Part 1

- Walk through the ray generation and primitive intersection parts of the rendering pipeline.
- Explain the triangle intersection algorithm you implemented in your own words.
- Show images with normal shading for a few small *.dae* files.

- For the ray generation, we transform the point `(x,y)` in image space to camera space first. This position in camera space is calculated using the Field of View (FOV) angle. Then, it is converted to the world space. Also, we want to do the pixel sampling in one image. Within one pixel area, we sample multiple rays (inspired by the Monte Carlo Method). This is implemented in the `raytrace_pixel` and updates the pixel radiance with an average buffer. Then for primitive intersections, we mainly divide it into two parts: the intersection_check and intersection_calculation. For the intersection_check, we check if the `t` in `r(t) = o _ td` is larger than 0 or not. Only when the solved `t` is larger than 0 can it indicate the ray interacts with the triangle/sphere plane. If it intersects, use the method in slides (Moller Trumbore Algorithm, etc.) to get the intersection point.
- In our implementation of the ray-triangle intersection, we use the Möller–Trumbore algorithm. This algorithm is efficient because it can determine the intersection point without having to compute the actual geometric intersection which could be more computationally expensive. We start by finding two edge vectors of the triangle: `e1`, which is from the first point of the triangle `p1` to the second point `p2`, and `e2`, which is from `p1` to the third point `p3`. These vectors help define the plane in which the triangle lies.

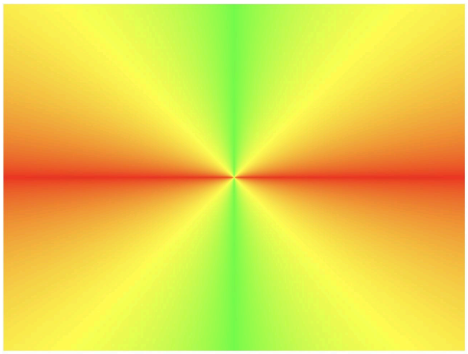
  We then take the cross product of the ray's direction `r.d` and the second edge vector `e2` to get a vector `h`. If the dot product of `h` with the first edge vector `e1` is near zero, we know that the ray is parallel to the triangle's plane and cannot intersect it within any finite distance.

  We calculate a scalar `f` that's the inverse of `a` (dot product of `e1` and `h`). This scalar will be used to transform certain values into barycentric coordinates, which are a coordinate system on the triangle's plane. We compute `u`, which represents the barycentric coordinate for one vertex, by multiplying `f` with the dot product of vector `s` (from `p1` to the ray origin `r.o`) and vector `h`. If `u` is not between 0 and 1, the intersection point is not within the triangle.

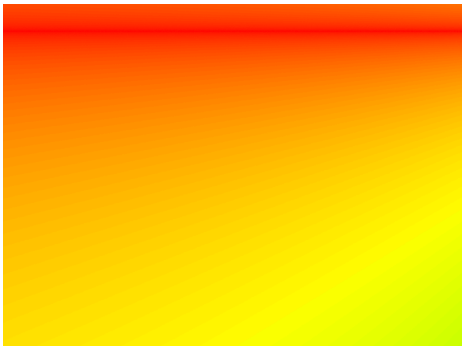
Using the cross product of `s` and `e1`, we find a vector `q` and use it to compute `v`, the barycentric coordinate for the second vertex. Like `u`, `v` must also be between 0 and 1 for the intersection to be within the triangle. Furthermore, the sum of `u` and `v` must not exceed 1; otherwise, the intersection is outside the triangle.

Finally, we calculate the distance `t` from the ray origin to the intersection point along the ray's direction, using `f`, the dot product of `e2` and `q`. If `t` falls within the valid range between `r.min_t` and `r.max_t`, the intersection is valid, and the ray does hit the triangle.

