CS148 Final Project Write Up

Name : Zhejin Huang, Pramod Srinivasan
SUNet ID : zhejinh, pramods

Scene Description:

We chose the famous Cornell Box scene. The basic environment consists of a green right wall, red left wall and white back wall. Two boxes were placed and two spheres were placed on top of the boxes. In order to incorporate geometric modeling, we added a succulent Plant Pot which was created and textured from scratch.

Lighting

- An area light on top of the scene.

Technical Contributions

Photon Mapping

Environment

The development of the photon map is in Blender 2.90.0 built-in Python. And we use Sublime to write classes and functions to be imported and called by the scripts in Blender.

Photon and Photon Map

We are implementing a Photon class which includes location and direction fields. We also include a depth field to record how many reflections/transmissions has a photon been going through. And an id to identify each photon when it is copied into a photon map. Photon ID grows from 0 and increases by 1 for each incoming photon. Photon ID is identical within each photon map.

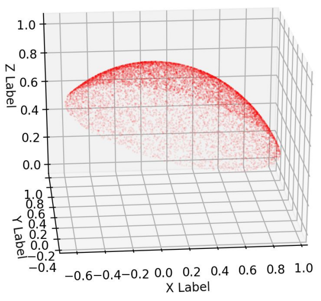
For the Photon Map, we use a map (dictionary) to store each photon's ID and the corresponding photon object. Then we use a KD-Tree to store all the photon locations to increase the performance of nearest neighbor searching in rendering. We experimented with using Scipy's KDTree and MathUtils' KDTree. Scipy's KDTree only supports storing an n-Dimension location, so we computed a hash value for each location and used that as a photon ID to correlate a location in KDTree with the corresponding photon object. The computing of hash has a negative impact on the map building performance. Also, we've found that MathUtils' KDTree is better optimized for 3D location and is much faster in tree building and searching. Given the above 2 reasons, the overall performance of our photon map implemented on mathutils' KDTree is ~10X faster than our photon map implemented on scipy's KDTree. So we went forward with the mathutils one.

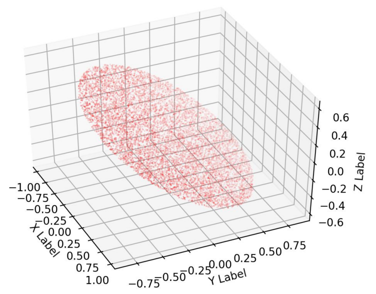
To further boost the efficiency of building a photon map, we added a photon to the map's dictionary when add_photon() is called. And after all the photons are added, we call build_tree() for one time to build and balance the KDTree. To save a photon map, we only save the dictionary; and to load a saved photon map, we load the dictionary and build the KDTree based on that dictionary. To search the nearest neighbors of a location, we use the built-in find_n() and find_range() functions of the KDTree.

Photon Emission

We implemented the photon emission of point light, spot light and area light based on 2 mechanisms: sampling within a disk area and sampling within a cone of a given direction. In order to sample from a disk area, we simply reused our implementation in HW5. Sampling from a cone given a direction is implemented based on the hemisphere sampling technique in HW5, the difference is that we use a number between 0 and 1 to control the range of the cone. 0

represents the line of given direction, 0 - 0.5 represents a cone, 0.5 represents a hemisphere, 1 represents a sphere. The following figures show a sampling of cone direction with the value set to 2 and a sampling from disk.





To implement photon emission from a point light or spot light we initialize each photon location with the light’s location and sample the photon direction from the above cone sampling (value 0-0.5 for spot light and 1 for point light). To implement an area light we sample the photon location using the above disk sampling and photon direction using the above cone sampling, with the value set to 0.5 to represent hemisphere sampling.

Photon Tracing

After emission of each photon, we use the same ray_casting function as HW3 and HW5 to find the intersection of the photon and the scene. If the photon hits any object in the scene then we move on, if it does not, then we ignore that photon. A photon that hits an object can have the following paths. Before proceeding to the following paths, we store a copy of the photon into the photon map:

Photon Absorption

Each photon has a chance to be absorbed, we use a Bernoulli sampling to give a photon a k_a = 0.5 probability of being absorbed. If the photon is absorbed, we return right away without doing anything else.

