Monte Carlo Ray Tracer, TNCG15

Jakob Bertlin and Tobias Matts

February 16, 2018

Abstract

This report presents the implementation of a Monte Carlo ray tracer as part of the course *Advanced Global Illumination*. The purpose of the project was to use the theory covered in the course lectures and to apply it to computer graphics.

A big part of computer graphics is to replicate physical phenomena like forces and illumination; ray tracing is an attempt at the latter. The basic idea is to simulate light rays going into the scene, bouncing between different objects and to finally end up on the eye, creating and image. This can be described by a so-called rendering equation, but since in real life there is basically an infinite amount of light rays reflecting and refracting between objects an infinite amount of times, it is impossible to solve the equation precisely. Therefore, its is only a realistic goal to try to approximate a solution to the equation. Different types of ray tracing use different methods and tools to tackle the problem, but they all have the same goal: to create a as good as possible replication of the behavior of light in a scene.

The Monte Carlo ray tracer in this report was implemented from scratch using the programming language *C++*. The final application renders an image of a scene consisting of a hexagonal room with a diffusely surfaced sphere and a transparent sphere. The final image has the dimensions 1000x1000 pixels and took ... to render.

Contents

Chapter 1

Introduction

There are many ways to simulate light distributions in a scene. A popular example of an explanation to light distribution is the Rendering equation [1]. The rendering equation is an integral equation which in short says any point on a surface is illuminated by everything around it. This is described in the rendering equation as taking any point on a surface and creating a hemisphere on that point, with the hemisphere rotated along the normal of that surface. The theory is then that light that illuminates that surface point will enter the hemisphere from all directions, by then integrating over the entire hemisphere the total amount of light that arrives on the surface point can be calculated. The light is quantified as "flux" arriving at the surface point which contributes to the total radiance of the surface.

$$L(x \leftarrow \omega) = L_e(x \leftarrow \omega) + \int_{\omega 1} f^*(x_1, -\omega, \omega_1)\, L(x_1 \leftarrow \omega_1)\, d\omega_1$$

Figure 1.1: Rendering equation for the radiance arriving at point x from the direction ω.

However one important aspect about the rendering equation is that it is infinite, and can never truly be calculated, only approximated. This is because if we take any direction on a surface hemisphere, that direction contains flux from another surface point, and that other surface point has it's own hemisphere which is also taking flux from all possible directions. And this goes on and on and on infinitely, which is why the rendering equation is impossible actually solve. But even if it is an impossible equation to solve it can still be sampled and estimated in other ways which will, given infinite amount of time, will converge to the true result of the rendering equation.

Ray tracing is a method and an attempt to replicate global and physically based lighting. The final goal is to achieve photo realism, meaning that with enough computing power and time the image from the ray tracing algorithm ultimately converges towards realism. In more common words this means that eventually your ray tracing images will be indistinguishable from the real world. However the term ray tracing in itself is very ambiguous and doesn't really relate to any specific implementation or solution.

There are many different approaches and algorithms for ray tracing. A famous method for dealing with reflection and refraction is Whitted ray tracing. It uses the perfect reflection and refraction laws to accurately compute how a ray would bounce on a perfect mirror or completely transparent glass for example. The main idea in Whitted ray tracing is to keep reflecting and refracting rays infinitely until it finally collides with a diffuse object. When it collides with a diffuse object it will check if that point of the object is visible by a light source. Checking for visibility is done by sending a handful of shadow rays towards the light source from the point on the object. If at least one of the shadow rays successfully hit the light source without colliding with anything else along the way the point on the object is considered to be illuminated by the light source. But as you may have noticed this algorithm

doesn’t actually calculate light distribution by sending rays from the light source, as in the real world. Instead it shoots rays from camera and and then into the scene to see if the intersection points on that ray path can be illuminated by a light source. This is called backwards tracing. Whitted ray tracing works very well in environments where objects are either mirrors, fully transparent or completely diffuse. But in the case where that is not true Whitted ray tracing by itself is more or less useless in terms of achieving a realistic image.

