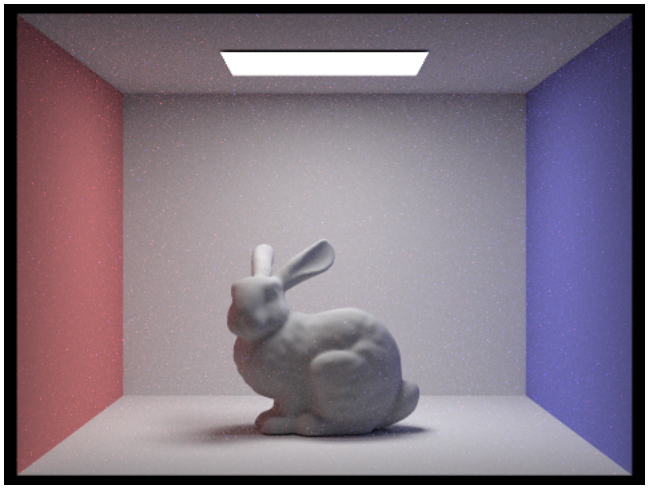
CS 184: Computer Graphics and Imaging, Spring 2023

Project 3-1: Path Tracer

Claire Liu, Justin Huey

Website URL: https://cal-cs184-student.github.io/project-webpages-sp23-JustinHuey1/



Results Caption: my bunny is the bounciest bunny

All of the text in your write-up should be *in your own words*. If you need to add additional HTML features to this document, you can search the http://www.w3schools.com/ website for instructions. To edit the HTML, you can just copy and paste existing chunks and fill in the text and image file names appropriately.

The website writeup is intended to be a self-contained walkthrough of the assignment: we want this to be a piece of work which showcases your understanding of relevant concepts through both mesh images as well as written explanations about what you did to complete each part of the assignment. Try to be as clear and organized as possible when writing about your own output files or extensions to the assignment. We want to understand what you've achieved and how you've done it!

If you are well-versed in web development, feel free to ditch this template and make a better looking page.

Here are a few problems students have encountered in the past. Test your website on the instructional machines early!

- Your main report page should be called index.html.
- Be sure to include and turn in all of the other files (such as images) that are linked in your report!
- Use only *relative* paths to files, such as

  ```
  "./images/image.jpg"
  ```

  Do *NOT* use absolute paths, such as

  ```
  "/Users/student/Desktop/image.jpg"
  ```

- Pay close attention to your filename extensions. Remember that on UNIX systems (such as the instructional machines), capitalization matters.

  ```
  .png != .jpeg != .jpg != .JPG
  ```

- Be sure to adjust the permissions on your files so that they are world readable. For more information on this please see this tutorial: http://www.grymoire.com/Unix/Permissions.html
- And again, test your website on the instructional machines early!

Here is an example of how to include a simple formula:

```
a^2 + b^2 = c^2
```

or, alternatively, you can include an SVG image of a LaTex formula.

Overview

In this assignment, we focused on creating a renderer using path tracing algorithms that can output images that are realistic and clean looking. We were able to do this by generating rays from the camera to the scene and detecting whether they intersected with primitives in the scene, most notably primitives that are triangles and spheres using basic and optimized equations that were taught in lecture. We were further able to optimize ray tracing and make rendering faster by implementing bounding volume hierachies to bound the shape of the object and detect intersections between the ray and the bvh. This made our rendering way faster as it optimized the run time to logarithmic rather than the regular basic linear runtime. To make the images more realistic looking, we calculated lighting within the scene using direct illumination and global illumination which lit up the scene and added soft shadows and coloration in the reflection of the objects. We concluded by implementing adaptive sampling to better filter out noise. This optimized how we were able to obtain our samples and made our rendering process much better.

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

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

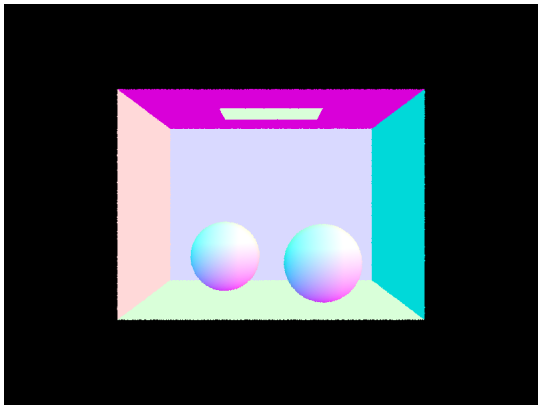
We followed the steps of transforming the image coordinates to camera space, then generating the ray in the camera space, and then transforming it into a ray in the world space. To transform the image coordinates to camera space, we had to keep the proportions correct so this involved calculating the width and height of the camera space sensor and then getting the camera world x and y values by adding the bottom left values (-tan(radians(hFov) / 2), -tan(radians(vFov) / 2)) to the correct proportional value (width * x, height * y). The ray in camera space therefore looks at the -z direction at this x,y coordinate. To transform the ray in world space, we get the world direction vector by multiplying c2w, the camera to world rotation matrix, by the camera space ray and then normalize. Using the camera position in the world space as origin and then the world direction vector as direction gives us the output ray in world space. For our ray, we also added a min and max bound.