Photon Diffusion

After the photon hits an object, we use the object’s diffusion intensity (between 0 and 1) of that channel (we are building a photon map for each channel) to be the probability of diffusion. And sample a new direction randomly in the hemisphere of the surface normal. Then trace the photon with the new direction recursively until it meets the max depth (number of diffusion, reflection or transmission) we want.

Photon Reflection and Transmission

Similar logic applies to reflection and transmission. We are using the same function as HW3 to compute the photon direction for reflection and transmission. For a transparent material we use fresnel or reflectivity to determine the ratio of reflection and transmission. After computing the new direction, we trace the new photon recursively until it meets max depth.

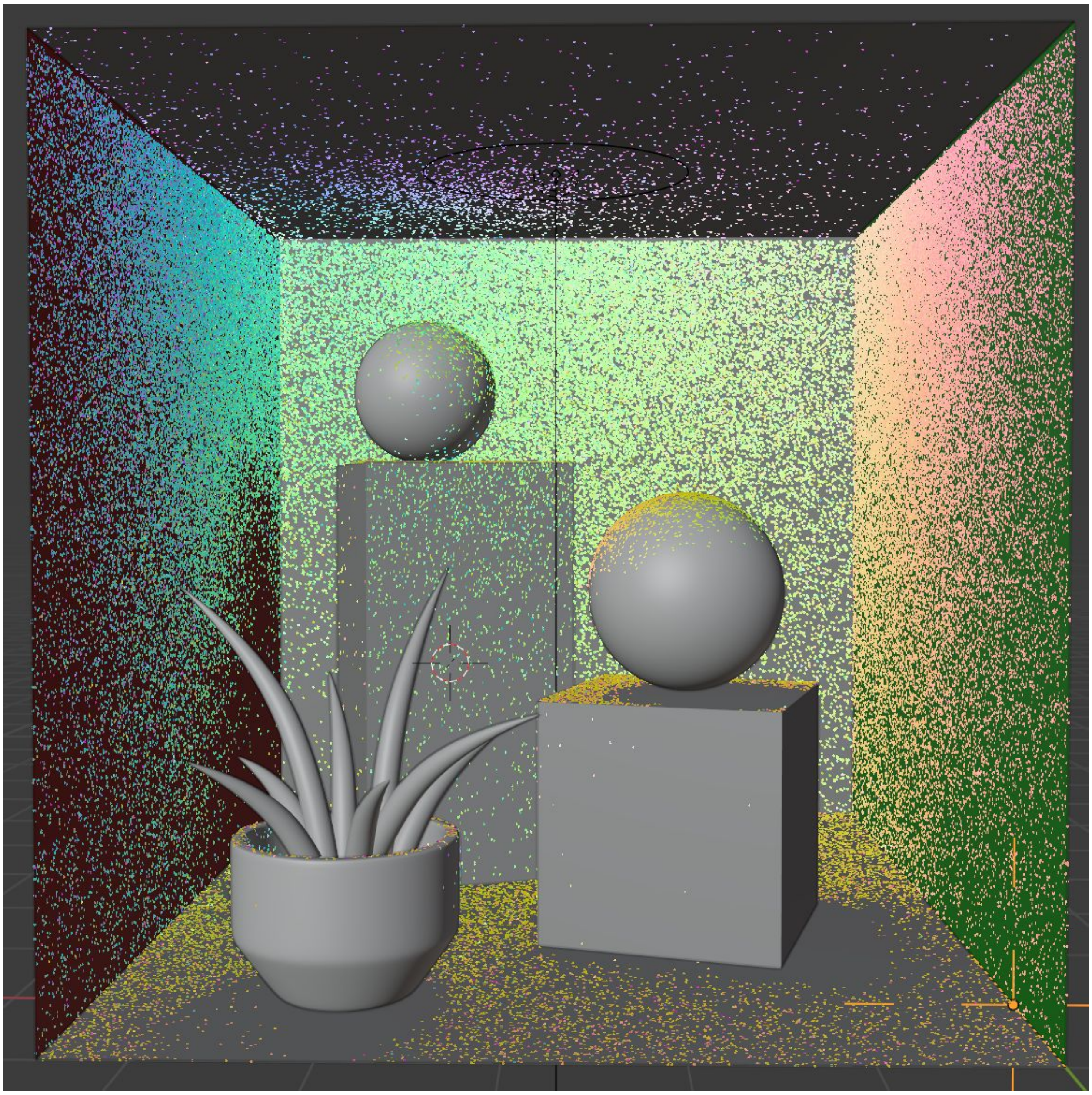
Multi-channel

The 3 photon maps corresponding to red, green and blue photons. Each photon map has different behaviors with respect to diffusion.