Radiosity is also a technique used to simulate lighting. But instead of sending out single rays through each pixel for example, which does the color computations for that pixel, it computes light based on forward tracing in contrary to Whitted ray tracing which is backward tracing. Radiosity first splits every geometry into small surface patches. Then the algorithm sends "light" from the light source to all these surface patches that are visible from the lightsource. And then every surface patch that was hit by that light then continue to distribute light to all other surface patches visible from that patch. This is then repeated for a set amount of times. Now a camera shoots rays through pixels which becomes the color of the surface patch that is hit by that ray. As nothing is randomized in radiosity computations and instead everything visible is accounted for this gives no noise in the result.

As radiosity does it’s computations from the light source and not from the viewers perspective you can move the "camera" around in the room without having to do any new computations on light distribution. However one crucial thing, which is very hard to simulate with radiosity alone, is specular reflection. This is because specular reflections are viewer dependent and as we have covered previously, radiosity isn’t. Whitted ray tracing may therefore be preferred for scenes which rely on objects with mirrors and transperent objects. Radiosity computations are also very demanding in both memory and compute because of the fact that it divides everything into small surface patches which all contribute to each other, but you only need to do the computations once if no objects and light sources move, or if the scene is simply completely static.

Chapter 2

Background

This paper is based on the implementation of a Monte Carlo ray tracer using the programming language C++. The following chapter describes the different classes of the program and how different techniques have been translated into code.

2.1 Basic Code Outline

To better understand the raytracing techniques described later on in this chapter, the basic concepts of the code will be presented in this section.

2.1.1 Vertex

Vertices are simply geomtrical points in the 3D space. These are used throughout the entire ray tracer to specifiy intersections points, directions or ray paths for example.

2.1.2 Pixel

What is ultimately presented to the user with a raytracing algorithm is a pixel matrix, or simply an image of the final scene. Each ray shot into the scene is used to compute the final color of a specific pixel. In the 3D scene there is a pixel plane, which is a virtual plane that is meant to represent the pixel matrix that is shown to the user. Each ray path starts from the camera/eye position through a point in pixel plane and into the scene where it eventually encounters some kind of object.

2.1.3 Ray

The ray is (for obvious reasons) a central part of raytracing, since its task is to represent the paths of photons travelling from a light source and between objects in the scene. A ray is defined by its start and end points (vertices) in the 3D space along with a color vector.

2.1.4 Objects

All objects that are geometrically suitable, are defined as a mesh of triangles. Each triangle consists of three vertex points, a normal vector and a color vector, where the normal represents which direction

the surface is facing. When a ray intersects with an object the normal direction and surface properties determines how the ray will continue its path.

The normal vectors for triangles are computed by taking the cross product between two triangle sides spanned by vertex points (see 2.1), thus resulting in a vector orthogonal to the triangle (see 2.1).

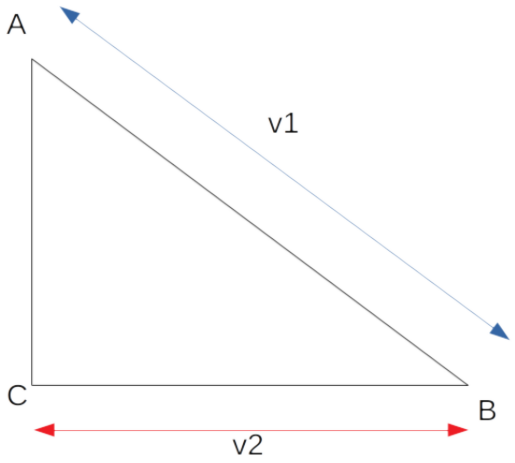

Figure 2.1: A triangle with vertex points *A*, *B*, *C* and side vectors v_1 and v_2

.

$$\begin{cases} v_1 = A - B \\ v_2 = C - B \end{cases} \Rightarrow n = |v_1 \times v_2| \Rightarrow \hat{n} = \frac{1}{|n|} n \tag{2.1}$$

