

# MultiThreaded PathTracer

OS Project

## Overview

**Path tracing** is a computer graphics Monte Carlo method of rendering images of three-dimensional scenes such that the global illumination is faithful to reality. Fundamentally, the algorithm is integrating over all the illuminance arriving to a single point on the surface of an object. **Global Illumination (GI)** is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light).

We see things because light emitted by light sources such as the sun bounces off of the surface of objects. When light rays bounce only once from the surface of an object to reach the eye, we speak of *direct illumination*. But when light rays are emitted by a light source, they can bounce off of the surface of objects multiple times before reaching the eye. This is what we call *indirect illumination* because light rays follow complex paths before entering the eye. This effect is clearly visible in the images above. Some surfaces are not exposed directly to any light sources (often the sun), and yet they are not completely black. This is because they still receive some light as an effect of light bouncing around from surface to surface.

So when would you want to use global illumination? Global illumination certainly isn't going to be the "one size fits all" way to achieving photorealistic renders in every project. For example, you may not want it for something like a toon-style render where you'd very specifically want to avoid indirect lighting effects. It is however, great for architectural visualization, interior renders, scenes with direct sunlight and photorealistic renders.

Basically, you would want to use global illumination whenever light needs to interreflect (or be cast back) and bounce multiple times over a large area in your scene. This is especially vital when trying to make things look as realistic as possible. Using global illumination gives you the ability to capture this indirect illumination, i.e. the real-world phenomenon where light bounces off anything in its path until it is completely absorbed.

The complexity of global illumination in terms of tracing not only direct illumination but also illumination from reflecting light across surfaces, coupled with its computationally intensive and expensive nature makes it a potential candidate for exploring parallelism to achieve better performance.

## Goals

1. To render a hardcoded scene description illuminated by an existing light source - from the perspective of a preset camera.
2. To investigate how varying SPP (samples per pixel) gives change in quality of rendering with a tradeoff in performance.

## Implementation

We have implemented this multi-threaded path tracer inspired by *Kevin Beason's smallpt*, but we have used just POSIX Pthreads instead of OpenMP for thread-level parallelism.

### Objects defined

We have defined a Vec class object to represent vectors, and overloaded all vector operations such as addition/subtraction, dot product, cross product and normalization. This allows us to define a Ray class object represented by 2 vectors - as symbolized by the ray equation  $Ray = o + td$ . We also define a struct Sphere - characterized not only by radius or position but also by color, emissivity and reflectivity of the surface. The Sphere object also has a function to determine if an input ray intersects the Sphere at any point. We also define the camera position using a Ray to denote direction and Vectors to define the position. This camera defines the perspective of our rendered image.

We then enumerate the possible reflective properties :

- DIFF - Diffused, which denotes rough surfaces on which reflecting light rays are not parallel, and therefore light reflected makes the diffused object visible to the user.
- SPEC - Spectral, which denotes polished surfaces on which reflecting light rays are parallel, and therefore form the image that the reflecting light is forming on the surface to the user.
- REFR - Refracting, which denotes surfaces that refract light rays instead of reflecting them.

We then define a hardcoded description of our sample scene by defining the appropriate Spheres - We have Spheres with INF length radii that enclose the whole frame and act as our closed environment. We then place two sample Spheres with different reflective properties for analysing. We also place a sphere on the top that acts as the light source. All Spheres are coded with different colors appropriately for better illustration.

We also define struct args that we use to pass appropriate arguments to each thread - since POSIX does not support direct use of defined variables in the main scope unlike OpenMP.

## Approach

We know that an image is split into constituent pixels. We further split each pixel into 4 subpixels and calculate the *radiance*(illumination) for each of them. We first check if the light rays generated meet the point - if it doesn't, we simply don't illuminate it (it is black). If it does, we identify which Sphere here is illuminated by the ray - it is notable that our walls, floor and ceiling are also surfaces of Spheres. We then normalize the vector and capture the component of the ray that is responsible for maximum reflection. We then analyse the property of the Sphere, and illuminate accordingly. Apart from DIFF, SPEC and REFR, we also account for TIR - Total Internal Reflection recursively.

If light rays do not diminish in an ideal environment, then how can we terminate illumination without ending up with pure white everywhere? We do this by a method called *Russian Roulette*. Russian roulette, similar to the gun-game of the same name, uses probability to determine at what point can we terminate the illumination of the particular ray. We terminate each ray randomly before 6 bounces, each while checking if the maximum reflectivity component is still valid enough to continue (and stop if it becomes negligible instead).

