6.338 – Parallel Computing — Brian Anderson
Final Project Report — May 16, 2005

Parallel Monte Carlo Raytracing

I. Introduction

Raytracing is a method for rendering 3D shapes onto a 2D image. In it's simplest form, a single ray is shot per pixel and colored using the closest object the ray intersects. This basic concept of raytracing is straightforward, but can be extended almost limitlessly to approximate reality more closely. In this project I aim to augment a simple raytracer to have fuzzy reflections, soft shadows, and depth of field effects by using Monte Carlo methods. Implementing these effects require the averaging of many randomly generated rays, which lends itself well to parallelization. Although distributing rays across processors for load balancing may seem straightforward, it will be seen that we must take care not to reduce the variance of the Monte Carlo methods in our selection of rays compared to the serial version.

II. Simple Raytracing

For this project I extended a simple raytracer that I made while I took 6.374 – Introduction to Computer Graphics. The simple raytracer has four basic aspects to it: ray selection, object intersection, shading/lighting, and reflections/refractions.

Ray selection is related to the type of camera being used to view the scene. The simple raytracer implements an orthographic camera and a perspective camera. The orthographic camera emits one ray per pixel, where each ray has a different offset origin and the same direction as the camera. This results in objects appearing to be the same size independent of the objects distance from the camera. The perspective camera emits rays that all originate from a single point, but with directions that are distributed evenly across a field of view angle. This results in objects appearing to grow smaller as they move away from the camera, similar to the way we view the world.

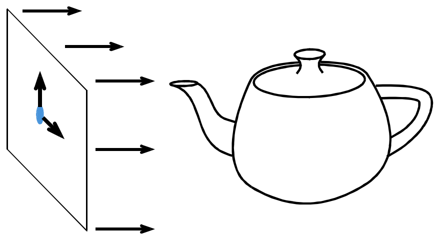
Figure 1: Orthographic Rays

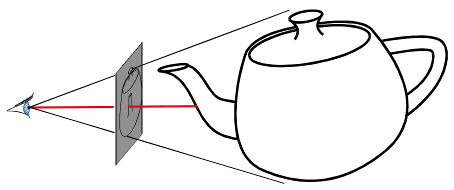
Figure 2: Perspective Rays

(Images taken from 6.837 notes by Durand & Cutler)

Object intersection takes one of these rays and an object and determines at what points along the ray it intersects the object. The simple raytracer implements intersections of planes, triangles, spheres, cylinders, groups of objects, transformation of objects, and boolean combinations of objects. A ray is intersected with every object in the scene and the closest intersection to the ray's origin is used as the contribution to that ray's color.

The color contribution is calculated by first shading the intersection point. A ray is shot from the intersection point to every light. If there is an intersection between the point and the light, it is considered to be in the shadow and no illumination is added. However, if there is no intersection, the diffuse and specular components of the illumination are calculated separately and combined. The diffuse component is essentially the flux of light hitting the surface area, and is calculated by taking the dot product of the surface normal with the relative light direction. The specular component simulates glossy materials whose reflection of light depends more on the incident angle of the light. It is calculated by taking the dot product of the ray with the mirrored direction of the light against the surface normal. It is then raised to some power to simulate "glossiness" where a higher power corresponds to more glossy materials.

If the ray intersection is with a reflective surface, we must also add the reflective contribution color. To do this, a reflection ray is created, whose origin is the point of intersection and whose direction is the mirrored direction of the incoming ray with the surface normal. The contribution of that ray is calculated just like any other ray and multiplied by the reflection coefficient of the intersection point before it is added. The refraction ray of transparent objects are handled similarly by using Snell's law to calculate the direction of the refracted ray.

III. Monte Carlo Soft Shadows

The simple raytracer assumes all light comes from a point source, which means that every point in the scene is either illuminated by a given light or not. Point light sources thus result in sharp and unrealistic looking shadows. Instead we would like to use area light sources. To implement this, we would ideally like to integrate the contribution of every point on the area of the light. Using Monte Carlo methods, however, we can approximate the integration by shooting many random rays to the area of the light source, averaging the contribution of all the rays.