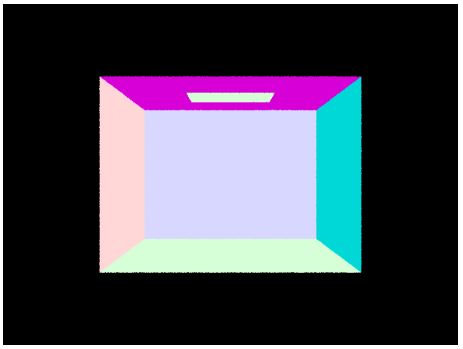
- `CBempty.png`



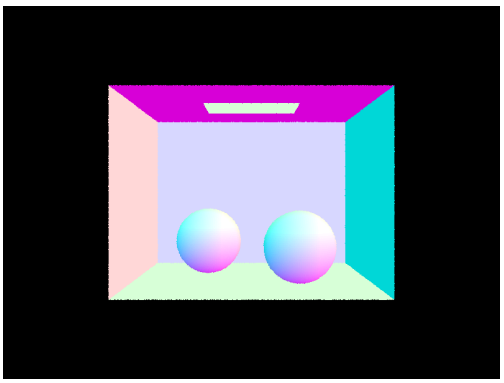
- `banana.png`



`CBempty.png`



- `CBspheres.png`

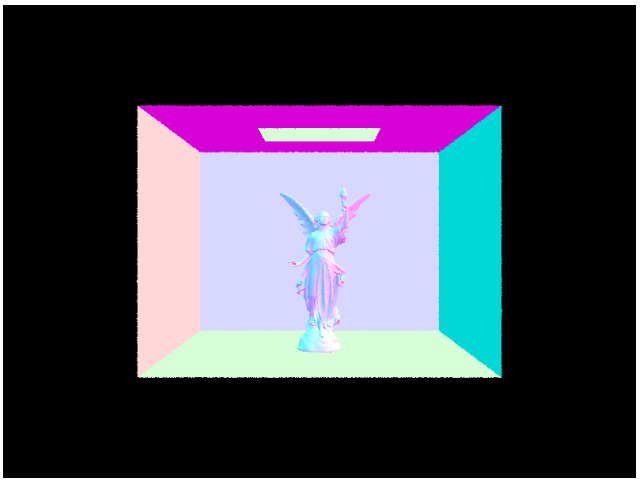


Part 2

- Walk through your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.
- Show images with normal shading for a few large *.dae* files that you can only render with BVH acceleration.
- Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.

- To construct such a BVH tree, we know that the base case is when all the remaining primitives' count is less than the `max_leaf_size`. Otherwise, we need to find the centroid of all the BBox. We choose the `max_extent` of one dimension (x,y, or z) and use this value and dimension as the splitting threshold. That is, all the primitives whose BBox in such dimension value is less than the threshold are classified into the left subtree, and the remaining ones are classified in the right subtree (Splitting heuristic).This heuristic is often chosen because it tends to create well-balanced trees where each subtree has a roughly equal number of primitives, which can greatly improve the efficiency of the ray-tracing process by minimizing the number of unnecessary ray-primitive intersection tests. Having found the splitting tree, we recurse left and right subtrees until the base case is hit.
- `maxplanck.png`

- CBlucy.png



- Comparision

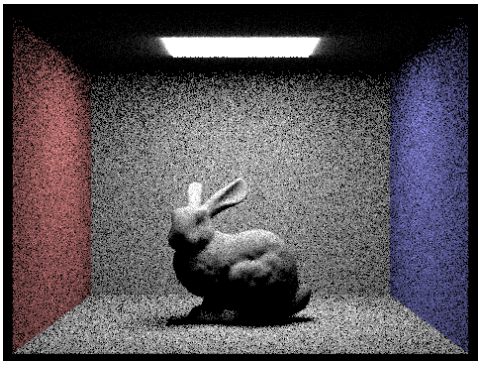
```
184\hw\hw3-pathtracer-sp24-6\out\build\x64-Release>pathtracer -t 8
-r 800 600 -f cow.png ../../../dae/meshedit/cow.dae
[PathTracer] Input scene file: ../../../dae/meshedit/cow.dae
[PathTracer] Rendering using 8 threads
[PathTracer] Collecting primitives... Done! (0.0018 sec)
[PathTracer] Building BVH from 5856 primitives... Done! (0.0027
sec)
[PathTracer] Rendering... 100%! (0.0437s)
[PathTracer] BVH traced 413363 rays.
[PathTracer] Average speed 9.4622 million rays per second.
[PathTracer] Averaged 0.000000 intersection tests per ray.
[PathTracer] Saving to file: cow.png... Done!
[PathTracer] Job completed.
```