The radiance is based upon Monte Carlo integration, on the rendering equation so that each sample has the same Expected Value - such that with increase in the samples per pixel, the render becomes increasingly accurate :

$$\widehat{L}(P \rightarrow D_v) = L_e(P \rightarrow D_v) + \frac{F_s(D_v, D_i) |\cos \theta| \widehat{L}(Y_i \rightarrow -D_i)}{p_{angle}^{tot}(D_i)}$$

$$\frac{F_s(D_v, D_i) |\cos \theta| \widehat{L}(Y_i \rightarrow -D_i)}{p_{angle}^{tot}(D_i)}$$

With respect to the multithreading, we have used POSIX Pthreads to parallelise the radiance calculation of each row of pixels in the rendered image. Each thread executes a runner function that calculates the radiance for each subpixel and SPP samples for each pixel. The resulting radiance values are finally clamped and mapped between 0 to 255 to support hex values of displays. We have also taken a gamma factor of 2.2 to compensate for decrease in monitor brightness.

## Code

```
#include <bits/stdc++.h>
#include <stdlib.h>
#include <pthread.h>
using namespace std;

const double INF = 1e20;

// GLOBAL DEFINITION
int h = 768, w = 1024, samps; // height, width and SPP

double erand()
{
    return (double)rand() / RAND_MAX;
}

// structure defining Vector
struct Vec
{
    // components of the vector
    double x, y, z;

    // constructor for Vec
    Vec(double x_ = 0, double y_ = 0, double z_ = 0)
    {
        x = x_;
        y = y_;
```

```
    z = z_;
```

```
    }
```

```
    // Vector addition
```

```
    Vec operator+(const Vec &b) const
```

```
    {
```

```
        return (Vec(x + b.x, y + b.y, z + b.z));
```

```
    }
```

```
    // Vector subtraction
```

```
    Vec operator-(const Vec &b) const
```

```
    {
```

```
        return (Vec(x - b.x, y - b.y, z - b.z));
```

```
    }
```

```
    // Scalar multiplication
```

```
    Vec operator*(double b) const
```

```
    {
```

```
        return (Vec(x * b, y * b, z * b));
```

```
    }
```

```
    // Vector multiplication
```

```
    Vec mult(const Vec &b) const
```

```
    {
```

```
        return (Vec(x * b.x, y * b.y, z * b.z));
```

```
    }
```

```
    // Dot Product
```

```
double dot(const Vec &b) const
{
    return (x * b.x + y * b.y + z * b.z);
}

// Normalize Vector
Vec &norm()
{
    return *this = *this * (1 / sqrt(x * x + y * y + z * z));
}

// Cross product
Vec operator%(const Vec &b) const
{
    return Vec(y * b.z - z * b.y, z * b.x - x * b.z, x * b.y - y *
b.x);
}
};

// structure defining a Ray
struct Ray
{
    Vec o, d; // Ray = o + td -> o,d are constant vectors

    // constructor
    Ray(Vec o_, Vec d_)
    {
        o = o_;
    }
};
```

```
        d = d_;
    }
};

// GLOBAL DECLARATION

// setting up camera position and direction
Ray cam(Vec(50, 52, 295.6), Vec(0, -0.042612, -1).norm());

Vec cx = Vec(w * .5135 / h), cy = (cx % cam.d).norm() * .5135;

// Materials used in radiance
enum Refl_t
{
    DIFF,
    SPEC,
    REFR
};

struct Sphere
{
    double rad; //radius

    Vec p, e, c; //position,emission,color

    Refl_t refl; //reflection type

    // constructor for a sphere
    Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_)
    {
```



```
rad = rad_;
p = p_;
e = e_;
c = c_;
refl = refl_;
}

// returns distance if intersects, 0 if no intersection
double intersect(const Ray &r) const
{
    //  $t^2 \cdot d \cdot d + 2 \cdot t \cdot (o-p) \cdot d + (o-p) \cdot (o-p) - R^2 = 0$  solves intersection
    point of Ray and Sphere
    Vec op = p - r.o;