For the primitive intersection, we relied on using the formulas given to us in lecture to be able to calculate whether a ray intersects with the shape object. In general, this meant identifying whether a given ray intersects with the object, and then if it does, being able to find the closest intersecting point and replace the rays max boundary and populating the intersection structure with data of the time where the intersection occurs, the surface normal at the intersection, primitive that was intersected, and surface material at the hit point.

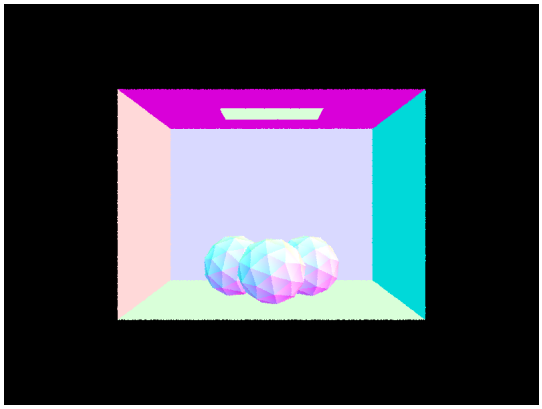
Explain the triangle intersection algorithm you implemented in your own words.

We decided to use the Moller Trumbore algorithm since it is an optimization technique to hopefully make our rendering faster. This algorithm calculates the intersection point of a ray with a triangle and outputs in barycentric coordinate for whether it is inside the triangle or not. We are basically solving for (1 - b1 - b2)P0 + b1P1 + b2P2, where Pi corresponds to the coordinates of the ith triangle vertex, to find the intersect point. To determine whether the ray intersects the triange, we have to check the output vector of the algorithm. If the b1, b2, and b3 are all greater than or equal to 0 and sum to 1 (the same case as barycentric coordinates), and the t value is within the min and max bounds, then there is an intersection.

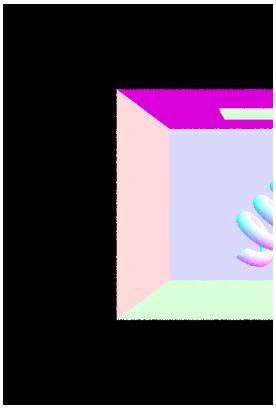
Show images with normal shading for a few small .dae files.



spheres



gems



co

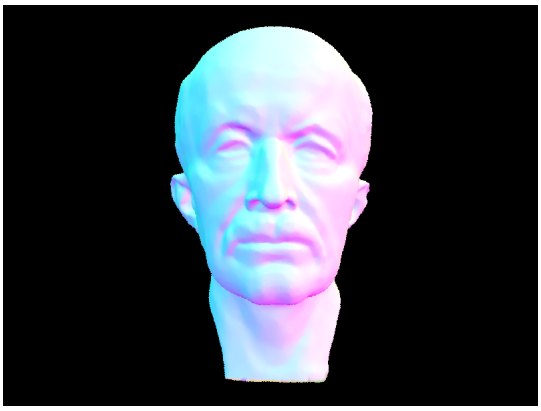
Part 2: Bounding Volume Hierarchy (20 Points)

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

We start by creating a bbox of the current node. This is done by looping through all the primitives in the bvh and unioning the bbox of all the primitives. As for what to return. We first check to see if the number of primitives in the bvh is less than or equal to the max_leaf_size. If that is true, then we make the node a leaf node and make node -> start = start and node -> end = end since nothing changed and we want to keep all the primitives. If the node is not a leaf, then we have to figure out how to break down the bvh. For our splitting aglorithm, we choose to split by the largest axis and partition about the middle of all primitives. This makes it so that half go to one box and the other half go to another box to evenly distribute the primitives, which will help with our runtime as each of the bounding volumes will have an equal number of primitives. We loop through the primitives from start to end and sort by their midpoint of the largest axis from the smallest midpoint to the biggest. We then recurse by splitting the first half into the left node and the remaining half into the right node.

