Let’s make image more realistic

1. Objective

In computer graphics, the goal of photorealistic rendering is to create an image of a 3D scene that is indistinguishable from photograph of same scene. Basically, it is a process of making a digital image in the computer. The digital image is fake but it looks just like real photo. The technic, or the level of this rendering is highly sought after by product design, interior design and architecture so that potential clients may get a feel of the actual product at least at a visual level.

Almost all photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. But a major disadvantage of ray tracing is that its computation complexity is about square of object number. More object on the scene, much more time needed for rendering. So finding a way to reduce computation time is worth exploring and researching.

2. Method and Algorithms

(1) How to tracing the light ?

1. Forward Tracing

It is emitted from the light source and travels in a straight line path until it hits the surface of our object. Ignoring photon absorption, we can assume the photon is reflected in a random direction. If the photons hits the surface of our eye, we "see" the point where the photon was reflected from.

But there are some problem: in forward tracing we assumed that the reflected photon always intersected the surface of the eye. In reality, rays are essentially reflected in every possible direction, each of which have a very, very small probability of actually hitting the eye. We would potentially have to cast zillions of photons from the light source to find only one photon that would strike the eye. In nature this is how it works, but in the computer world, simulating the interaction of that many photons with objects in a scene is just not practical solution for reasons we will now explain.

2. Backward Tracing

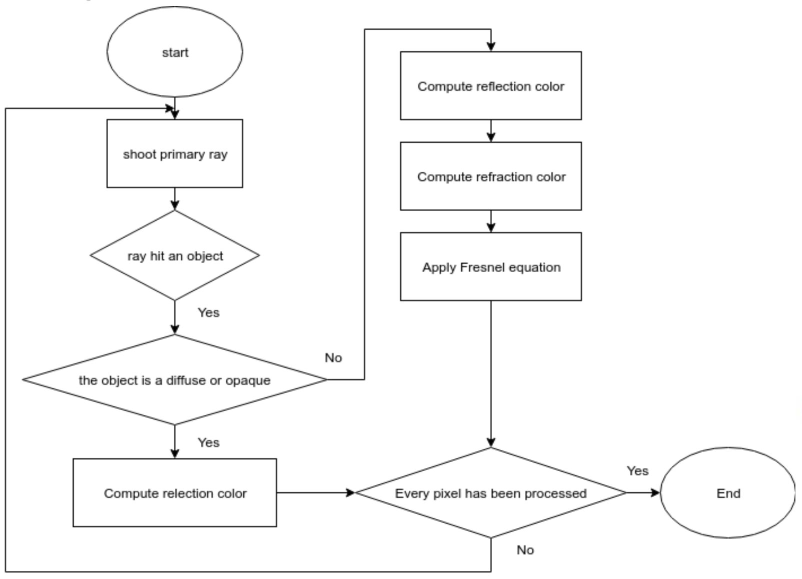
Instead of tracing rays from the light source to the receptor (such as our eye), we trace rays backwards from the receptor to the objects. This method provides a convenient solution to the flaw of forward ray tracing. If the ray hits an object then we find out how much light it receives by throwing another ray (called a light or shadow ray) from the hit point to the scene's light. Occasionally this "light ray" is obstructed by another object from the scene, meaning that our original hit point is in a shadow; it doesn't receive any illumination from the light. And the first ray we shoot from the eye into the scene is called a primary ray or camera ray.

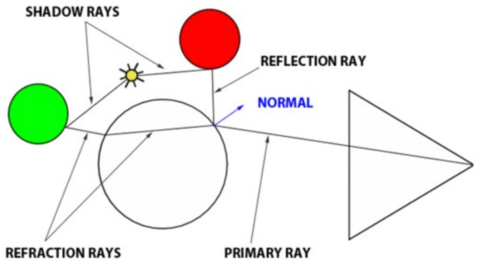
(2) The Ray Tracing Algorithm

The ray tracing algorithm takes an image made of pixels. For each pixel in the image, it shoots a primary ray into the scene. The direction of that primary ray is obtained by tracing a line from the eye to the center of that pixel. Once we have that primary ray's direction set, we check every object of the scene to see if it intersects with any of them. In some cases, the primary ray will intersect more than one object. When that happens, we select the object whose intersection point is the closest to the eye. We then shoot a shadow ray from the intersection point to the light. If this particular ray does not intersect an object on its way to the light, the hit point is illuminated. If it does intersect with another object, that object casts a shadow on it.