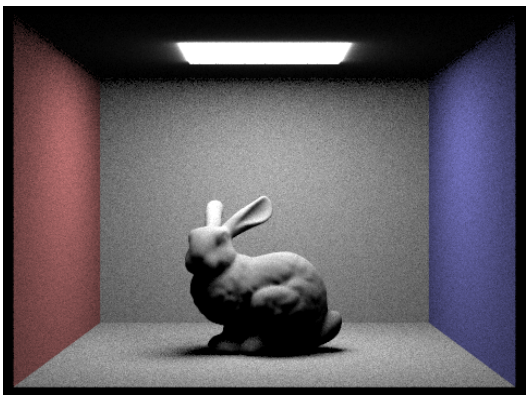
For the cow which needs **`40`** seconds with no acceleration method, now with the implemented acceleration method, here the rendering just takes **`0.0437`** seconds (shown log above). Furthermore, the above `maxplanck.png` and `CBlucy.png` take **`0.0538`** seconds and **`0.0479`** seconds to render, respectively. These results comparison reveals BVH acceleration greatly reduces the rendering time.

Part 3

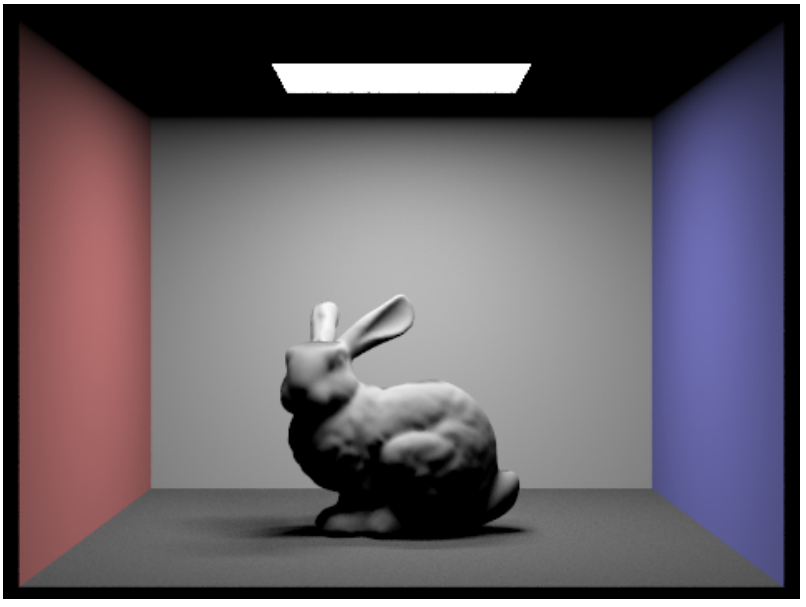
- Walk through both implementations of the direct lighting function.
- Show some images rendered with both implementations of the direct lighting function.
- Focus on one particular scene with at least one area light and compare the noise levels in soft shadows when rendering with 1, 4, 16, and 64 light rays (the `-l` flag) and with 1 sample per pixel (the `-s` flag) using (important) light sampling, **not** uniform hemisphere sampling.
- Compare the results between uniform hemisphere sampling and lighting sampling in a one-paragraph analysis.

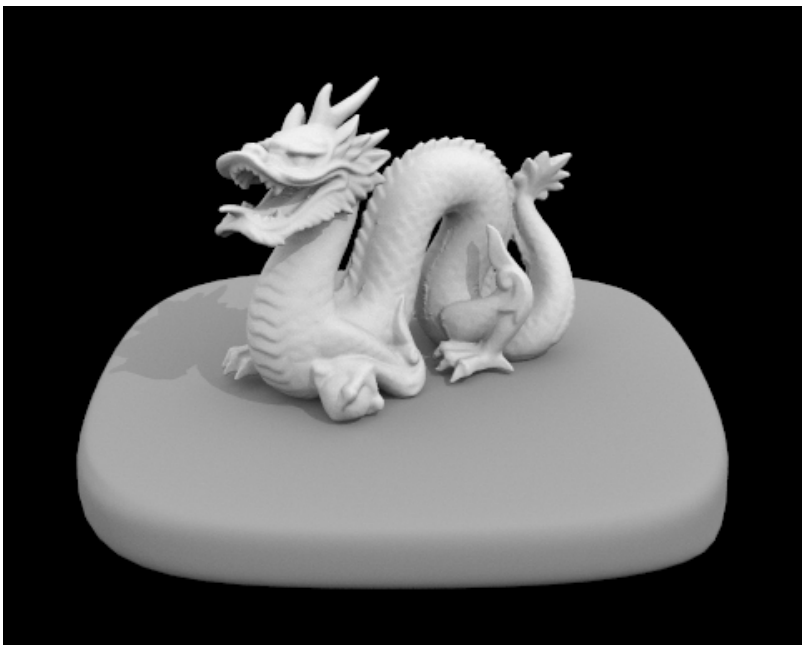
- Uniform hemisphere sampling: First, a coordinate system is created at the point of intersection (`isect`) where the surface normal (`isect.n`) aligns with the Z-axis of this local coordinate system. This is done using the `make_coord_space` function. The hit point (hit_p) is calculated using the ray origin (r.o) and direction (r.d), scaled by the intersection distance (isect.t). The outgoing direction (w_out) is the negative ray direction transformed into the local coordinate system, pointing toward the source of the ray. The core of this function lies in the loop that samples directions over the hemisphere. For each sample, we sample a direction, transform the direction to world space, offset the hit point, and then trace a new ray. If the new ray intersects with a light source, we calculates the contribution of that light to the intersection point