    double t, eps = 1e-4; // eps - epsilon
    double b = op.dot(r.d); // 1/2 b from
    quadratic equation
    double det = b * b - op.dot(op) + rad * rad; //  $(b^2 - 4ac)/4$  : a=1
    because ray is normalized

    if (det < 0)
    {
        // ray misses sphere
        return 0;
    }
    else
    {
        // ray hits sphere
        det = sqrt(det);
    }
}
```

```
    if (b - det > eps)
    {
        t = b - det;
    }
    else if (b + det > eps)
    {
        t = b + det;
    }
    else
    {
        t = 0;
    }

    return t;
}
};

// Scene definition - hardcoded -could be modified to render different
// images
Sphere spheres[] = {
    //Scene: radius, position, emission, color, material
    Sphere(1e5, Vec(1e5 + 1, 40.8, 81.6), Vec(), Vec(.75, .25, .25), DIFF),
//Left
    Sphere(1e5, Vec(-1e5 + 99, 40.8, 81.6), Vec(), Vec(.25, .25, .75),
DIFF), //Right
    Sphere(1e5, Vec(50, 40.8, 1e5), Vec(), Vec(.75, .75, .75), DIFF),
//Back
```

```
    Sphere(1e5, Vec(50, 40.8, -1e5 + 170), Vec(), Vec(), DIFF),
//Front

    Sphere(1e5, Vec(50, 1e5, 81.6), Vec(), Vec(.75, .75, .75), DIFF),
//Bottom

    Sphere(1e5, Vec(50, -1e5 + 81.6, 81.6), Vec(), Vec(.75, .75, .75),
DIFF), //Top

    Sphere(16.5, Vec(27, 16.5, 47), Vec(), Vec(1, 1, 1) * .999, SPEC),
//Mirror

    Sphere(16.5, Vec(73, 16.5, 78), Vec(), Vec(1, 1, 1) * .999, REFR),
//Glass

    Sphere(600, Vec(50, 681.6 - .27, 81.6), Vec(12, 12, 12), Vec(), DIFF)
//Light
};

// alternate scene definitions can be found in
https://www.kevinbeason.com/smallpt/extraScenes.txt

// clamps value between 0 and 1
inline double clamp(double x)
{
    return x < 0 ? 0 : x > 1 ? 1 : x;
}

// Maps between 0 to 255, takes gamma factor 2.2 into account
inline int toInt(double x)
{
    return int(pow(clamp(x), 1 / 2.2) * 255 + .5);
}

// Checks if ray intersects the spheres
inline bool intersect(const Ray &r, double &t, int &id)
```

```
{  
  
    double n = sizeof(spheres) / sizeof(Sphere), d;  
    t = INF;  
  
    for (int i = int(n) - 1; i >= 0; i--)  
    {  
  
        d = spheres[i].intersect(r);  
  
        if (d > 0 && d < t)  
        {  
  
            t = d;  
  
            id = i;  
  
        }  
  
    }  
  
    return t < INF;  
}
```

```
Vec radiance(const Ray &r, int depth)  
{  
  
    double t; // distance to intersection  
    int id = 0; // id of intersected object  
  
    if (!(intersect(r, t, id)))  
    {  
  
        return Vec(); // if miss, return black  
  
    }  
  
    const Sphere &obj = spheres[id]; // hit object
```

```
Vec x = r.o + r.d * t, n = (x - obj.p).norm();
Vec nl;
if (n.dot(r.d) < 0)
{
    nl = n;
}
else
{
    nl = n * -1;
}
Vec f = obj.c;

double p = max({f.x, f.y, f.z}); // maximum reflection

if (++depth > 5)
{
    if (erand() < p)
    {
        f = f * (1 / p);
    }
    else
    {
        return obj.e; // Russian Roulette
    }
}

if (obj.refl == DIFF)
{
```

```

// Ideal DIFFUSE reflection

double r1 = 2 * M_PI * erand(), r2 = erand(), r2s = sqrt(r2);
Vec w = nl, u;

if (fabs(w.x) > 0.1)
{
    u = Vec(0, 1) % w;
}
else
{
    u = Vec(1) % w;
}

u = u.norm();
Vec v = w % u;

Vec d = (u * cos(r1) * r2s + v * sin(r1) * r2s + w * sqrt(1 -
r2)).norm();

return obj.e + f.mult(radiance(Ray(x, d), depth));
}

else if (obj.refl == SPEC)
{
    // Ideal specular reflection

    return obj.e + f.mult(radiance(Ray(x, r.d - n * 2 * n.dot(r.d)),
depth));
}

Ray reflRay(x, r.d - n * 2 * n.dot(r.d)); // Ideal dielectric
Refraction