And finally, If we repeat this operation for every pixel, we obtain a two dimensional representation of our three dimensional scene.

(3) Algorithm Flowchart





We shoot a primary ray from the eye and the closest intersection (if any) with objects in the scene. If the ray hits an object which is not a diffuse or opaque object, we must do extra computational work. To compute the resulting color at that point on, say for example, the glass ball, you need to compute the reflection color and the refraction color and mix them together.

First we compute the reflection direction. For that we need two items: the normal at the point of intersection and the primary ray's direction. Once we obtain the reflection direction, we shoot a new ray in that direction. We find out how much light reaches that point on the red sphere by shooting a shadow ray to the light. That obtains a color (black if it is shadowed) which is then multiplied by the light intensity and returned to the glass ball's surface.

Now we do the same for the refraction. To compute the transmission direction we need the normal at the hit point, the primary ray direction, and the refractive index of the material. With the new direction computed, the refractive ray continues on its course to the other side of the glass ball. There again, because it changes medium, the ray is refracted one more time. As you can see in the adjacent image, the direction of the ray changes when the ray enters and leaves the glass object. Refraction takes place every time there's a change of medium and that two media, the one the ray exits from and the one it gets in, have a different index of refraction. Refraction has for effect to bend the ray slightly.

Lastly, we compute the Fresnel equation. We need the refractive index of the glass ball, the angle between the primary ray, and the normal at the hit point. Using a dot product, the Fresnel equation returns the two mixing values.

(4) Parallelization Technique, Reasons, and Flowchart

In this project, we use OpenMP to make some section of the program execute parallel. To know where the section is, we use GNU profiler to find out the bottleneck of the program.

3. Application

When presenting design proposals to clients, it is very important that architects have realistic renderings accompaning their designs. And architects are also primarily interested in creating visually realistic images. Because modeling light is so essential to the field, the development of programs that incorporate ray tracing has been a dream come true for many architects.

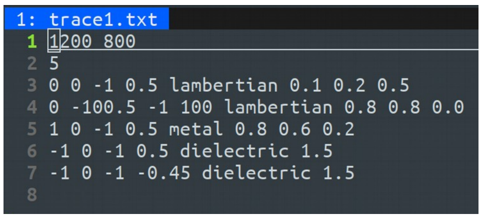
4. Serial Implementation

(1) Input of your program

Read the file that contains the location and material of the spheres to be drawn. The file contains the following information:

1. The height and width of the image.
2. The number of spheres to be drawn.
3. The location and radius and material and vector of the sphere.

For example:

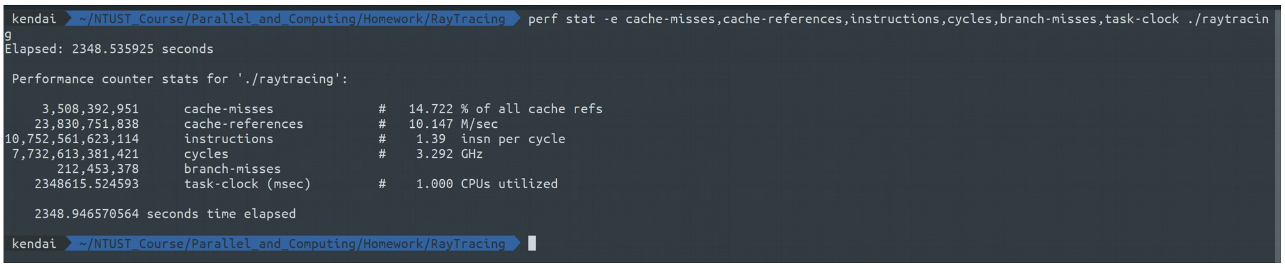


```
1: trace1.txt
1200 800
5
0 0 -1 0.5 lambertian 0.1 0.2 0.5
0 -100.5 -1 100 lambertian 0.8 0.8 0.0
1 0 -1 0.5 metal 0.8 0.6 0.2
-1 0 -1 0.5 dielectric 1.5
-1 0 -1 -0.45 dielectric 1.5
```

Use command line arguments to obtain user-specified filenames. User is required to provide input filename that contains sphere data and output filename that contains image ppm data.