Photon Visualization

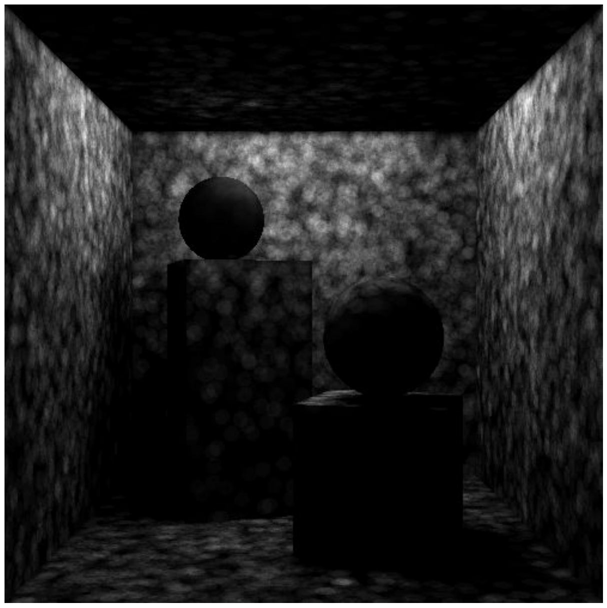
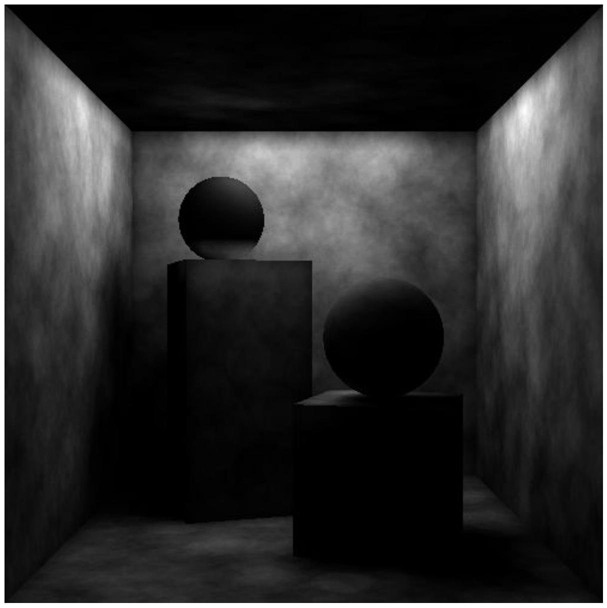
To make debugging easier, we converted a photon map to PLY point cloud to store photons' locations and directions and visualized them in Blender using a point cloud visualizer (in the reference below). Below is a visualization of a red photon map with max depth being 6, the color represents the direction of a photon.

The visualization is important because it gives us a feeling of what the rendering of global illumination looks like before rendering. It can also make it easier to find any unexpected issues. The right side of the small box does not have any red photons, which is expected since the green wall has 0 value for red diffusion. However the left side of the tall box has some but not many red photons on it. This is not expected since the red wall is highly diffusive for red photons. This is something that we want to figure out the next step.