```

```
    bool into = n.dot(n1) > 0; // Check if ray is going from outside
    towards inside

    double nc = 1, nt = 1.5, nnt;

    if (into)
    {
        nnt = nc / nt;
    }
    else
    {
        nnt = nt / nc;
    }

    double ddn = r.d.dot(n1), cos2t;

    // Total Internal Reflection TIR
    if ((cos2t = 1 - nnt * nnt * (1 - ddn * ddn)) < 0)
    {
        return obj.e + f.mult(radiance(reflRay, depth));
    }

    Vec tdir = (r.d * nnt - n * ((into ? 1 : -1) * (ddn * nnt +
sqrt(cos2t)))) .norm();

    double a = nt - nc, b = nt + nc, R0 = a * a / (b * b), c;

    if (into)
    {
        c = 1 + ddn;
    }
}
```

```
else
{
    c = 1 - tdir.dot(n);
}

double Re = R0 + (1 - R0) * c * c * c * c * c, Tr = 1 - Re, P = .25 +
.5 * Re, RP = Re / P, TP = Tr / (1 - P);

if (depth > 2)
{
    if (erand() < P)
    {
        return obj.e + f.mult(radiance(reflRay, depth) * RP);
    }
    else
    {
        return obj.e + f.mult(radiance(Ray(x, tdir), depth) * TP);
    }
}
else
{
    return obj.e + f.mult(radiance(reflRay, depth) * Re +
radiance(Ray(x, tdir), depth) * Tr);
}
}

// structure defining argument for appropriate passing to threads
struct args
```



```
{
    int id;
    Vec *c;
};

// mutex m;

void *runner(void *arg)
{
    args *item = (args *)arg;
    int y = item->id;
    Vec *cc = item->c;

    Vec r;

    // Loop columns
    for (unsigned short x = 0; x < w; x++)
    {
        // 2x2 subpixel rows
        for (int sy = 0, i = (h - y - 1) * w + x; sy < 2; sy++)
        {
            for (int sx = 0; sx < 2; sx++, r = Vec())
            { // 2x2 subpixel cols
                for (int s = 0; s < samps; s++)
                {
                    double r1 = 2 * erand(), dx = r1 < 1 ? sqrt(r1) - 1 : 1
- sqrt(2 - r1);
```

```
double r2 = 2 * erand(), dy = r2 < 1 ? sqrt(r2) - 1 : 1
- sqrt(2 - r2);

Vec d = cx * (((sx + .5 + dx) / 2 + x) / w - .5) +
        cy * (((sy + .5 + dy) / 2 + y) / h - .5) +
cam.d;

r = r + radiance(Ray(cam.o + d * 140, d.norm()), 0) *
(1. / samps);

} // Camera rays are pushed forward to start in interior
cc[i] = cc[i] + Vec(clamp(r.x), clamp(r.y), clamp(r.z)) *
.25;

}

}

}

}

int main(int argc, char const *argv[])
{
    srand(time(0));

    // samps -> samples
    if (argc == 2)
    {
        samps = atoi(argv[1]) / 4;
    }
    else
    {
        samps = 1;
    }
}
```

```
Vec *c = new Vec[w * h];

// setting up arguments for passing to threads
args A[h];
for (int i = 0; i < h; i++)
{
    A[i].id = i;
    A[i].c = c;
}

pthread_t threads[h];

// double time_;
// time_ = clock();
fprintf(stderr, "\rRendering (%d spp)\n", samps * 4);
// parallel code
// spawn threads for each row of the image
for (int i = 0; i < h; i++)
{
    pthread_create(&threads[i], NULL, runner, &A[i]);
}

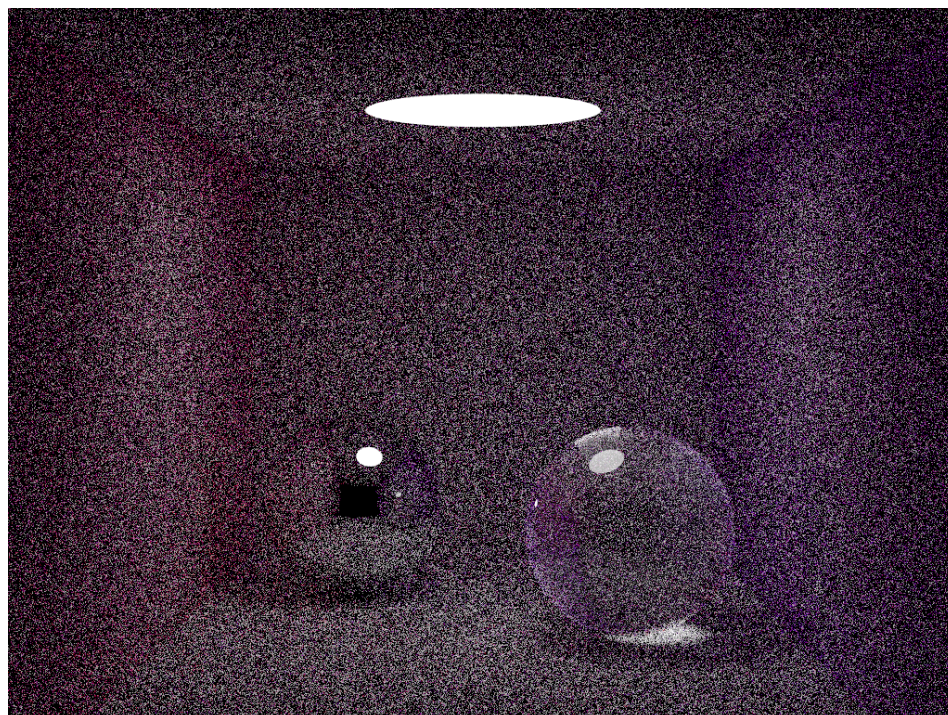
// join all threads spawned back to master
for (int i = 0; i < h; i++)
{
    pthread_join(threads[i], NULL);
}
```

```
// write image to ppm file
FILE *f = fopen("Rendered_image.ppm", "w");
fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
for (int i = 0; i < w * h; i++)
{
    fprintf(f, "%d %d %d ", toInt(c[i].x), toInt(c[i].y),
toInt(c[i].z));
}
// time_ = clock() - time_;
// cout << "Processor Time taken : " << (double)time_ / CLOCKS_PER_SEC
<< " seconds\n";
}
```