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


cow


maxplanck


Lu

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

```
[PathTracer] Rendering... 100%! (32.9896s)
[PathTracer] BVH traced 478494 rays.
[PathTracer] Average speed 0.0145 million
    rays per second.
[PathTracer] Averaged 1039.138928
    intersection tests per ray.
```
cow regular time

```
[PathTracer] Rendering... 100%! (0.3093s)
[PathTracer] BVH traced 413213 rays.
[PathTracer] Average speed 1.3359 million
    rays per second.
```
cow bvh acceleration time

```
[PathTracer] Rendering... 100%! (334.1890s)
[PathTracer] BVH traced 473598 rays.
[PathTracer] Average speed 0.0014 million
    rays per second.
[PathTracer] Averaged 10487.033262
    intersection tests per ray.
```
maxplanck regular time

```
[PathTracer] Rendering... 100%! (0.3292s)
[PathTracer] BVH traced 429130 rays.
[PathTracer] Average speed 1.3035 million
    rays per second.
[PathTracer] Averaged 6.552341 intersection
```
maxplanck acceleration bvh time

As you can see, implementing bounding value hierarchy for ray tracing allows increased acceleration for the rendering time. This can take objects that might take a couple of minutes to normally render into just a couple of seconds. The acceleraition is even more apparent for models with complex objects and hundreds of thousands of triangles as seen in the maxplanck case. This is due to the nature of bvh where we can quickly compute if a ray will miss the bounding volume, which means that there is no chance for the ray to hit the object so we don't have to use more computing power to check. This allows us to achieve o(log n) time complexity from o(n). We can reduce this even more if we implement better heuristics for splitting the bvh to make primitives more evenly distributed and remove extra empty space in the bounding volume.

Part 3: Direct Illumination (20 Points)

Walk through both implementations of the direct lighting function.

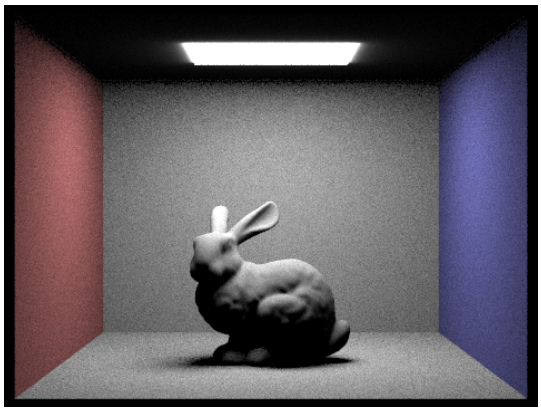
Hemisphere: For the hemisphere lighting, we create a for loop to loop through num_samples times. We create a sample using the hemisphere sampler. We currently can't create a ray out of this sample since it is in object space. Since the sample we use is in the object space, we want to transform it into the world space so we multiply o2w by the sample to get to the world space. We can then

create the ray using hit_p + (EPS_F * directionWorld sample) as the origin and directionWorld sample as the direction. We then have to test to see if the ray intersects with a primitive. If it does intersect, then we use the reflection equation to calculate the radiance being reflected. We sum all of these radiance for all the samples and multiple by (2 * pi) since its in a hemisphere and then take the average by dividing by num_samples.

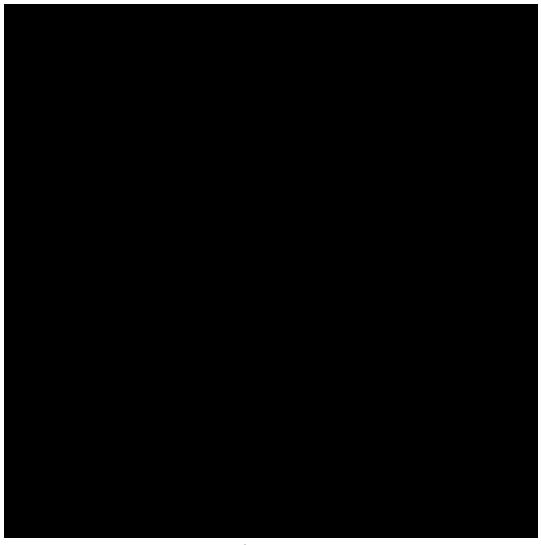
Importance: For the importance lighting, we have two loops: one loop to go through all the light sources and then another loop to loop through num_samples time similar to the hemisphere case. This num_samples is dependent on whether or not the light source is a point source. If the light is a point source, then we only have to sample one time since, the samples over the area all the same, so this can save us some computing power. If the light isn't a point source, then we have to sample the area light. We can sample using the sample_L function. We only want to take the light emitted from this source if it isn't behind another object blocking the ray. We can do this by finding the z value of the direction to tell. This sample returns us a wi vector that is in world space, but we need the object space so we can transform it into object space by multipling w2o by wi. To check if the sample is behind another object, we can just see whether or not the z value of the direction is greater than 0. This greatly accelerates our run time since we only need to worry about cases where there isn't anything blocking the ray. If this is the case, then we can calculate the ray and check for intersection. We can create the ray using hit_p + (EPS_F * wi) as the origin and wi as the direction. We also set the max and min of the ray to avoid intersecting with the light. We then have to test if this ray intersects with a primitive. If it does intersect, then we use the importance sampled monte carlo estimator formula to calculate the radiance being reflected. We sum all these radiance and for each of the light sources we add the radiance divided by the num_samples to get an average.