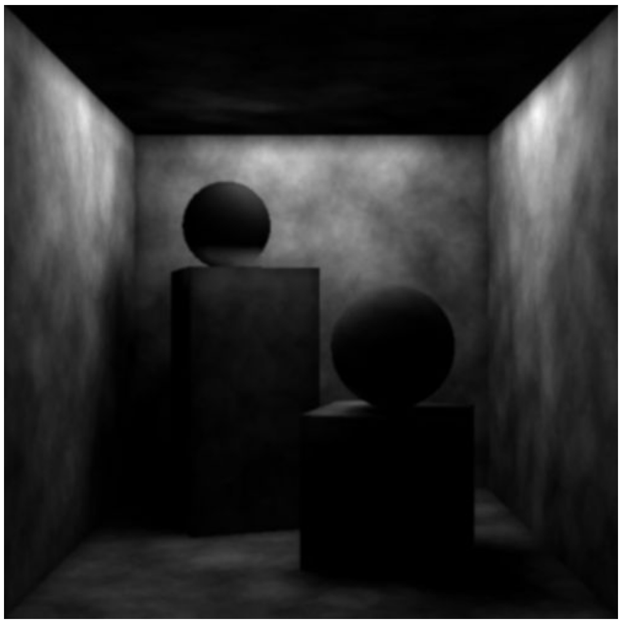


Rendering

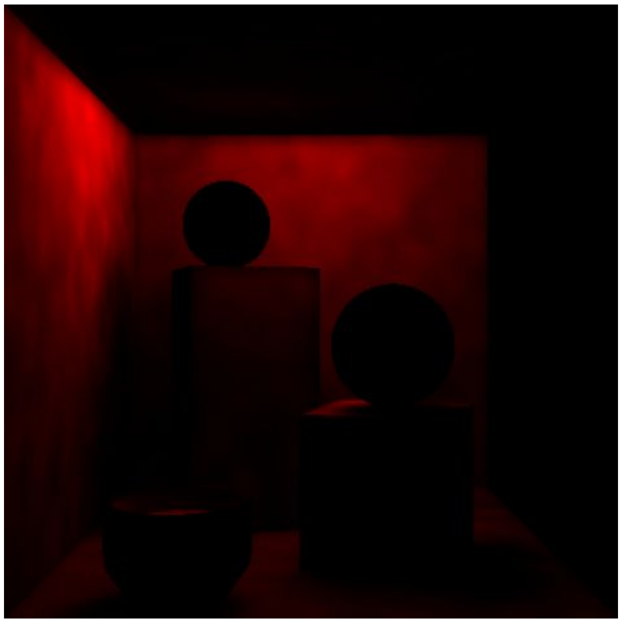
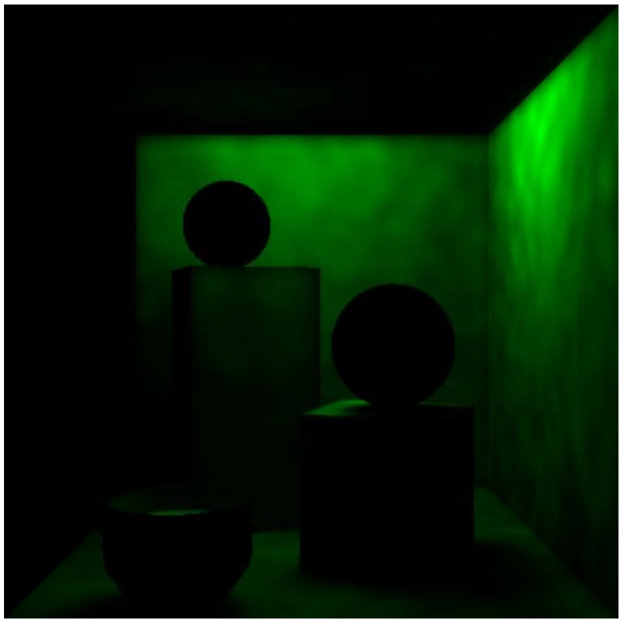
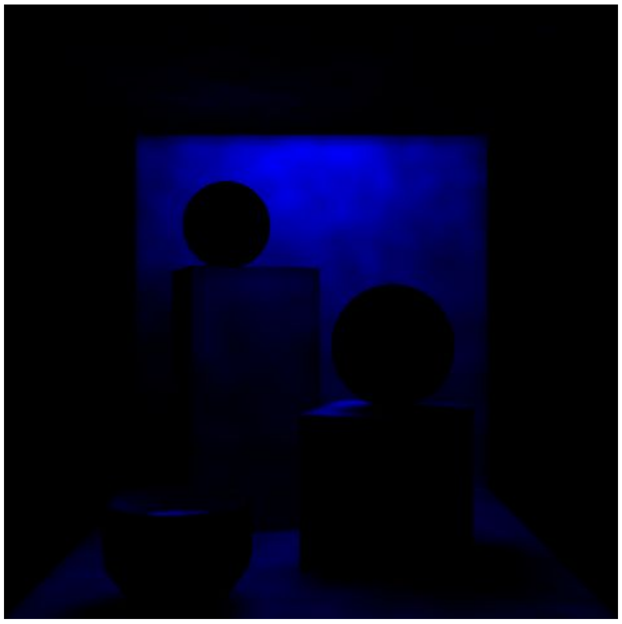
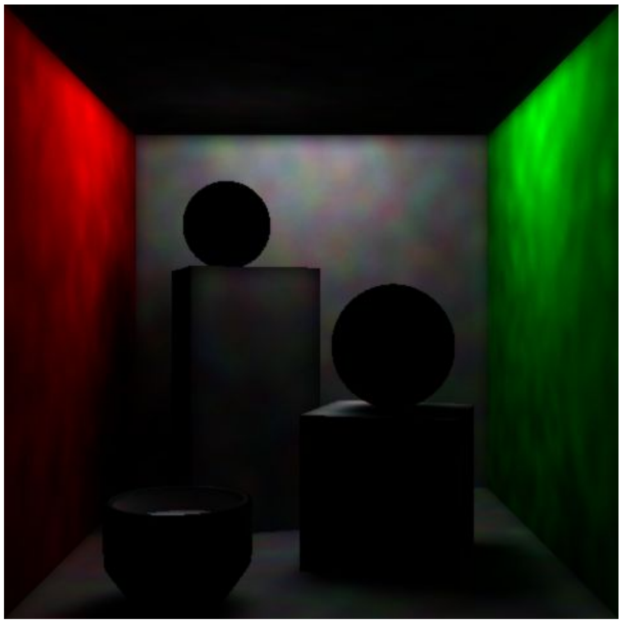
To render the global illumination of the scene, we cast rays from camera to the scene to get hit locations, and search the nearest neighbors for each hit location. For each location we search all the photons within radius r to that location and add the cosine of photon direction and surface normal to the global illumination color at that location's color. The following images compare r = 0.1 and r = 0.3 for single channel rendering. We can see r = 0.3 is much smoother and r = 0.1 is too sharp. In our final image we use r = 0.25 and a gaussian filter to smooth the image.