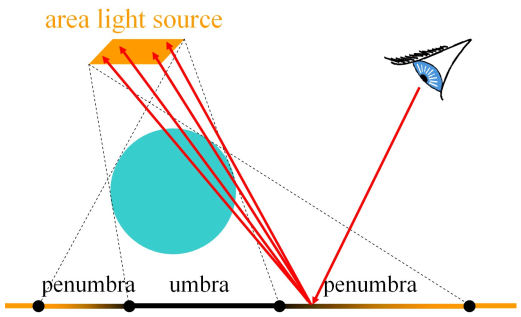

Figure 3: Soft Shadows
(Image taken from 6.837 notes by Durand & Cutler)

IV. Monte Carlo Fuzzy Reflections

The simple raytracer assumes that all reflections are perfect mirror reflections. In reality, there are reflective surfaces that reflect light from many directions. The amount of reflection from each direction can be represented by a probability density function of that direction dotted with the mirrored ray direction. Using the Monte Carlo method, we can shoot many rays in random directions for each reflection, averaging their contributions as weighted by the probability density function.

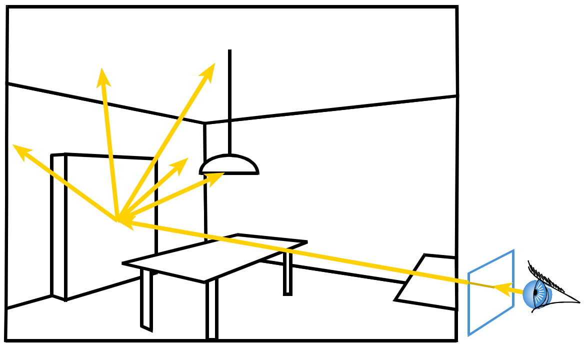
Figure 4: Fuzzy Reflections
(Image taken from 6.837 notes by Durand & Cutler)

V. Monte Carlo Depth of Field

The simple raytracer assumes a camera with an infinitely small aperture, which results in the entire scene being in focus no matter how close of far an object is. To implement depth of field, we need to introduce a new camera that has a focal distance and an aperture. To calculate the value of a pixel now, we would like to integrate all of the light that hits the aperture. Again, this can be approximated using Monte Carlo by shooting many rays with a random origin on the aperture and a direction that goes through the pixel's focal point, averaging the contribution of all the rays.

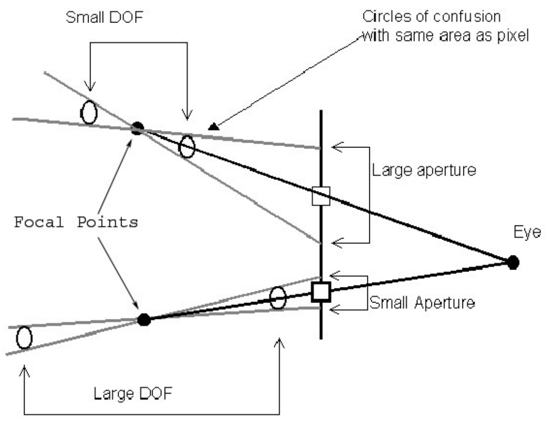

Figure 5: Depth of Field
Image adapted from: http://glasnost.itcarlow.ie/~powerk/Graphics/Notes/node12.html

VI. Reducing the Variance of Monte Carlo Methods

The noise visible in the Monte Carlo images is due directly to the variance introduced by the random sampling. Choosing completely random rays from a uniform distribution results in a variance that converges with $\sqrt{N}$, where N is the number of rays sampled. There exists many methods to reduce this variance for a given N, which in turn reduces the noise for a given amount of calculation. The two methods I use in this project are importance sampling (Figure 6) and stratification (Figure 7).