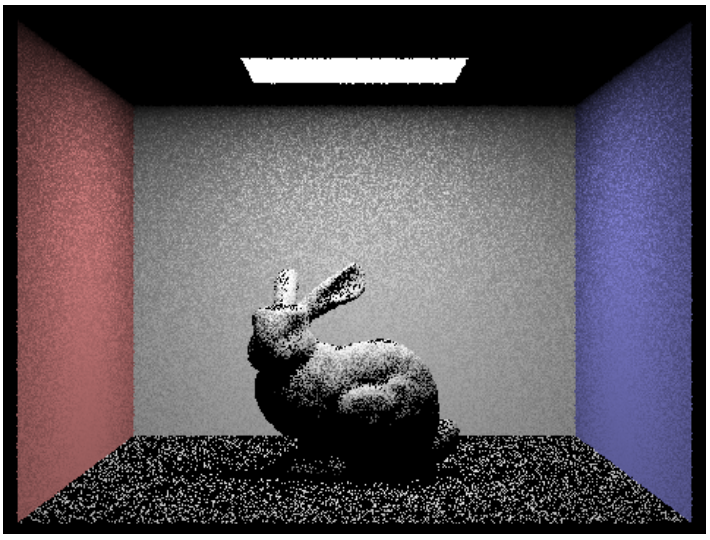
- light importance sampling: Just like in uniform sampling, a local coordinate system is set up at the intersection point, aligning the surface normal with the local Z-axis. Then we iterate over each light source in the scene. For each light, it aims to sample points or directions that are likely to contribute to the light at the intersection point. For each light, we sample a direction (`wi`) towards the light from the intersection point. This sampling process also calculates the distance to the light (`distToLight`) and a probability density function (`pdf`) for the sampled direction. We use a shadow ray that is cast from the intersection point in the sampled direction to check if there are any occlusions between the point and the light source. If the shadow ray does not intersect any other objects, the contribution of this light sample to the overall illumination (`L_out`) is calculated.