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

Uniform Hemisphere Sampling

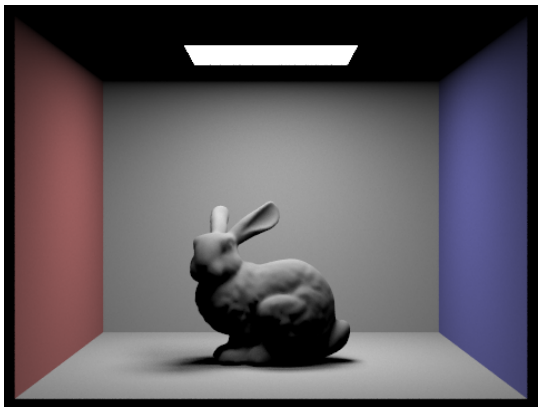


bunny

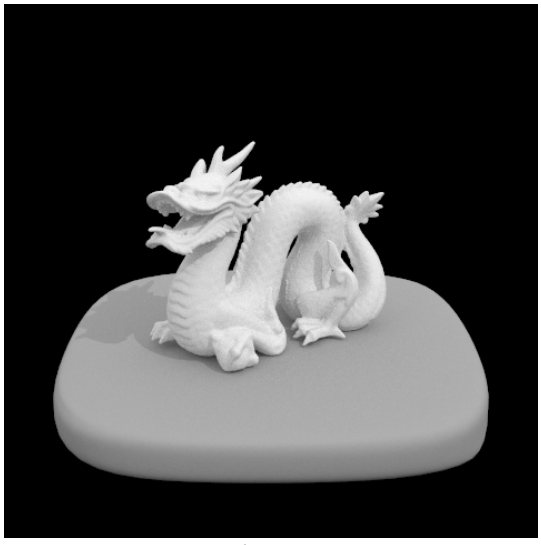


dragon

Light Sampling

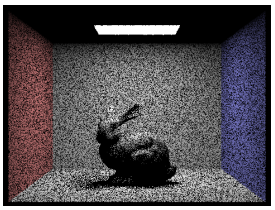


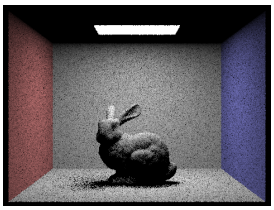
bunny

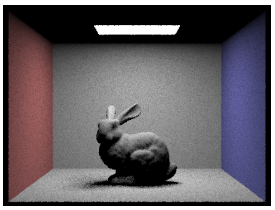


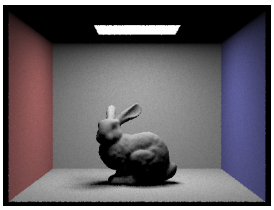
dragon

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 bunny


4 Light Rays bunny


16 Light Rays bunny


64 Light Rays bunny

The more light rays we use, the higher quality and clearer the images will become. This is because more rays will allow us to filter the noise better. More rays means more more information being calculated and a better grasp of what the image is showing. This is apparent in how the soft shadows are represented in the images. At 1 light ray, it is hard to determine what is a soft shadow and what is shown by the noise. The contrast between the soft shadow and noise is more apparent when there are 16 light rays.

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

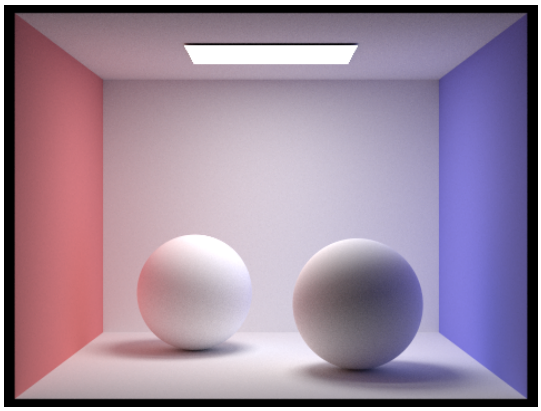
The uniform hemisphere sampling is going to render scenes with darker, more grainy images. This is caused by the noise from rays since we are sampling all over the whole area. This means that there is the possibility that some rays will not be hitting the light source at all and add zero to the radiance, leading to a darker image. The lighting sampling on the other hand renders a cleaner and crisp image. The lighting sampling only takes rays that intercept with the light source so it is guaranteed that these rays will contribute radiance to the overall light emitted.