Importance sampling chooses rays that have a higher contribution with a higher probability. It can be used in the case of fuzzy reflections by choosing directions according to the probability density function which is already defined.

Stratification partitions an area to be sampled into N equal area parts. A single random sample is taken within each partition, resulting in a good distribution of selected points. Stratification allows the variance to converge with N – which is a tremendous improvement. This method can be applied directly to the sampling of area lights for soft shadows as well as apertures for depth of field.

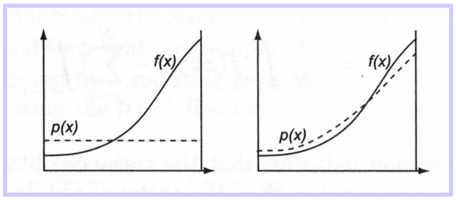

Figure 6: Importance Sampling

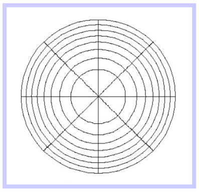

Figure 7: Stratification

(Images taken from 6.839 notes by Durand & Cutler)

VII. Parallelization

There are many ways to attempt to load balance the rays across multiple processes. One way is to have each process calculate a region of the image independently. This method however, may result in poor load balancing if some parts of the image have more reflections than others.

A better way would be for each process to calculate the entire image independently using N/P rays per pixel, where P is the number of processes, and then average all of the images. We have to make sure though that the selection of the N/P rays does not reduce the variance improvements brought about in the serial version.

Importance sampling is purely probabilistic, so there is nothing that must be communicated between processes when selecting the rays. Stratification, however, is different. It is better to have, say, an area partitioned into 16 parts with 1 sample per partition than to have the area partitioned into 4 parts with 4 samples per partition. Therefore, we must make sure that each process calculates a part of the total stratification of N rays instead of it's own stratification of N/P rays.

VIII. Results

If parallelized according to the method described in the previous section, there should be a decrease in runtime linearly proportional to the number of processes. This is indeed the case. Runtimes for a 200x150 image with 16 aperture, 4 shadow, and 4 reflection rays are 2:31 for 1 process, 1:16 for 2 processes, 0:38 for 4 processes, 0:21 for 8 processes, and 0:15 for 16 processes. Runtime gains begin to fall off at 16 processes, most likely due to the increased communication time relative to the processing time.

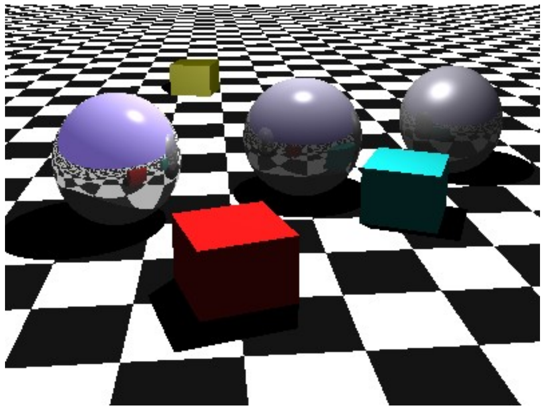

Figure 8: Simple Raytraced Version

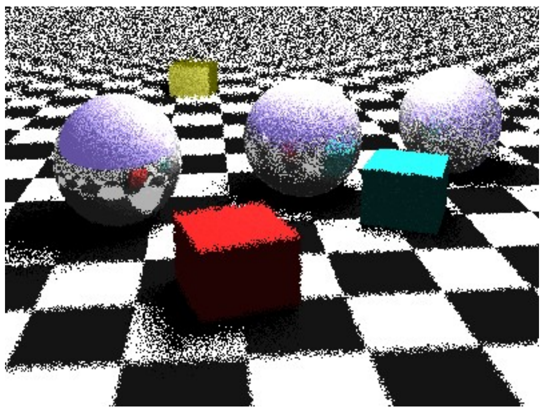

Figure 9: Monte Carlo (1 ray per everything)