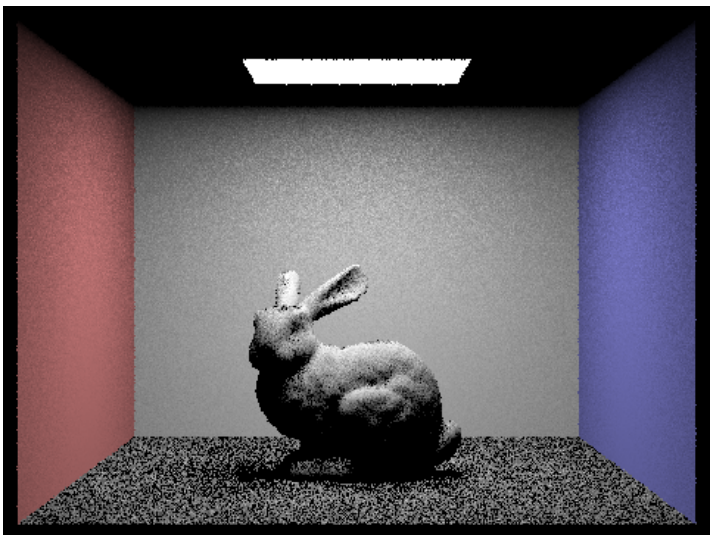


- Compare both sampling schemes: Uniform hemisphere sampling distributes rays evenly across the hemisphere, which does not prioritize light sources, leading to a higher number of unnecessary calculations and potentially more noise, especially with fewer samples. As the number of samples increases, however, the uniform sampling method's image will converge to a result similar to that of importance sampling, although it generally takes longer to reach the same level of clarity and noise reduction. In contrast, importance sampling strategically directs sampling toward the light sources, resulting in a more efficient rendering process. It concentrates computational resources where they are most needed, yielding a clearer image with less noise in a shorter amount of time. Thus, with equal or similar rendering time, importance sampling is likely to produce a superior image with better-defined shadows and lighting details compared to uniform sampling.

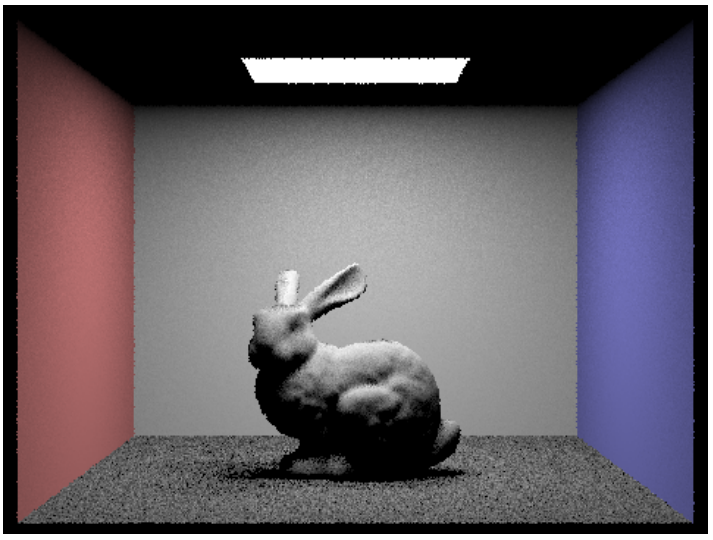
./pathtracer -t 8 -s 1 -l 1 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae



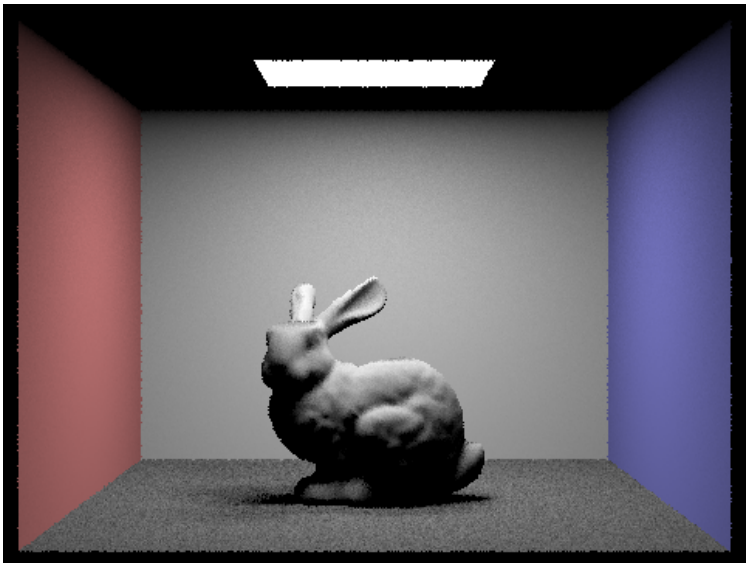
./pathtracer -t 8 -s 1 -l 4 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 16 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae



./pathtracer -t 8 -s 1 -l 64 -m 6 -f bunny_64_32.png -r 480 360 ../dae/sky/CBbunny.dae



In the rendering results above, as the number of light rays increases from 1 to 64 using importance light sampling, there is a clear trend of decreasing noise in the soft shadows cast by the bunny.

- With just 1 light ray, the image shows significant noise, manifesting as grainy, scattered specks of light and dark, which obscures the detail in the shadows.
- As the number of rays increases to 4, the noise in the shadows is reduced, but the image still exhibits a considerable amount of graininess.
- With 16 light rays, the shadows cast by the bunny begin to show more definition, and the noise is further reduced. The shadows start to blend more smoothly with the illuminated areas, although some noise is still apparent.
- Finally, with 64 light rays, the soft shadows have a much cleaner appearance, with significantly less noise compared to the result with 1 light ray. The shadow edges and the transition areas between light and dark are smoother and more natural-looking.

