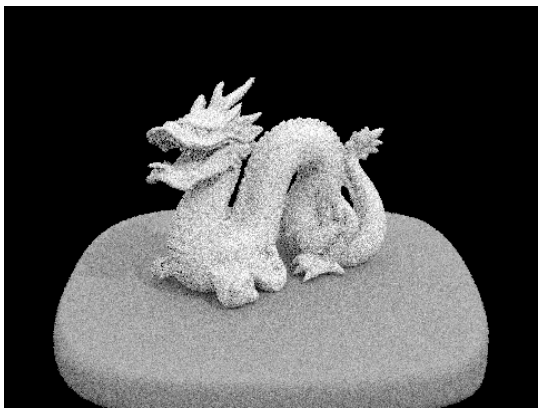


*Bunny: sample rate 1024 per pixel, 32 per light, max ray depth 100: dae/sky/CBbunny.dae.*

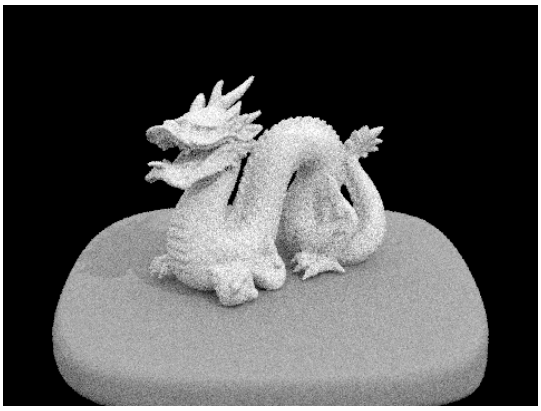
Shown below are renderings of the dragon with global illumination and ray-per-pixel sampling rates in powers of 2, scaling up to 1024. Global illumination makes the image much brighter due to self-reflectance, and scaling up to 1024 rays per pixel gives the rendering solid definition.



*Dragon: sample rate 1 per pixel, 4 per light: dae/sky/dragon.dae.*

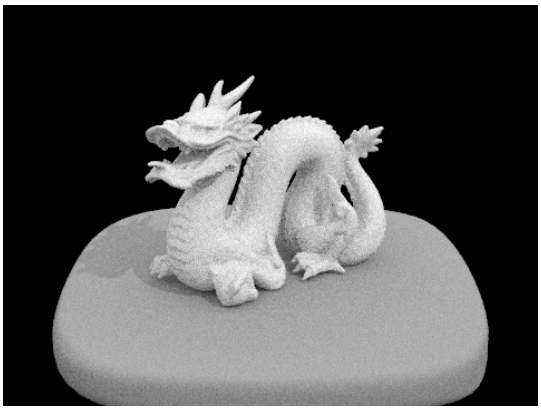


*Dragon: sample rate 2 per pixel, 4 per light: dae/sky/dragon.dae.*

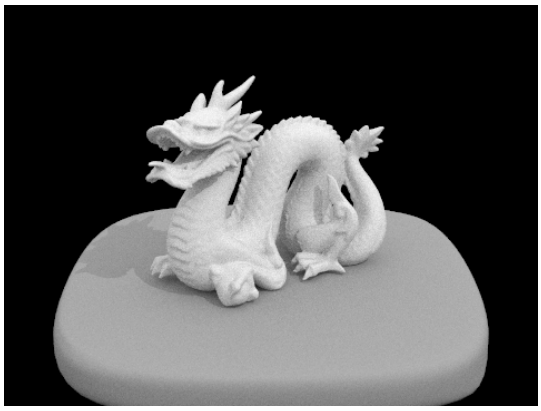


*Dragon: sample rate 4 per pixel, 4 per light: dae/sky/dragon.dae.*

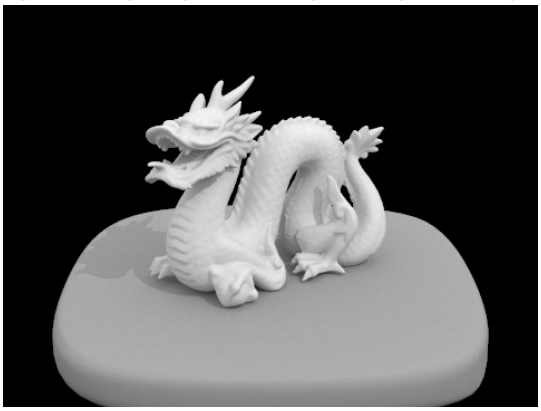
*Dragon: sample rate 8 per pixel, 4 per light: dae/sky/dragon.dae.*