Another important function is the ray intersection function, which determines if an object is intersected by a ray, and if so, it returns the intersection point. The function is an implementation of the Möller-Trumbore [3] intersection algorithm for triangles which, given three corner points and a ray, calculates the intersection point between the ray and the triangle spanned by the three points. What makes the algorithm beneficial for this task is the fact that it can compute the intersection point without the need for precomputing the equation for the plane containing the triangle. This saves memory and makes the method fast, which is always a good thing when dealing with computationally heavy tasks like raytracing.

For spheres a simple geometrical computation is done to check whether there is a point on the ray path that exists on the sphere aswell, see figure 2.2.

When a ray intersects with a sphere, it does so in two points on its surface. The first point will for this reasoning be called *P* and the other *P'*. The coordinates for these two points can be calculated by first determining their distance *t0* and *t1* from the ray origin respectively. Since the center of the sphere is known along with the ray origin, we get $L = C - O$. The direction *D* of the ray is already known, therefore by taking the dot product of *L* and *D*, L is projected onto D, thus giving: $t_{ca} = L \cdot D$. We now have two sides of the triangle $t_{ca}Ld$ and using the pythagorean theorem d can be determined: $d = \sqrt{L^2 - t_{ca}^2}$. The last variable needed to determine the intersection points is t_{hc}, which after determining d can be calculated with another use of the pythagorean theorem: $t_{hc} = \sqrt{radius^2 - d^2}$.

When the variables t_{ca} and t_{hc} have been determined *t0* and *t1* can be derived from them: $t0 = t_{ca} - t_{hc}$, $t_{ca} + t_{hc}$ and consequently the intersection points $P = O + t0 * D$ and $P' = O + t1 * D$.

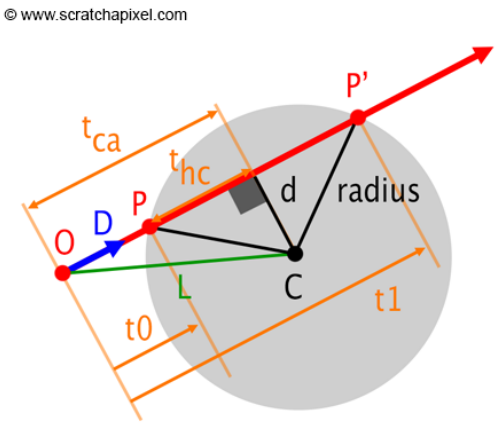

Figure 2.2: Sphere intersection [2]

2.1.5 Scene

The ultimate goal of raytracing is to create a realistic image of a synthetic scene. To do so, the scene must first be defined. For this particular case the scene is a hexagonal room (six walls, a roof and a floor), 2.3, that consists of multiple instances of objects. The room also contains a diffusely surfaced spehere, a transparent sphere and a light source. All the physical objects (in this case the triangles and spheres) are contained in a vector for easy accessibility later in the process, while for this particular scene, the only light source is treated separately.

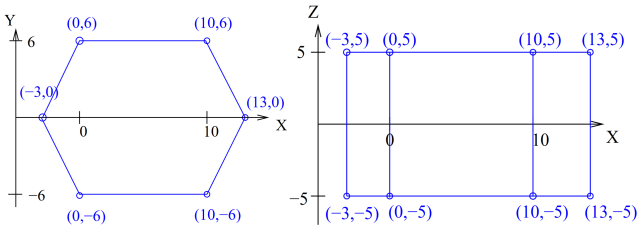

Figure 2.3: The room vertex points in the xy- and xz-plane.

2.1.6 Camera

Producing a two-dimensional image of a three-dimensional scene is essentially what a camera does when a picture is taken. The camera can be placed anywhere in the scene, but for the implementation in this report only two different camera positions were used.