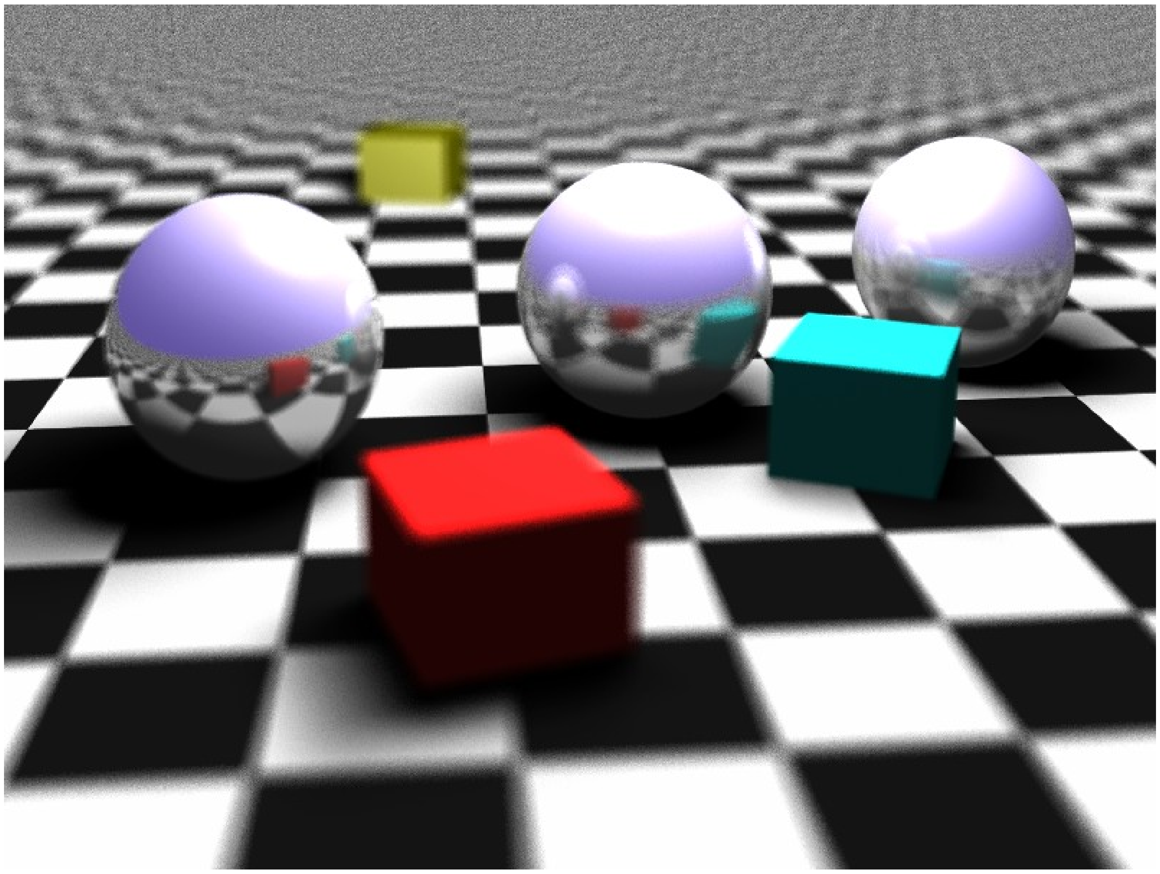

Figure 10: Mote Carlo (49 rays per aperture, 9 per area light, 16 per reflection)

IX. Conclusion

Raytracing is an embarrassingly parallelize-able algorithm, and the Monte Carlo methods are almost as embarrassing. As long as we take care not to dilute the effect of the variance optimizations when we distribute the rays across processes, we can expect an X times speed up for an X times increase in the number of processes.