Part 4

- Walk through your implementation of the indirect lighting function.
- Show some images rendered with global (direct and indirect) illumination. Use 1024 samples per pixel.
- 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.)
- For *CBbunny.dae*, render the mth bounce of light with `max_ray_depth` set to 0, 1, 2, 3, 4, and 5 (the `-m` flag), and `isAccumBounces=false`. Explain in your writeup what you see for the 2nd and 3rd bounce of light, and how it

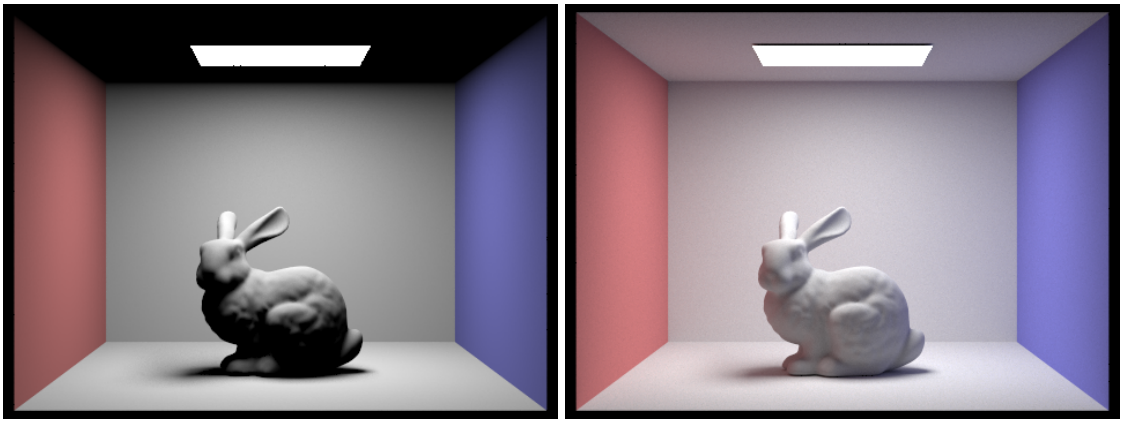
contributes to the quality of the rendered image compared to rasterization. Use 1024 samples per pixel.

- For *CBbunny.dae*, compare rendered views with `max_ray_depth` set to 0, 1, 2, 3, 4, and 5(the `-m` flag). Use 1024 samples per pixel.
- For *CBbunny.dae*, output the Russian Roulette rendering with `max_ray_depth` set to 0, 1, 2, 3, 4, and 100(the `-m` flag). Use 1024 samples per pixel.
- 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.
- You will probably want to use the instructional machines for the above renders in order to not burn up your own computer for hours.

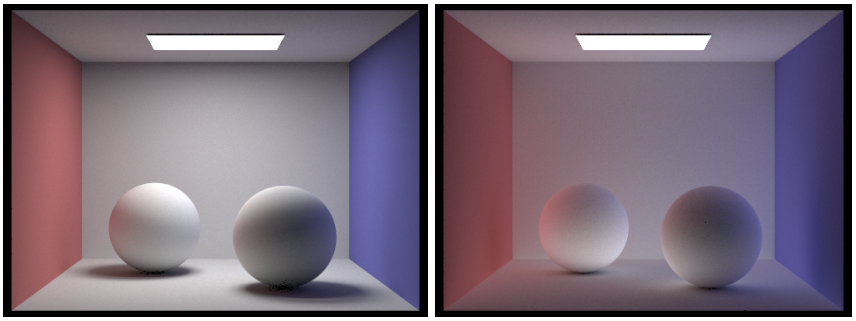
- Initially, our function checks if the global variable `isAccumBounces` is set to zero and if `max_ray_depth` equals one. If so, it returns the result of `one_bounce_radiance(r, isect)`, which computes direct lighting using the functions implemented in part 3. This step handles the case where only direct lighting is considered without any indirect illumination.
- Then we accumulate direct lighting. If `isAccumBounces` is not zero, the function adds the result of `one_bounce_radiance(r, isect)` to `L_out`, starting the accumulation of light contributions.
- To sample Indirect Lighting, our function then proceeds to sample a new direction based on the surface's BSDF by calling `isect.bsdf->sample_f`. It converts the outgoing direction `w_out` from world to local coordinates, samples a new incoming direction `w_in`, and calculates the corresponding PDF. This step is crucial for simulating the scattering of light off surfaces, which is the essence of global illumination.
- With the sampled direction, the function constructs a new ray and checks for intersections with the scene. If an intersection is found, it recursively calls itself with the new ray and intersection. This recursive process allows the simulation of light bouncing off multiple surfaces, accumulating contributions from indirect lighting.
- For each bounce, the function calculates the contribution of the current path to the final color. It multiplies the BSDF's value by the dot product of the incoming direction and the surface normal, divided by the PDF of the sampled direction. This calculation is based on the rendering equation, accounting for the BSDF's role in determining the proportion of light that is scattered in the incoming direction.
- The recursion depth is controlled by `r.depth` and `max_ray_depth`. If the maximum depth is reached, the function may terminate early or continue accumulating light based on the `isAccumBounces` flag. This mechanism prevents infinite recursion and allows for control over the trade-off between accuracy and computation time.