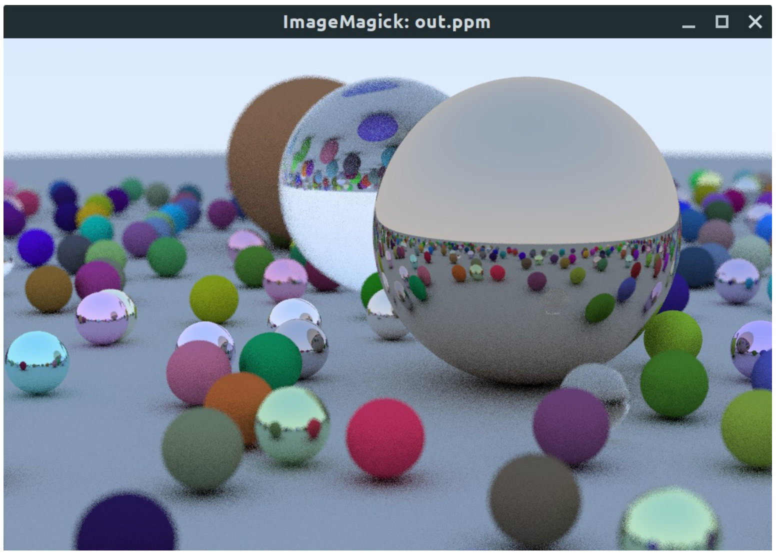
For example: `$ ./raytracing trace1.txt out1.ppm`

(2) Execution time



```
kendai  ~/NTUST_Course/Parallel_and_Computing/Homework/RayTracing  perf stat -e cache-misses,cache-references,instructions,cycles,branch-misses,task-clock ./raytracin
g
Elapsed: 2348.535925 seconds

 Performance counter stats for './raytracing':

     3,508,392,951      cache-misses              #   14.722 % of all cache refs
    23,830,751,838      cache-references          #   10.147 M/sec
10,752,561,623,114      instructions              #    1.39  insn per cycle
 7,732,613,381,421      cycles                    #    3.292 GHz
       212,453,378      branch-misses
    2348615.524593      task-clock (msec)         #    1.000 CPUs utilized

    2348.946570564 seconds time elapsed

kendai  ~/NTUST_Course/Parallel_and_Computing/Homework/RayTracing  
```

(3) Output of your program



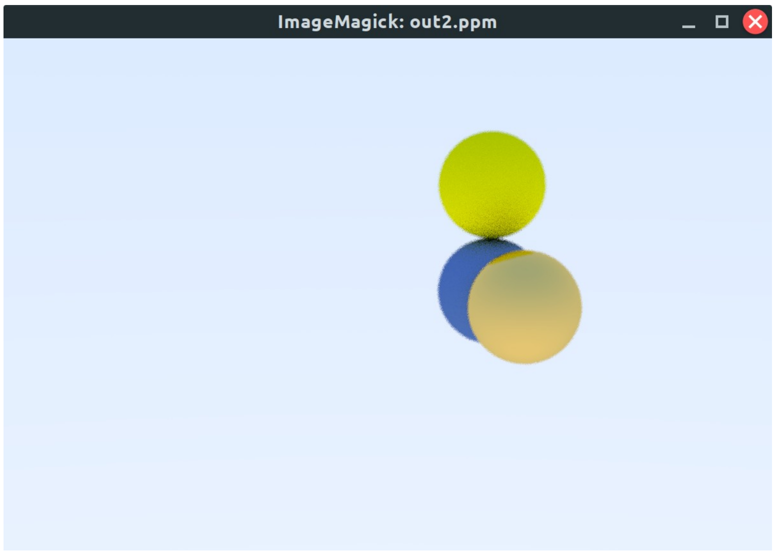
(4) Implementation

The code is placed in svn://140.118.105.174:36900/m10702130/RayTracing, we can use make to generate a executable file called raytracing.