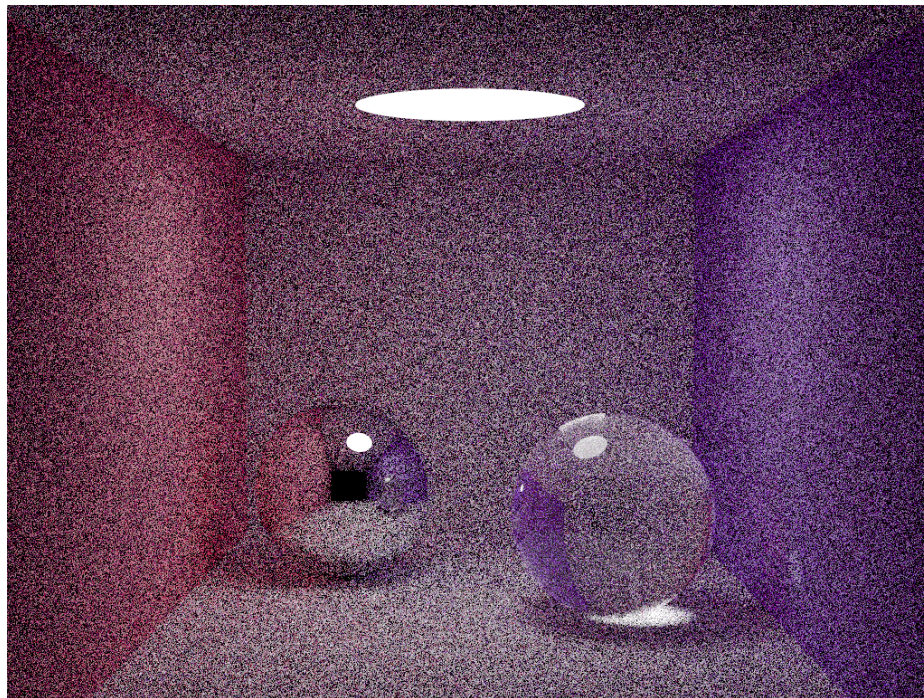
## Outputs

It is noted that in the absence of exceedingly parallel systems like GPUs, relying solely on CPU parallelism for rendering is computationally draining. Yet, we are able to observe a stark contrast and increasingly superior quality in the rendered images as we increase the SPP value on execution. To see the demo video, click [here](#). The corresponding outputs are attached below :