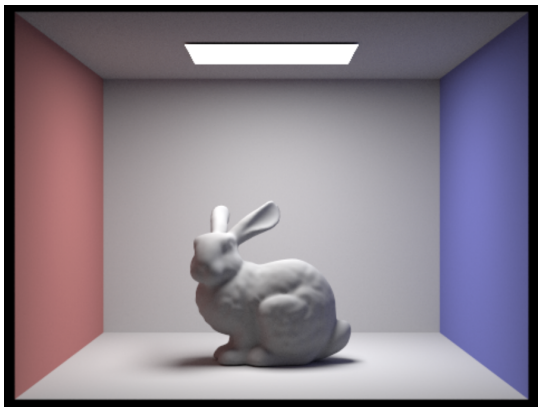
Part 4: Global Illumination (20 Points)

Walk through your implementation of the indirect lighting function.

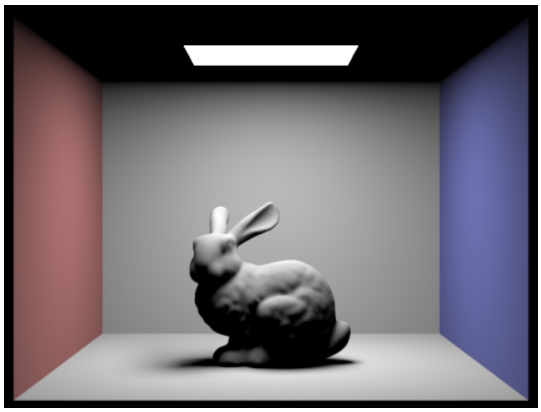
There are a couple of things that we need to check. We first need to check the depth of the ray. If the depth is zero, then we know that the ray has already gone through the maximum number of bounces it can take so we just return L_out which is the 0 vector. Next we have to figure out when to terminate randomly since we are integrating over all paths of all length which can span infinitly. We use the russian roulette aglorithm to determine this. We also have to take into account that if the max number of rays > 1, then indirect illumination is turned on and the russian roulette outcome won't matter. To end the recursive algorithm, with 0.3 probability and making sure that ray depth <= 1, we calculate the one bounce radiance and return it. If we don't end, then we do similar things as the other tasks. We take a sample_f to obtain the evalatuation of the BSDF at wo, wi. We then create the new ray. We want the ray to be in the world space so we need to transform the input direction from the sample into world space. We can create the ray using hit_p as the origin and the transformed wi as the direction. We also set EPS_F for the ray's min and subtract the depth of the ray by 1 since we are computing another bounce. We then test to see if the ray intersects with a primitive. If it does intersect, then we calculate the radiance emitted from this bounce. We use the same equation as the one bounce but recursivly call at_least_one_bounce_radiance as the evalatuation of the BSDF. We also have to normalize by the continuation probablity. For our output, we sum the past radiance from bounces and add to this radiance we calculated.

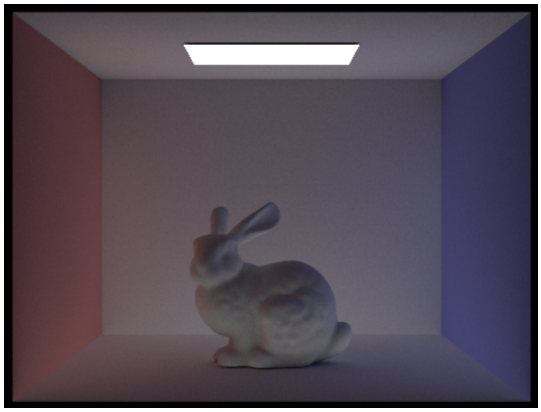
Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.


spheres


bunny

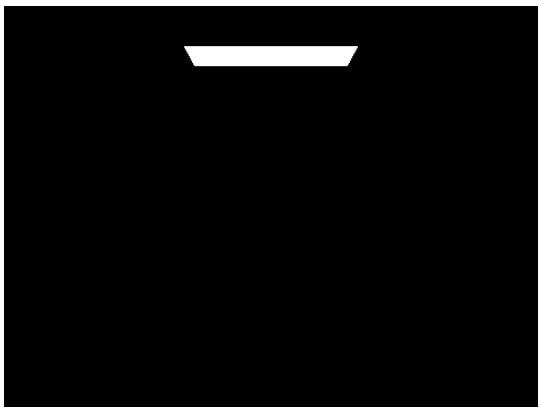
Pick one scene and compare rendered views first with only direct illumination, then only indirect illumination. Use 1024 samples per pixel. (You will have to edit PathTracer::at_least_one_bounce_radiance(...) in your code to generate these views.)