In front of the camera there is an image plane which is defined by four vertex points. The plane consists of pixels, which we want to find RGB values for. To do so, one or more rays are sent through each pixel of the image plane into the scene. Then each ray is checked for intersection with each surface in the scene. If a ray intersects with a surface a shadow ray is shot towards the light source to check if the intersection point is directly illuminated by it. See 2.4.

The radiance returned by each intersected surface depends on contribution from other surfaces in the scene. The radiance that falls onto a surface at a certain point can be described by the rendering equation 1.

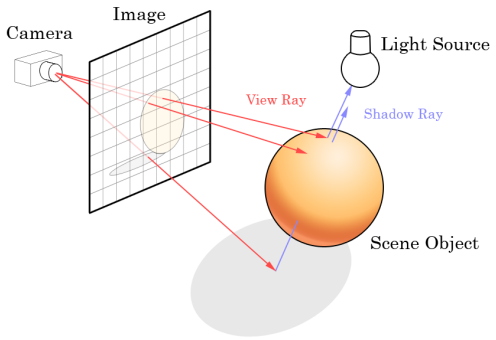

Figure 2.4: The figure illustrates how rays are sent into the scene.

2.2 How the Monte Carlo Raytracer Works

Monte-Carlo ray tracing is an example on how to solve the rendering equation using path tracing. It is similar to Whitted ray tracing in the sense that it launches rays from the camera, reflects them and then uses the information to calculate the color of pixels from which the rays were shot from. But basic Monte-Carlo path tracing uses only the rendering equation to handle lambertian reflectors, which are completely diffuse surfaces. When a ray is first shot from the camera it creates a hemisphere around the surface point it hits facing the direction of the surface normal. But instead of integrating over the entire hemisphere as seen in the rendering equation, one or more random directions on the hemisphere are sampled. Since the directions are randomized the result will be an unbiased sample but may produce noisy results if not enough rays are sampled. Monte-Carlo path tracing can be extended to include many more different techniques such as the perfect reflection and refraction laws as seen in Whitted ray tracing.

When Monte-Carlo path tracing samples a direction on the hemisphere it uses the BRDF corresponding to that surface to sample a new direction. The BRDF can be described as the function which holds information about the correlation between ingoing and outgoing rays. For example, in a perfect mirror the BRFD is simply the perfect reflection law. And in a lambertian reflector the BRDF says the the outgoing ray direction is completely random, since a lambertian reflector reflects light equally in all directions.

One problem with using Monte-Carlo path tracing in only the way that has been discussed here is that rays will only contribute to the pixel color value if it eventually hits a light source that feeds radiance into the ray path. And since few rays will actually hit a light source, unless we send out an unmanagable amount of rays from the camera, the image will be mostly black. To handle this we check for every diffuse surface that is hit by a ray if it is illuminated by the light source and then feed flux into the ray path. This is done by sending shadow rays towards the light source and check if nothing obstructs the view to the light source, just like in the example of Whitted ray tracing. Now, suddenly every ray we send will most likely contribute with color to the final image. Thus avoiding to do time consuming computations for rays that don't contribute.

But sending shadow rays to check for illumination every time a ray hits a surface point is very inefficient and increases rendering time. There are a few ways to solve this issue, and one of them is to introduce photon mapping. Photon mapping in short consists of sending photons out into the scene and then storing the positions where they intersect with the geometry in the scene. And now we can instead approximate if a given surface is illuminated or not by checking if there are nearby photons stored in the scene. This does actually mean that the image will not converge towards realism since a bias is introduced, but any amount of accuracy is still achievable by simply increasing the amount of

photons sent out into the scene. In the same manner we can also easily check if an object is in shadow. This is done by also storing "shadow photons", which are the intersections of the photons if they were to continue through the first object they intersect with. So if a point is surrounded by shadow photons it can be assumed that the point is indeed in shadow.

However, for this report there is no photon mapping implemented. But the ray tracer is planned to be further extended with this at a later point.

