Ray tracing on GPU

Contents

1. Introduction

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration) [1].

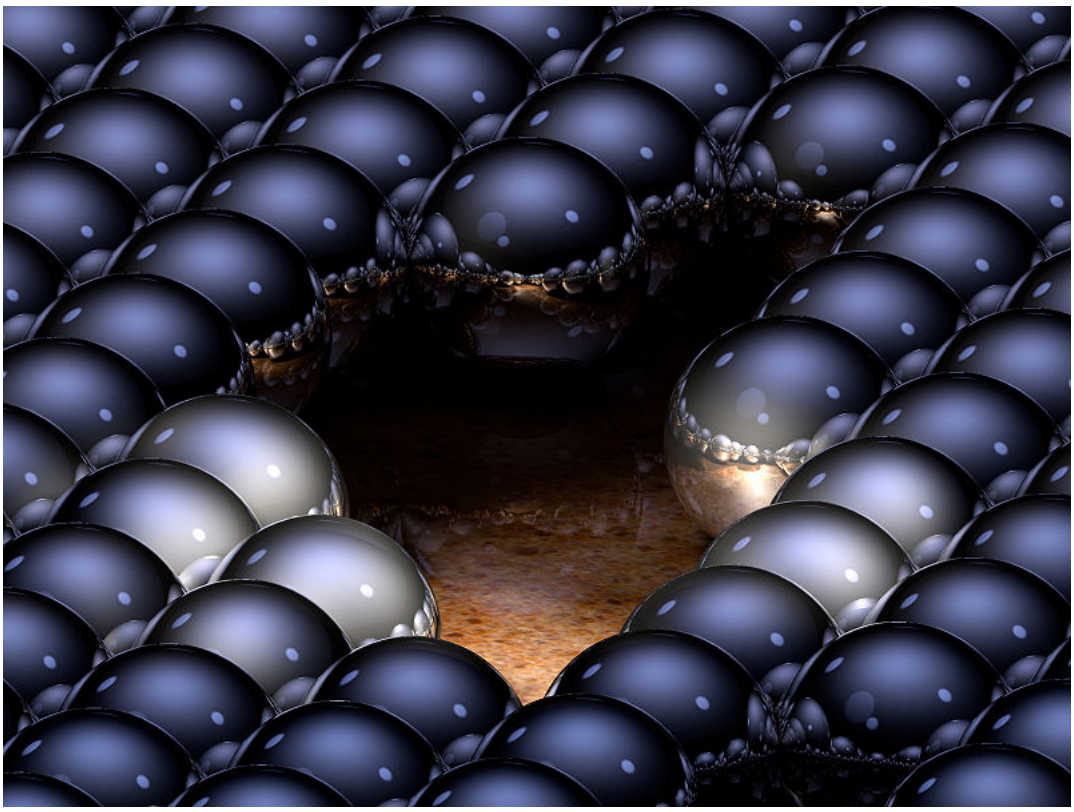

Figure 1. An image obtained using ray tracing

Ray tracing's popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. Relatively simple to implement yet yielding impressive visual results, ray tracing often represents a first foray into graphics programming. The computational independence of each ray makes ray tracing amenable to parallelization [3].

The ray tracing algorithm renders an image by casting rays into the scene. A ray is cast through every pixel of the image, tracing the light coming from that direction. When the ray hits an object, the light incident of that point is evaluated to compute the amount of light reflected back to the camera, casting more rays into the scene from this