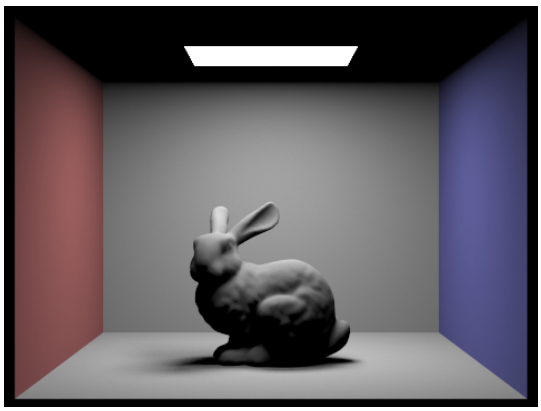

bunny direct illumination

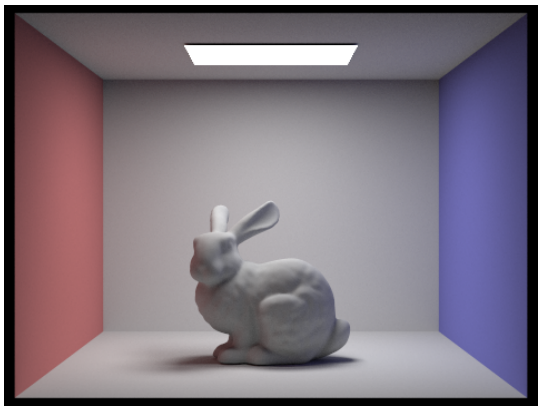

bunny indirect illumination

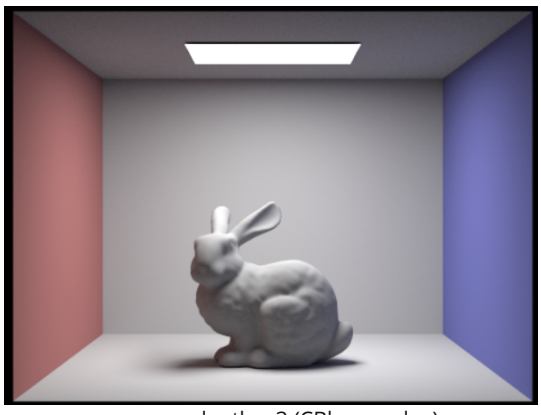
The direct illumination has much more contrast compared to the indirect illumination. This is because the scene is only capturing light that is directly coming from the light source, so the only parts that have the light are spots that are right under the light source. This makes it more like an all or nothing for the lighting so the parts are either all lit up or dark. The indirect illumination on the other hand is a much moodier scene. This scene captures the light that is reflected and bounced around in the scene, so that is why it is not as bright of a scene since the light is being reflected and loses its full value as the ray bounces around. There is also difference in how the shadows are depicted.

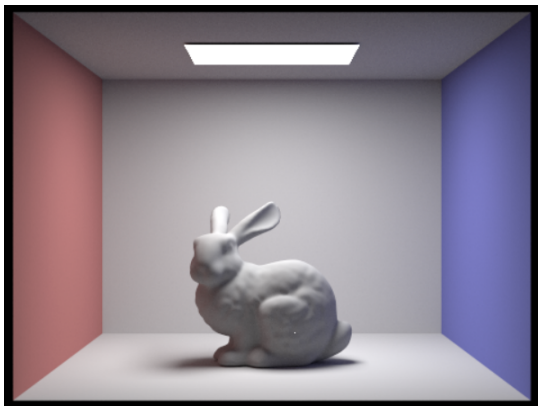
For CBbunny.dae, compare rendered views with max_ray_depth set to 0, 1, 2, 3, and 100 (the -m flag). Use 1024 samples per pixel.


max_ray_depth = 0 (CBbunny.dae)


max_ray_depth = 1 (CBbunny.dae)

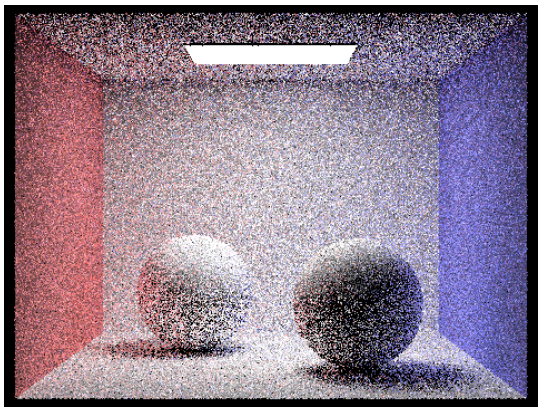

max_ray_depth = 2 (CBbunny.dae)