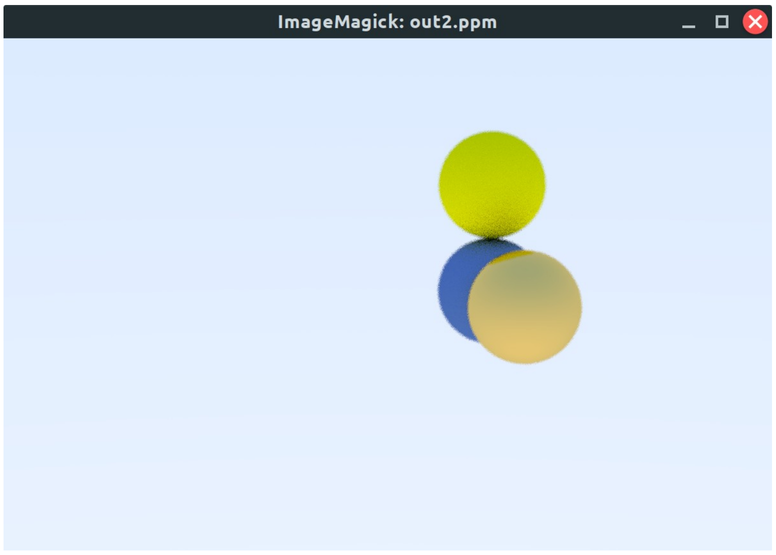
(5) Validations

Draw three spheres, the vector is (0, 0, -1), (0, 1, -1), (1, 0, -1), and the radius are 0.5, and the material are diffuse, diffuse and metal, respectively. Then we use diff tool to compare actual output vs. expected output.

My actual output:



Expected output:

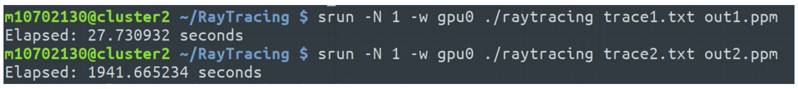


5. Parallel Implementation

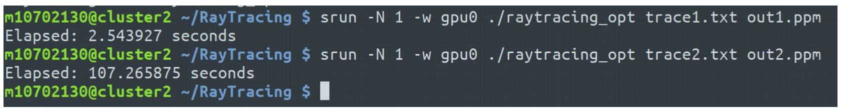
(1) Input of your program

The input and output file format same as serial version, but during compilation, we need to add OPT=1 option after make to compile parallelized version, and the output executable file called raytracing_opt.