*Dragon: sample rate 16 per pixel, 4 per light: dae/sky/dragon.dae.*



*Dragon: sample rate 64 per pixel, 4 per light: dae/sky/dragon.dae.*

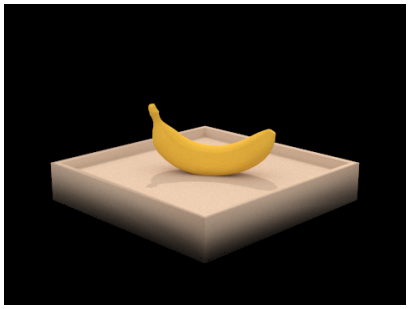


*Dragon: sample rate 1024 per pixel, 4 per light: dae/sky/dragon.dae.*

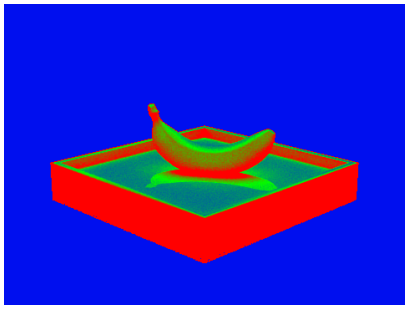
## Part V: Adaptive Sampling

Adaptive sampling is a performance optimization that empirically checks for convergence to determine whether more rays need to be sampled for that pixel. In general, the pixel will converge early if the contours around the point of intersection are relatively flat, facing the camera, meaning most generated rays intersect scene in roughly the same way. In contrast, surfaces that face diagonally or near perpendicular to the camera vary significantly even with small deviations in the rays, meaning that averaging many rays is required to accurately estimate the pixel shading.

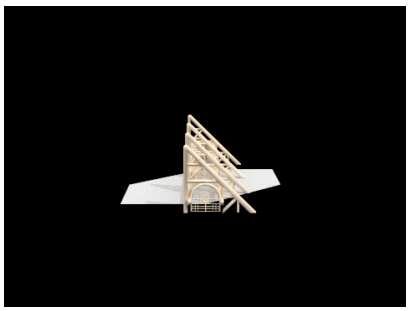
Our implementation keeps a running sum of the illuminance and illuminance squared, denoted $s1$ and $s2$ respectively. The check for convergence is done at the beginning, where the counting index $n$ also represents the number of samples counted. If it is both greater than zero, a multiple of the number of samples per batch, and satisfies the convergence requirements, then we break the loop. To minimize the number of computations, we rearrange the equation to $n * s2 \leq ((n - 1) * c - 1) * s1^2$ where $c$ is a precomputed constant $(maxTolerance / 1.96)^2$ which is an equivalent and much faster check for convergence. Finally, rather than dividing by $num\_samples$, we divide by $n$ and record that as the true number of samples.



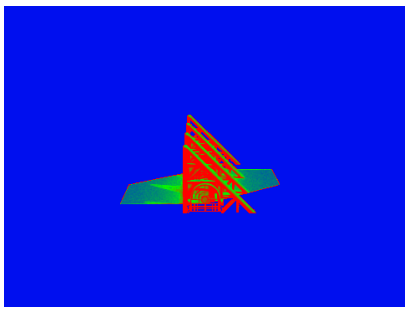
Rendering of banana with adaptive sampling.



Sampling rates with adaptive sampling.



Rendering of building with adapative sampling.



Sampling rates with adaptive sampling.

In the above renderings of the banana and building, we can see clear differences between the number of samples used for each pixel. Especially in the outer regions where there are no objects, rather than use 2048 rays per pixel, we are now only using 32 rays per pixel. The pixels that correspond to more detail or vary more in exposure to lighting converge more slowly and require more samples.