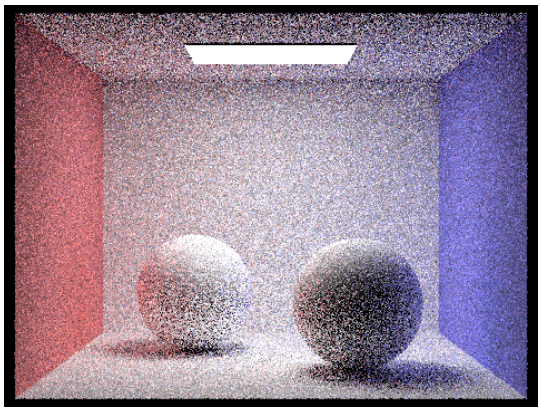

max_ray_depth = 3 (CBbunny.dae)

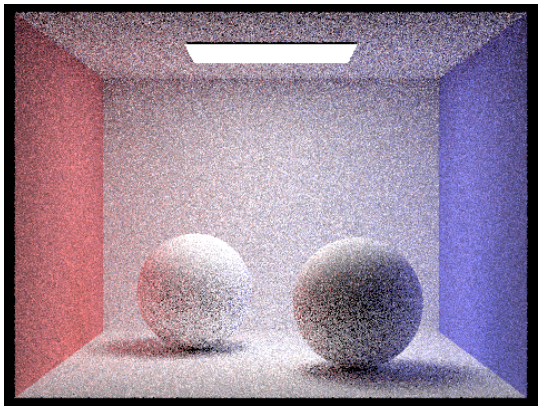

max_ray_depth = 100 (CBbunny.dae)

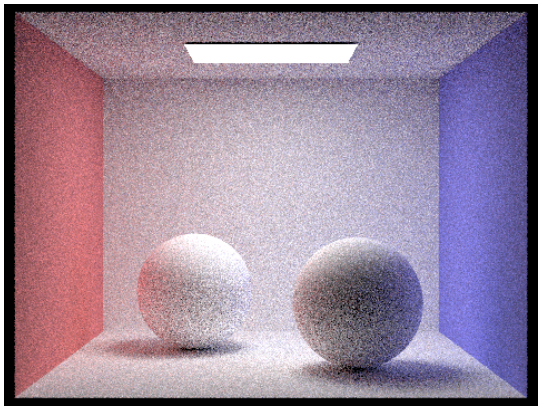
YOUR EXPLANATION GOES HERE

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.