(2) Performance



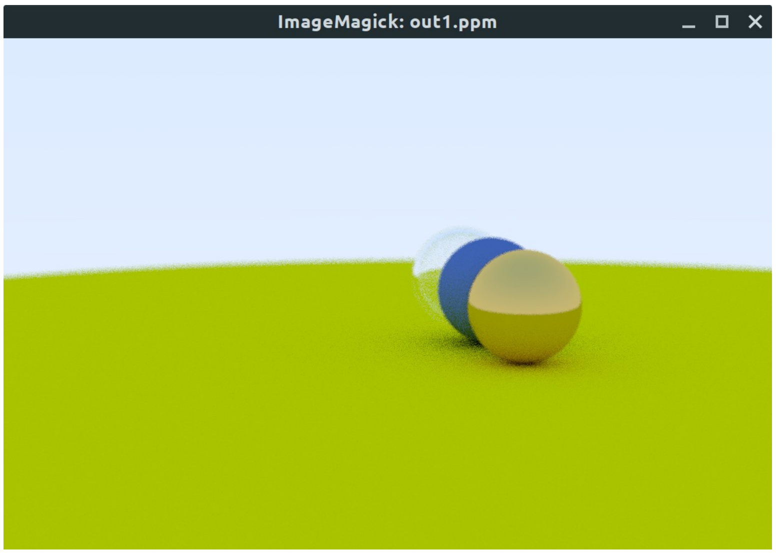
```
m10702130@cluster2 ~/RayTracing $ srun -N 1 -w gpu0 ./raytracing trace1.txt out1.ppm
Elapsed: 27.730932 seconds
m10702130@cluster2 ~/RayTracing $ srun -N 1 -w gpu0 ./raytracing trace2.txt out2.ppm
Elapsed: 1941.665234 seconds
```



```
m10702130@cluster2 ~/RayTracing $ srun -N 1 -w gpu0 ./raytracing_opt trace1.txt out1.ppm
Elapsed: 2.543927 seconds
m10702130@cluster2 ~/RayTracing $ srun -N 1 -w gpu0 ./raytracing_opt trace2.txt out2.ppm
Elapsed: 107.265875 seconds
m10702130@cluster2 ~/RayTracing $ 
```

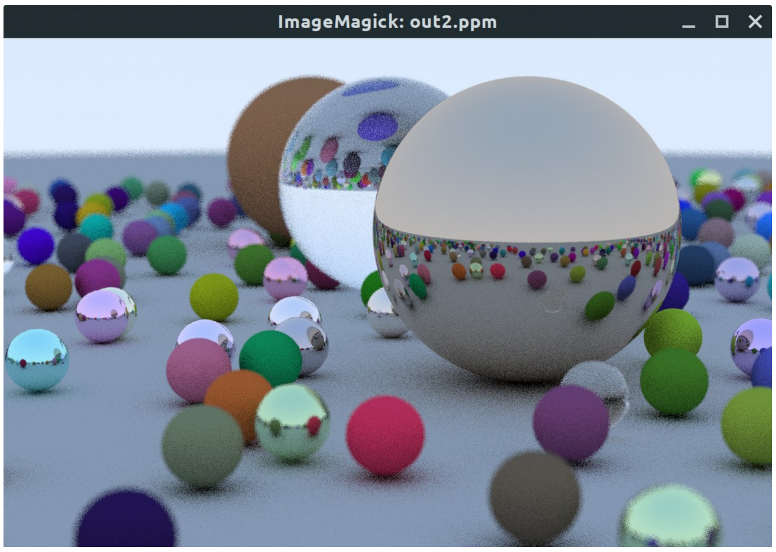
trace1.txt and out1.ppm

The size of image is 1200x800 and the image contains 5 spheres. The serial version runs 27.7 seconds and parallelized version runs 2.5 seconds. Speedup 1108%



trace1.txt and out2.ppm

The size of image is 1200x800 and the image contains 485 spheres. The serial version runs 1941.7 seconds and parallelized version runs 107.3 seconds. Speedup 1809.6%



(3) Implementation

The code is placed in svn://140.118.105.174:36900/m10702130/RayTracing, we can use make OPT=1 to generate a executable file called raytracing_opt and use make OPT=1 test to auto testing.

(4) Validations

Although the actual output and expected output look the same, but actually some pixels are different. For the image no jaggies along edges, we need to make the edge pixels are a blend of some foreground and some background. We can implement by averaging a bunch of samples inside each pixel.

So we need is a random number generator that returns real random numbers. For a given pixel we have several samples within that pixel and send rays through each of the samples. The colors of these rays are then averaged.

Here is the problem. When the thread gets the random number in different order, the number is different, so the result are also different. But it is not a big deal, because these two images still look the same.

6. Reference

[1] https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

[2] http://web.cs.wpi.edu/~matt/courses/cs563/talks/ray.html

[3] http://www.pbr-book.org/3ed-2018/Introduction/Photorealistic_Rendering_and_the_Ray-Tracing_Algorithm.html

[4] https://developer.nvidia.com/rtx/raytracing