- After all recursive calls complete, the accumulated light contributions from direct and indirect lighting are added to `L_out`, which is returned as the final color contribution from the current path.

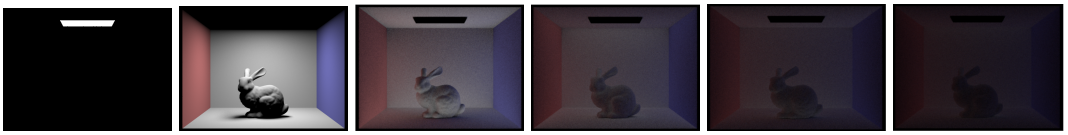
- Direct Illumination v.s. Indirect Illumination



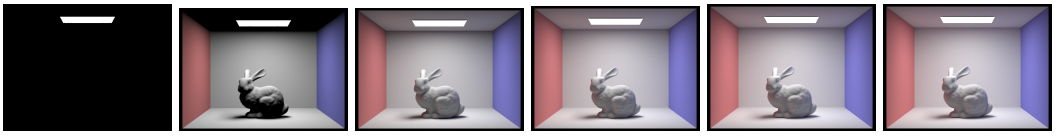
- Only direct illumination v.s. Only Indirect Illumination



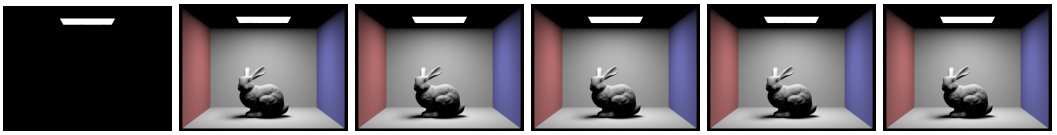
- isAccumBounces = False (m from 0 to 5)



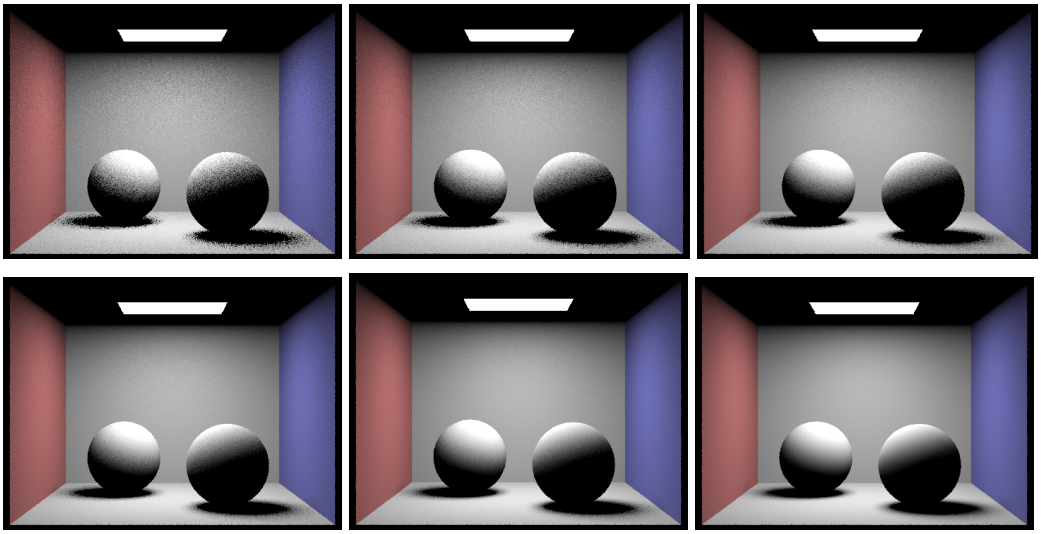
- isAccumBounces = True (m from 0 to 5) (*Rendered without Russian Roulette*)