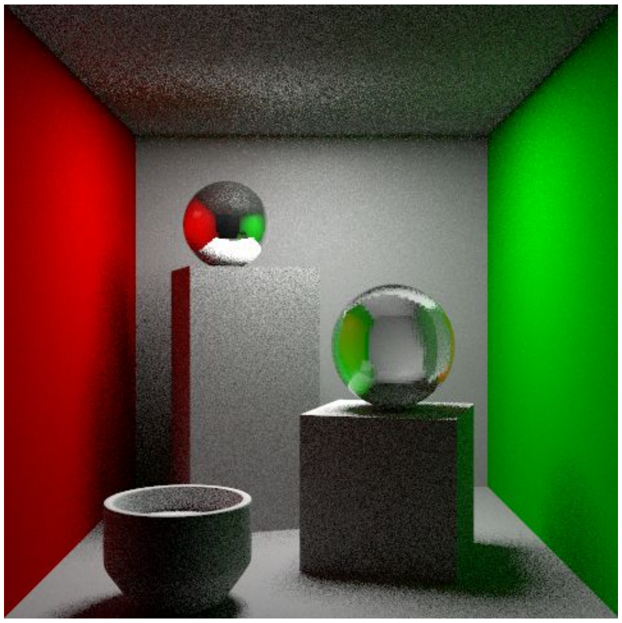
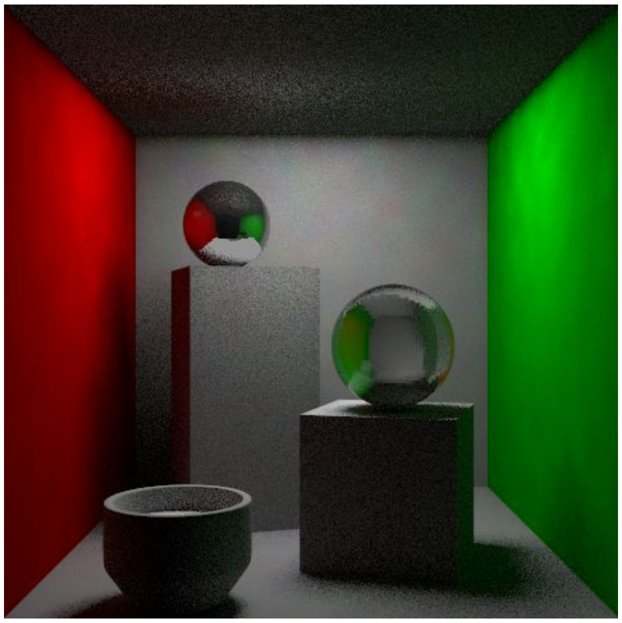
The following image adds a gaussian smoother to the r = 0.3 image.