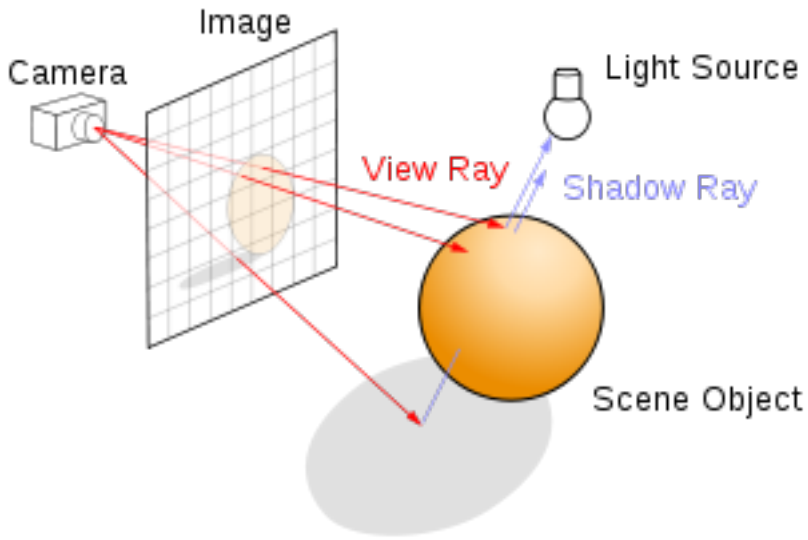

Figure 2. The ray tracing algorithm renders an image by casting rays into the scene (adopted from wikipedia.org)

point. Sampling the scene with many rays from each point of interest would be way to achieve interactive frame rates, so only the most important directions are sampled.

For diffuse surfaces, most reflected light will usually come directly from light sources, thus one "shadow ray" is cast towards each light source, to find whether the source is visible or not. If it is not shadowed, the light incident is used to compute the shading of the point of interest, according to the surface properties. The steps described above would render an image with shadows and surface shading, but to obtain light reflection and eventually refraction, we need to introduce recursion into the algorithm. If the surface is reflective, another ray is cast in the direction of ideal reflection. This ray is traced in exactly the same way as the primary rays (rays cast from the camera). The recursion stops when a maximal depth is reached, or the contribution factor drops below certain threshold.

All pixels of the resulting rendered image are calculated independently from each other. Thus, the ray tracing algorithm suits quite good for GPU implementation. Here is a video from GPU Technology Conference 2012 showing real-time ray tracing on Kepler GPU.

2. Task definition

Implement a simple ray tracing algorithm without refraction rays using GPU. The generated scene should consist of 5–10 spheres of different colors and 1 or 2 point-like light sources. Maximum depth of recursion is 5. All objects in the scene are not transparent.

3. Proposed method

The general application workflow should look as follows:

1. Generate 3D scene (parameters of 5-10 spheres and coordinates of light sources);
2. Copy all used data to GPU memory;

3. Render scene on GPU: compute the output image pixels color and shininess;

4. Copy the resulting pixels array back to host and save as BMP image.

The following simple method of scene rendering should be implemented on GPU:

```
TraceRay(ray, depth)
{
    if(depth > maximal depth)
    return 0;
    find closest ray object/intersection;
    if(intersection exists)
    {
        for each light source in the scene
        {
            if(light source is visible)
            {
                illumination += light contribution;
            }
        }
        if(surface is reflective)
        {
            illumination += TraceRay(reflected ray, depth+1);
        }
        if(surface is transparent)
        {
            illumination += TraceRay(refracted ray, depth+1);
        }
        return illumination modulated according to the surface properties;
    }
    else return EnvironmentMap(ray);
}

for each pixel
{
    compute ray starting point and direction;
    illumination = TraceRay(ray, 0);
    pixel color = illumination tone mapped to displayable range;
}
```

4. Implementation requirements

4.1. Input data

- The number of spheres (5–10);
- The number of light sources (1–2);
- The resulting image dimensions (800×600–1920×1080);
- The output image filename.

4.2. Output data

- The time of scene processing using GPU;
- The resulting image in BMP format.

4.3. Implementation

The program is required to work on Linux machine. The use of CURAND library for generating random 3D scene parameters is mandatory. The resulting image could be exported in BMP format, using many open-source libraries, for instance, EasyBMP [2]:

```
#include "EasyBMP.h"
...
BMP AnImage;
AnImage.SetSize(WIDTH, HEIGHT);
for (int i = 0; i < WIDTH; i++)
    for (int j = 0; j < HEIGHT; j++)
    {
        RGBApixel pixel;
        pixel.Red = pR[j * WIDTH + i];
        pixel.Green = pG[j * WIDTH + i];
        pixel.Blue = pB[j * WIDTH + i];
        pixel.Alpha = 0;
        AnImage.SetPixel(i, j, pixel);
    }
AnImage.WriteToFile(FILENAME);
```

5. Outcome

1. Getting familiar with CUDA applications development and CURAND library.

References

[1] Ray Tracing http://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[2] EasyBMP http://easybmp.sourceforge.net.

[3] T. Davis A. Chalmers and E. Reinhard. *Practical parallel rendering*. AK Peters, Ltd., 2002.