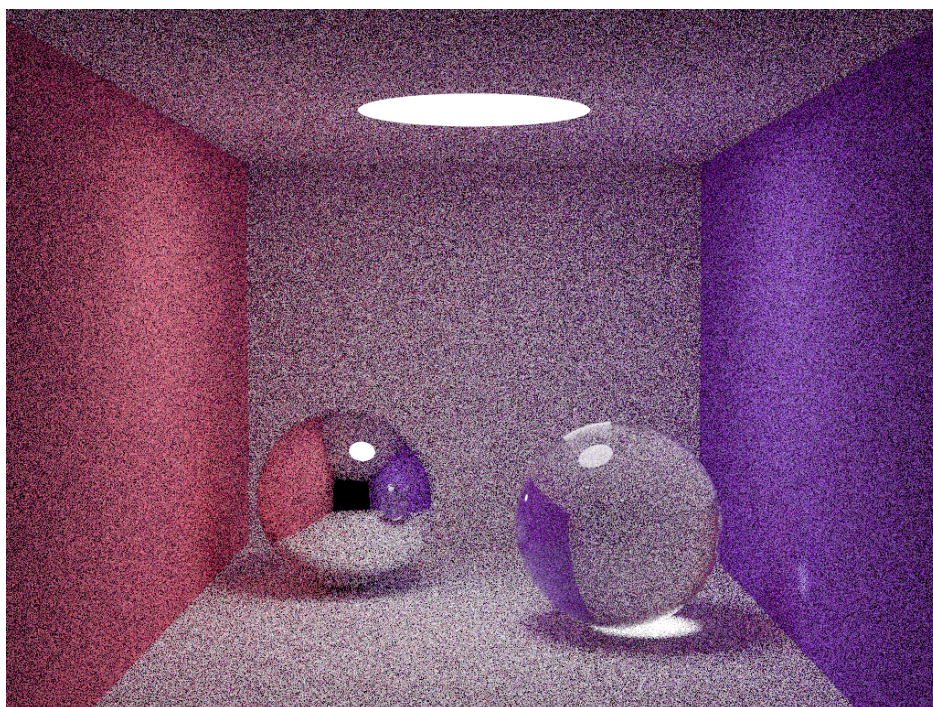
8 SPP - Render time : 1m 11sec



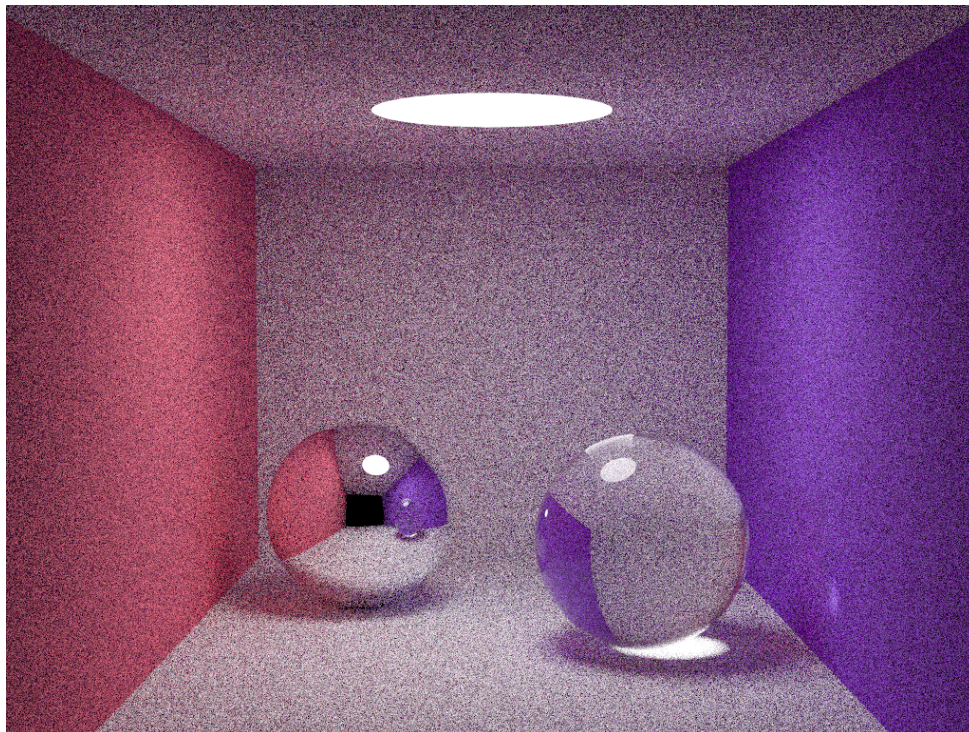
20 SPP - Render time : 2m 46sec



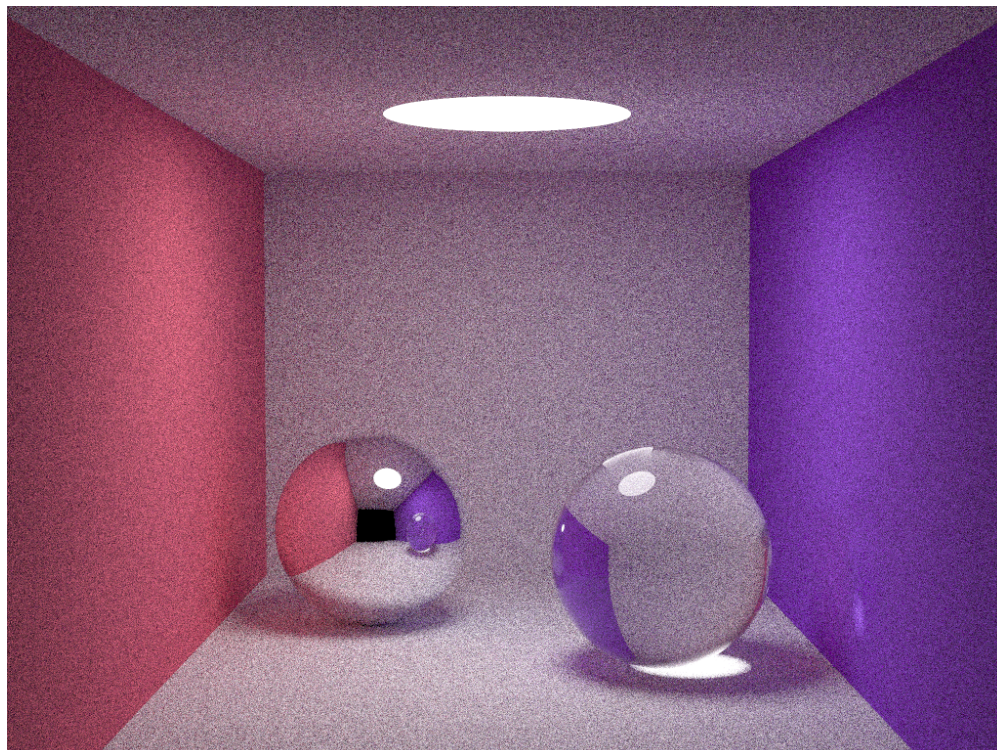
40 SPP - Render time : 6m 9sec



80 SPP - Render time : 11m 27sec

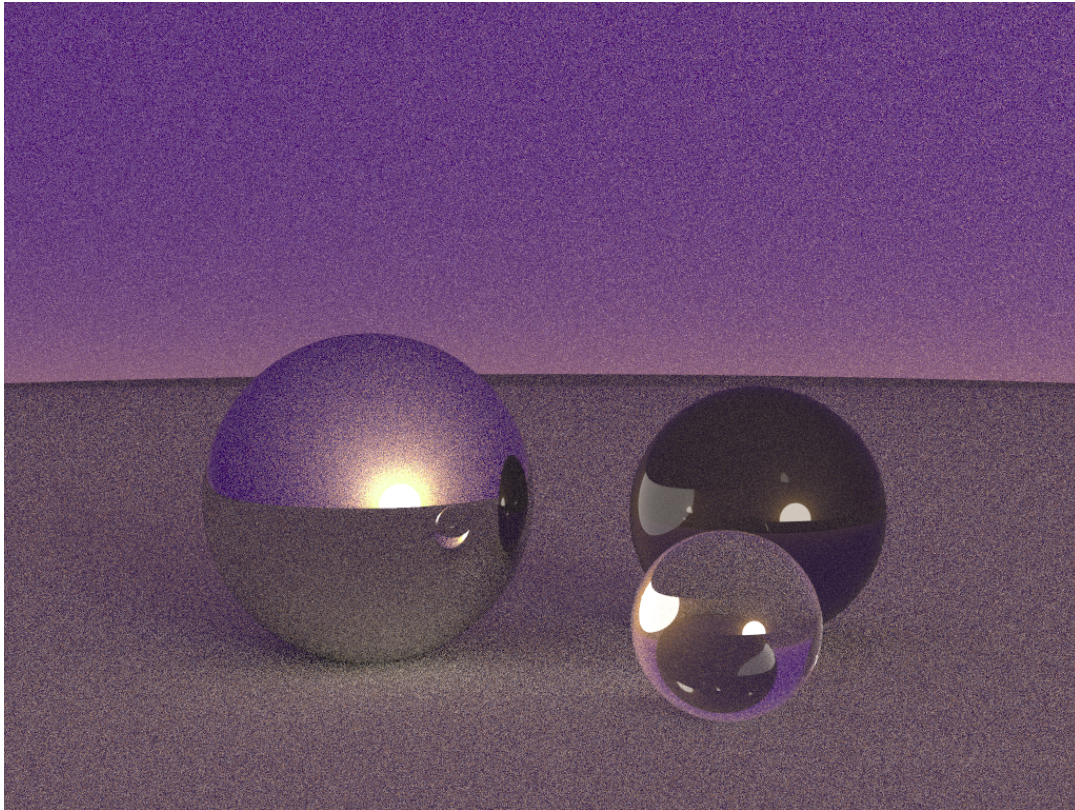


200 SPP - Render time : 28m 1sec

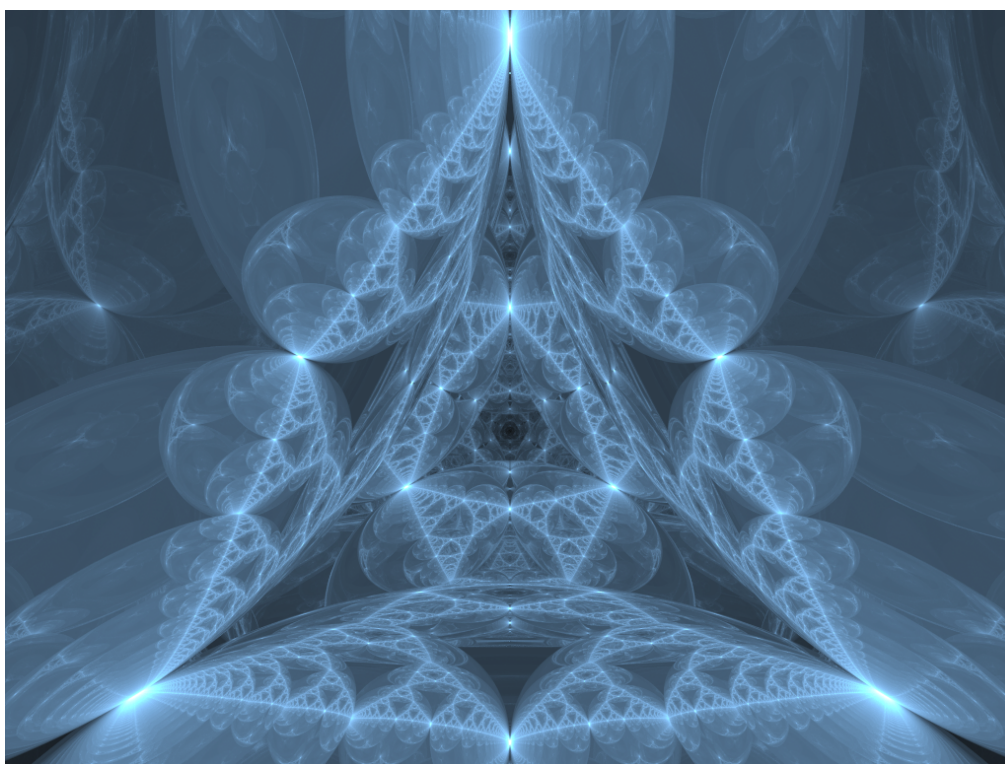
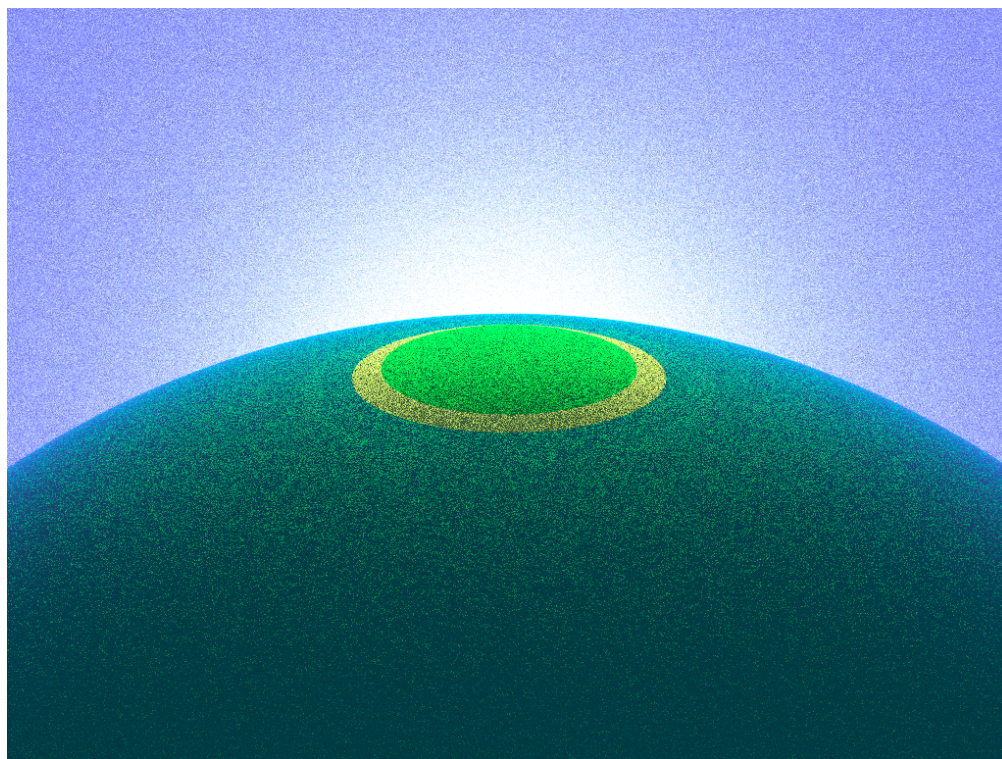


**Note :** All testing and benchmarking was done on a Lenovo 1.6GHz i5-8th Gen machine with 12GB RAM, running Ubuntu 18.04.


We have also rendered some alternate scene descriptions, also at 200 SPP :







Notice in the last image how the lack of reflective surfaces leads to lesser light bounces and thus greatly increases the accuracy of the render. This is due to the nature of Monte Carlo experiments wherein probabilistic accuracy is inversely proportional to the number of



traces in sampling (in other words, lesser light bounces lead to lesser calculations and thus gives better and more accurate output at a lesser SPP value).

## Conclusion

As we increase the SPP (samples per pixel), we are clearly able to observe the sharp increase in the quality of the rendered image. This is because as we increase the number of samples per pixel, we also increase the probability of our accuracy in calculating the precise radiance of each pixel. This demonstration of variance in sampling is the fundamental idea behind Monte Carlo experiments. The use of multithreading also enables us to perform the computation at around  $(1/h)$  times faster - i.e  $(1/768)$  times as per our experiment where we render a 768x384 image .

Thus, we have demonstrated the working of a multithreaded PathTracer and shown the variance of render quality with respect to change in samples per pixel values.