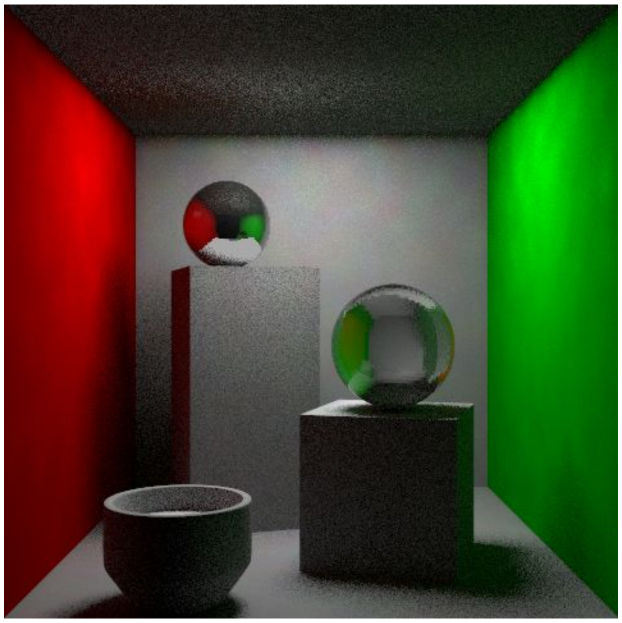
Still we can see that the top of the tall box and the bottom of the ball on that box is incorrectly illuminated due to the searching area 0.3 is pretty large. To solve this problem we make sure to only consider photons which are located above the hit surface and are going towards the hit surface. The rendering time is much higher after implementing this, while the correctness is better guaranteed. Below shows the global illumination for the 3 channels and the combination of them. The intensity values of each single channel are stored in a numpy file and combined to an image by simply adding them to each channel of an image.

