Report 2

Hrishikesh Jedhe Deshmukh - 210050073
Sankalan Baidya - 210050141

March 2024

1 Monte Carlo Path Tracing

1.1 Multi Sampling

For this we modified the `image_t` class and added a new variable named `num_samples`. We also modified the XML parser appropriately. For multisampling, instead of shooting a ray from a pixel at (i,j), we are shooting multiple rays from the group of 3x3 pixels around (i,j) randomly, num_sample times and then averaging the output color. This essentially reduces the aliasing of the image, but the image also appears a little bit blurred.

1.2 Area Lights and BRDFs

We created a new class `area_light_t` which inherits from `point_light_t`. It only has a color associated to it and an `object_t` type pointer which corresponds to the object associated with the arealight, just to represent general area lights. Since, each object is also associated with an arealight, we added a new `light_t` type pointer inside the object class as well. The material of the object associated with the area light doesn't really matter, the object just determines the position and shape of the light. We also created a new `disc_t` class, just to represent disc shaped area lights and there was in the final cornell box image. We modified the XML parser appropriately, and the area lights associated objects are also added to the object list.
The `area_light_t` direct method, just returns the color of the light and nothing else. For lighting computations, we shoot multiple rays for a particular pixel and let it bounce around. For diffuse materials, we sample a ray from its **BRDF** i.e. a uniform hemisphere and for imperfect specular surfaces, we sample a ray in a random direction pointed towards the reflected ray. For perfectly reflecting and for refracting surfaces, the code is same as `whitted_integrator_t`. If a ray hits the area light object, we return it's color, otherwise we let it bounce until a certain depth. We modulate the indirect lighting, using the coefficients defined in the XML file.

By doing this, the image we get is pretty noisy, but looks way more realistic. Increasing the number of bounces, also lets us see **caustics**. We also get soft shadows implicitly using this method, but since the question asks us to sample soft shadows, we have made a function inside `area_light_t` which basically samples certain number of points(`num_samples`) on the surface of the area light, and then shoots a ray towards that point from the hit point of the object, and sees, if there is an intersection. Because we are shooting multiple shadow rays, towards different points, there may

or may not be intersections, so we average them out by dividing using `num_samples`. This is then subtracted from unity , is then multiplied with the lighting component. If the aggregate is small, we get no shadow for the current pixel and if it is large, we get a dark pixel, while the edges of the shadow are relatively softer because of this aggregation.
For to generate the final image we commented out this function, because we get shadows implicitly, and also that creating shadows this way blocks effects like caustics, as the shadow ray doesn't know if the object it intersects with is a transparent object or not. We felt, handling all those cases would've been a pain, so we were fine with the soft shadows, which were implicitly created.