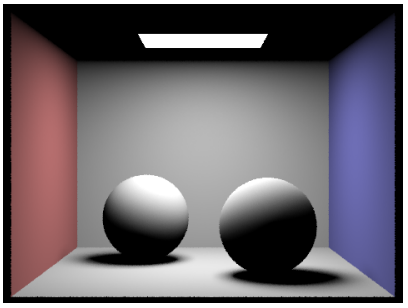


- Russian Roulette (m = 0, 1, 2, 3, 4, and 100) Note that the size of each figure differs, thus proving there must be slight difference between any of these two figures with different size.



- rendered views with various sample-per-pixel rates (1, 2, 4, 8, 16, 64, and 1024)

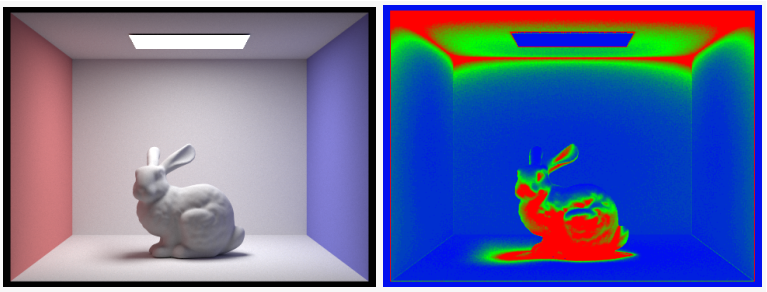




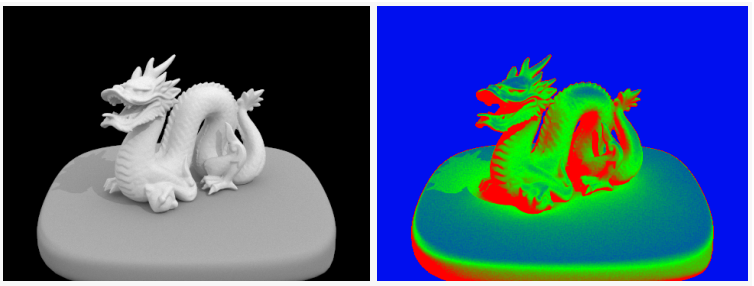
Part 5

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

- Adaptive sampling dynamically adjusts the number of light ray samples per pixel in an image to reduce huge resource consumption. In our implementation, each pixel starts with an initial set of samples, and additional samples are taken in batches. After each batch, we calculate the pixel's current color (mean radiance) and its variability (standard deviation of illuminance). Using these statistics, we compute a convergence measure (I) to decide if the pixel's color has sufficiently converged to its true value. If this measure is below a predefined tolerance (indicating high confidence in the color accuracy), we stop sampling; otherwise, we continue in batches until either convergence is achieved or a maximum sample count (ns_aa) is reached. This method ensures that pixels requiring more detail due to complex lighting conditions receive more samples, whereas simpler areas use fewer samples, optimizing overall rendering time and computational resources.
- Two scenes:
    1. `./pathtracer -t 8 -s 2048 -a 64 0.05 -l 1 -m 5 -r 480 360 -f bunny.png ../dae/sky/CBbunny.dae`



    2. `pathtracer -t 8 -s 2048 -a 64 0.05 -l 1 -m 5 -r 480 360 -f dragon.png ../../../dae/sky/dragon.dae`



From the two groups, it is obviously observed that the background is sampled with less ray and they appear to be blue in the right rate figure. The background gets a lower sample rate. However, the regions with complex lighting or shadow edges have relatively more sample rates, indicating that more computational effort was focused there, which reveals red in the two groups of images. The purpose of the two groups is to demonstrate the effectiveness of adaptive sampling in dynamically balancing and allocating resources to achieve computational efficiency.