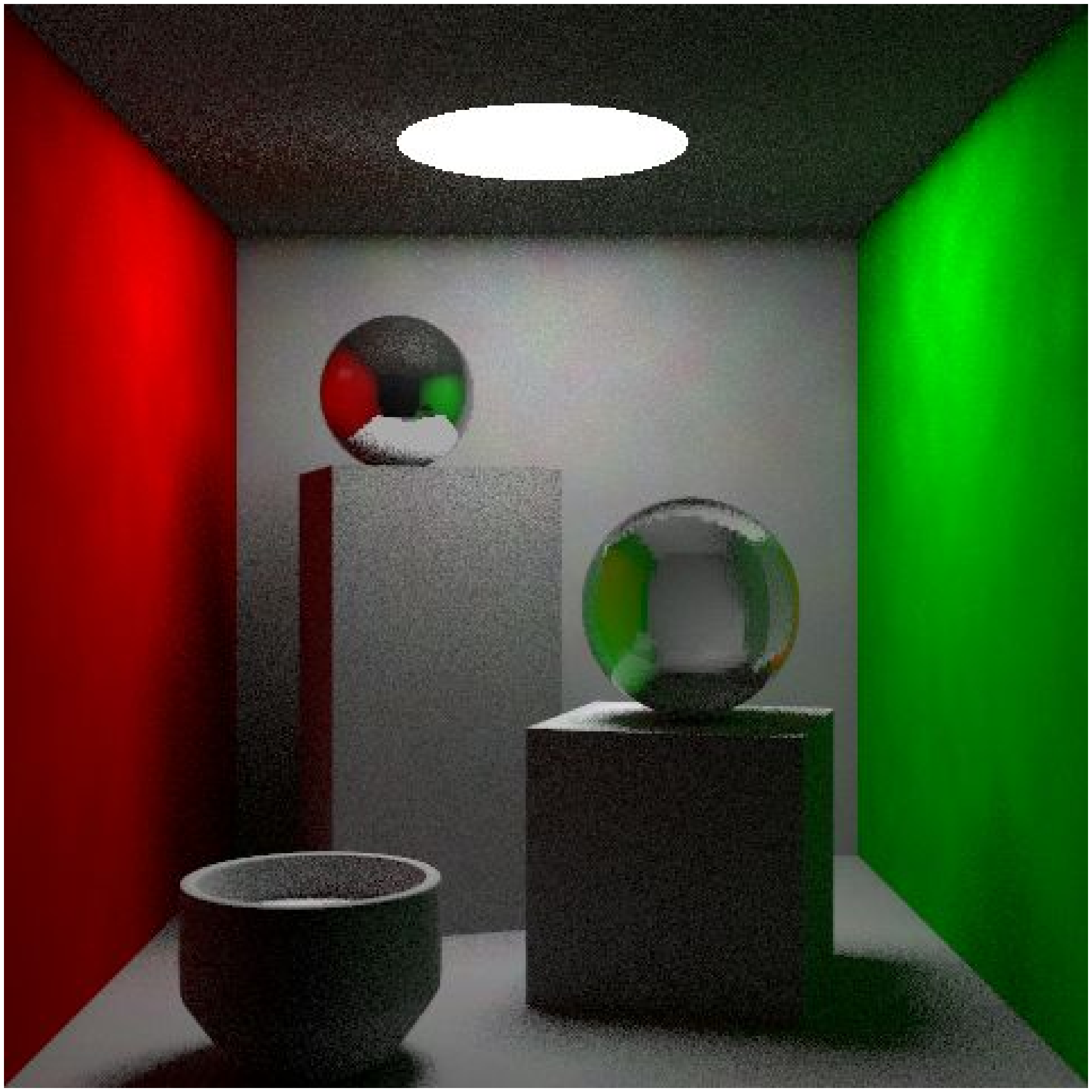
The bottom right image is the combination of the 3 channels, with the resolution being 480 by 480. We rendered the scene with SimpleRT implemented in HW5 with 16 samples and the same 480 by 480 resolution separately, as shown below. Note that we are deleting the ambient light from HW5. However we did not delete the color bleeding since there are not enough red photons going to the left surface of the tall box and green photons to the right surface of the small box. This is not expected and we are going to solve this issue in the future. Right now we use the color bleeding implemented in HW5 and use the global illumination computed from the photon maps to replace ambient light only. The following left image is the rendering result from simpleRT and the right image is the left one combined with photon map global illumination.

After adding the global illumination the image seems darker, this is due to the range of intensity being increased by adding the illumination and we need to scale down to keep it within range 0 to 255. To lower this effect we tuned the coefficients of the global illumination and direct illumination a little bit and got a better image below.



Finally we showed the light at the top.



Geometric Modeling

We employed a procedural modeling technique to build a 3D plant pot using modifiers and curves in Blender. Specifically we used the subdivision surface modifier [1] to great effect in order to improve the smoothness of the pot. We also added a circular surface for the soil. We then modeled the leaves of the plants starting from Nurbs surfaces as they suited well for smooth organic shapes.

Work Breakdown

- Zhejin : Photon Mapping
- Pramod : Geometric Modeling

Repository

https://github.com/YHHHCF/CS148Project

References:

1. Subdivision Surface Modifier, https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html
2. Modeling Plant 3D, https://youtu.be/TXuYNcp1erc
3. PLY Writer: https://github.com/dranjan/python-plyfile
4. Point Cloud Visualizer:https://github.com/daavoo/pyntcloud
5. Photon Map paper: http://graphics.ucsd.edu/~henrik/papers/photon_map/
6. Slides: http://web.stanford.edu/class/cs148/pdf/class_10_photon_mapping.pdf
7. Previous CS148 HW8: http://web.stanford.edu/class/cs148/assignments/assignment8.pdf