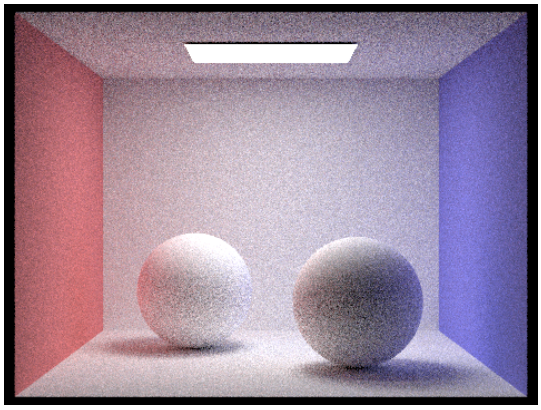

1 sample per pixel sphere


2 samples per pixel sphere

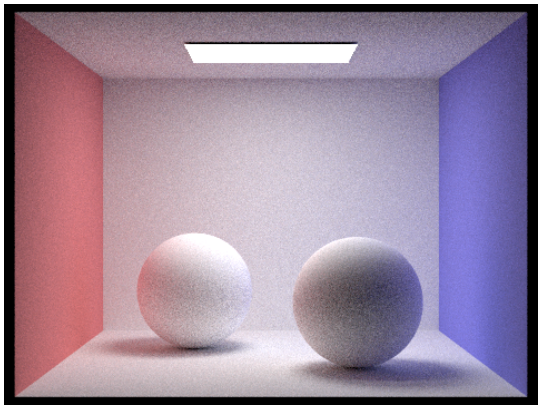




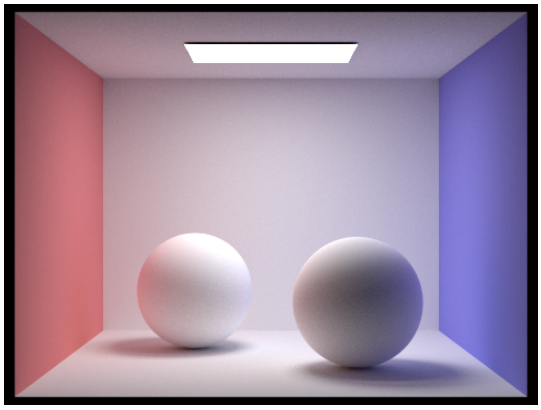
4 samples per pixel sphere



8 samples per pixel sphere



16 samples per pixel sphere



64 samples per pixel sphere

1024 samples per pixel sphere

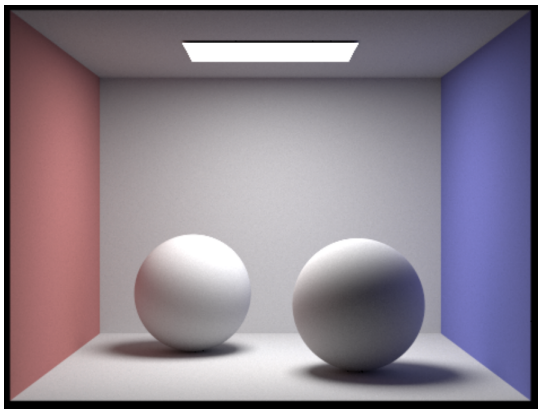
The more samples per pixel we take, the clearer the images become. This is because more samples means more information, so there is less noise that can come from our sampels. With less samples, the images become very grainy and hard to tell what is going on. This becomes less of an issue around the 64 samples per pixel mark, which has a clearer and less blocked image.

Part 5: Adaptive Sampling (20 Points)

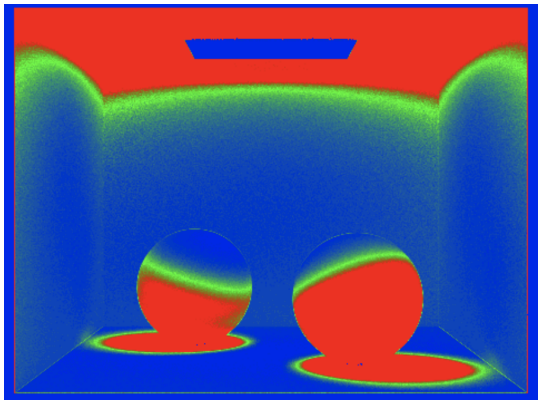
Explain adaptive sampling. Walk through your implementation of the adaptive sampling.

Adaptive sampling basically allows us to concentrate more work into samples that need more information rather than more simple ones. This optimization allows us to render more efficiently and have less noise in our images. In addition to our regular ray tracing, we replaced and added more work to be able to implement adaptive sampling. Using the est_radiance_global_illumination, we calculated the illuminance coming from the ray, which will be used to find our mean and variance variables. Through out our samples, we keep track of an s1 and s2 sum. The s1 sum adds the illum, while the s2 sum adds the illum squared. We calculate the mean by using s1 / i (where i is the number of samples taken so far) and variance square as (1 / (i - 1)) * (s2 - (s1 * s1 / i)). To maximize efficiency, we don't check a pixel's converence everytime we take a new sample. Instead we, only check when the number of samples we have taken so far is a factor of the samplesPerBatch. To check, we find the I value which is 1.96 * variance / sqrt(i). We use the I value to check whether the pixel converged. If I <= maxTolerance * mean, then we know that the pixel has converged so we can stop tracing rays through the pixel. Otherwise, we continue ray tracing. Lastly, we normalize our value by i and place i in the sampleCountBuffer since we only take i samples.

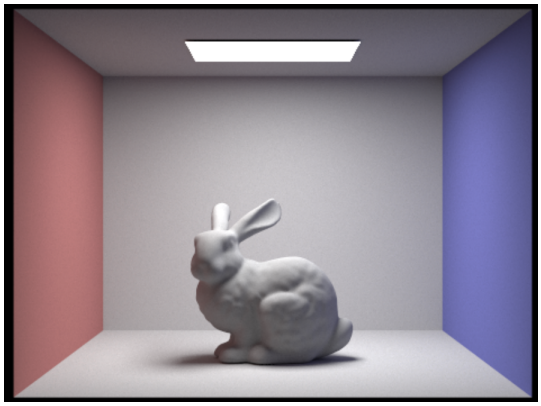
Pick two scenes and render them with at least 2048 samples per pixel. Show a good sampling rate image with clearly visible differences in sampling rate over various regions and pixels. Include both your sample rate image, which shows your how your adaptive sampling changes depending on which part of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



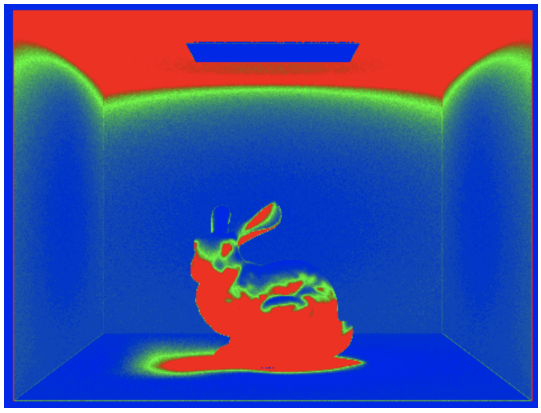
Rendered image sphere



Sample rate image sphere



Rendered image bunny



Sample rate image bunny