Chapter 3

Results and Benchmarks

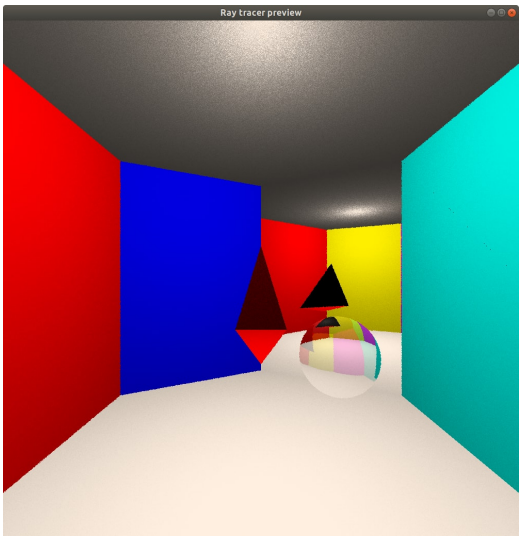

Figure 3.1: Scene directly illuminated, no reflection on diffuse surfaces. Wall in the back right is a mirror.

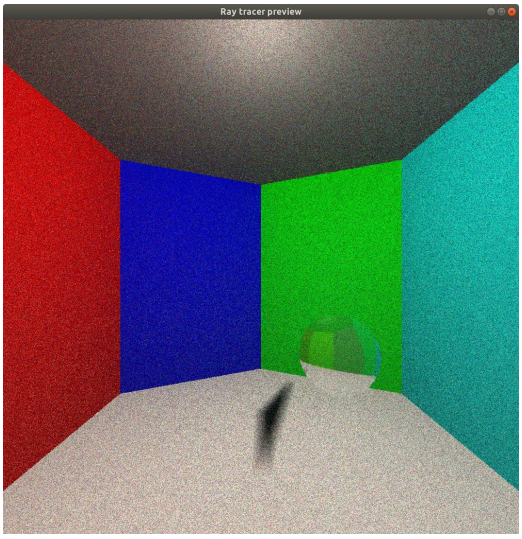

Figure 3.2: 1 ray per pixel, 15 shadowrays, 5 reflections on average

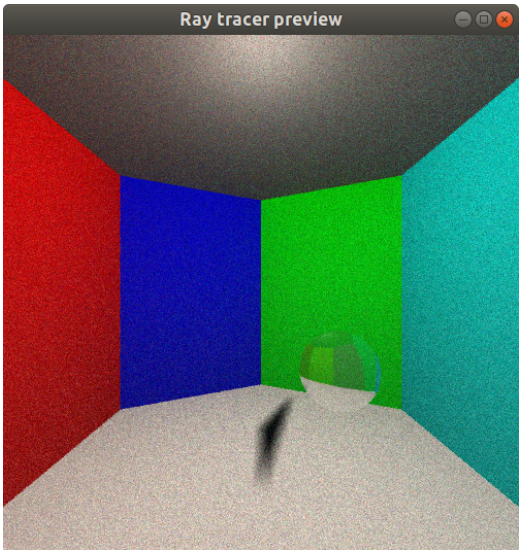

Figure 3.3: 5 rays per pixel, 10 shadowrays, 4 reflections on average. 11 minutes render time

Chapter 4

Discussion and Conclusions

Bibliography

[1] James T. Kajiya, *Rendering Integral Equation*, California Institute of Technology 1986, hämtad: 2017-10-25
http://www.dca.fee.unicamp.br/ leopini/DISCIPLINAS/IA725/ia725-12010/kajiya-SIG86-p143.pdf

[2] Scratchapixel. *Ray-Sphere Intersection*[Internet]. 2016[cited 2017-12-18]. Available from: https://www.scratchapixel.com

[3] Möller, Tomas, and Ben Trumbore. "Fast, minimum storage ray/triangle intersection." ACM SIGGRAPH 2005 Courses. ACM, 2005.

[4] *OpenGL*, The Khronos Group 2017, hämtad: 2017-02-15
https://www.opengl.org/