Ray tracing with Compute Shader

Philip Pedersen* Tobias Elinder†

Lund University
Sweden

Abstract

For the project a real-time ray tracer was implemented, we wanted a scene with multiple physics-driven spheres and other objects in order to capitalize on the fact that it's rendered in real-time. We began simple and added concepts and complexity over time, resulting in a balanced and in our humble opinion good looking ray tracer.

1 Introduction

The scope of the project was to create a real-time ray tracer demo in OpenGL with simple physics on the rendered objects. Ray tracing is a method to generate realistic looking images by shooting "rays", representing light, through pixels in an image plane. The pixel then get the color of the object which are hit by the ray. The result is physically more accurate than rasterization, and allows for accurate real-time reflections, as opposed to cube mapping, but with a higher computational cost. We use OpenGL Compute Shader to render our scene to a single texture, each thread handling one pixel at a time. This allows us to render it to a full screen quad in OpenGL. Compute Shader allows parallel computations of rays and was used to speed up the process by allowing us to render the scene on the GPU. The scene used in the demo consists of spheres and boxes of different materials and sizes. Simple physics such as velocity, friction and collision are applied to the objects, which positions are calculated in real-time.

2 3D Graphics Project

The project is written in GLSL and C++ using the OpenGL API. The classes handling the physics and the window is written in C++. The ray tracing is done in the Compute Shader to allow us to do the computations on the GPU. A Compute Shader is a general shader without well-defined inputs and outputs and are used for arbitrary computations. The code in the Compute Shader is written in GLSL, OpenGL Shading Language.

2.1 Scene and objects

The scene contains 6 planes which are defined by a point and the normal which is sufficient to calculate if a ray will hit the plane. A reflectivity index and a color of the plane is used to describe the material of the plane. The roofs and the floors color is computed with simple procedural generation. There's two types of objects in the scene, spheres and boxes. Spheres are defined by a centre point, the length of the radius and the material of the sphere as the color, a diffuse value, a reflectivity value and refractive index. The box are defined by two end points, a color and a reflectivity value.

We use a pinhole camera, sitting in a fixed spot looking straight at the scene. The camera consists of a point and an image plane.

2.2 Physics

Our scene has two main components that are almost totally separated, the ray-traced rendering, and the physics of the objects in the scene. The physics are all calculated in C++. We give each sphere a velocity vector which is recalculated each frame, when it can potentially collide with other objects. The planes and the box are static in the sense that they have no velocity vector and are unable to move.

2.3 Ray tracing

The ray tracing method works by tracing a ray from the camera position through a image plane as is illustrated in fig1. and test for intersections. If the ray hit an object the ray is split into two rays, a reflected and a refracted. The ray is reflected around the normal for specular materials and is split for glossy materials, this as well as the refraction is illustrated in fig.3. At the intersections a ray towards the light is traced to decide if the impact point is shadowed.

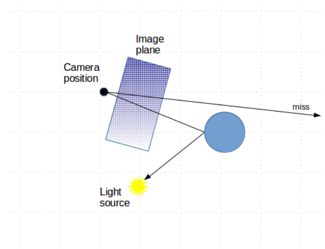

Figure 1: Tracing rays from the camera

In order to allow us to render our image in real-time without visual artifacts, we decided early on a backward ray tracer. This means that we (not entirely correctly) trace the rays backwards from our camera back into the scene which guarantees a value for each pixel without noise but doesn't give as accurate results as a converged forward ray tracing, which fires the rays from the light source and recognizes rays that eventually ends up in the camera.

Our ray tracer follows the basic, premise of a Whitted[Whitted 1980] ray tracer (shadow, reflection, refraction). With one significant limitation. Since GLSL does not currently support recursion, we have to somehow modify our functions. Luckily non-branching recursion can easily be interchanged with a while loop as described by [Lawlor 2012]. This solves our problem as long as we don't have to split/branch the ray, unfortunately we need to do that in order to render a simultaneously refractive and reflective material, for example glass. As a crude compromise, we have created an additional, almost identical function to our first (the one containing the while-loop), in order to allow a ray to split once, making it possible to render our glass material correctly on the first bounce that touches the material, in succeeding bounces, we only either refract or reflect, depending on which factor is strongest for the specific material. This is obviously not correct, but it's almost impossible to notice the small error that occurs.

In order to render more noisy surfaces we allow rays to bounce irregularly from glossy surfaces, this means that we add a pseudo-random offset to the reflected/refracted ray. When only reflecting a single ray the surface looks very noisy and "grainy" so we decided to use the same method as described above, allowing the rays to split one time, by firing multiple rays from the surface of a glossy material and take the average of the result. This works well as long as the number of rays fired is large, and we noticed that if we fired 30 rays from the surface it looked perfect, but this number was not

*e-mail: dat12ppe@student.lu.se
†e-mail: dat12tel@student.lu.se

feasible performance-wise, so in the end we settled with 5 rays, which unfortunately leaves some visible noise if you look close enough. For performance reasons, we have limited ourselves to specular and glossy materials, there are no diffuse materials in our scene, since it needs too many rays in order to look accurate.

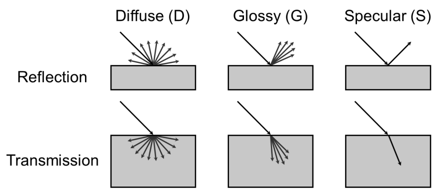

Figure 2: Source: https://elementalray.files.wordpress.com/2013/01/dgs.png

3 Result

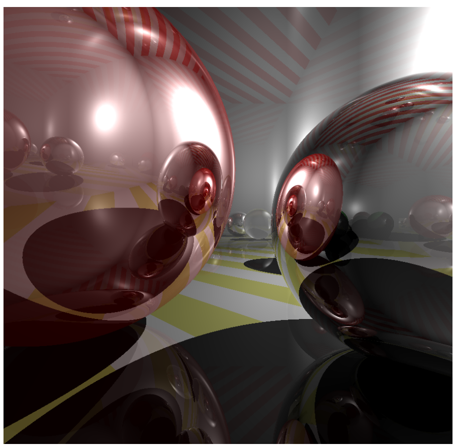

Figure 3: Two highly reflective surfaces

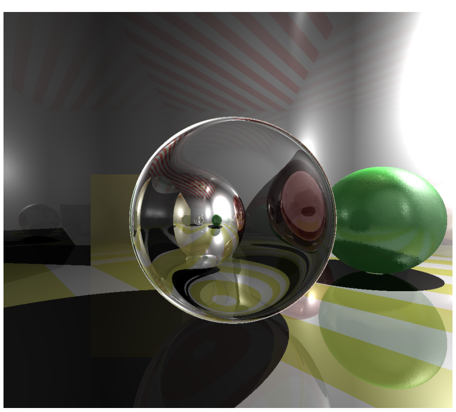

Figure 4: A refractive Surface

4 Discussion

Over the last three weeks we have experimented with many different aspects of ray tracing, our goal, to render the scene in real-time on the GPU made a lot of the resources online hard to use, since they were heavily intertwined with the concept of recursion. In the end we were able to achieve far better performance than we had

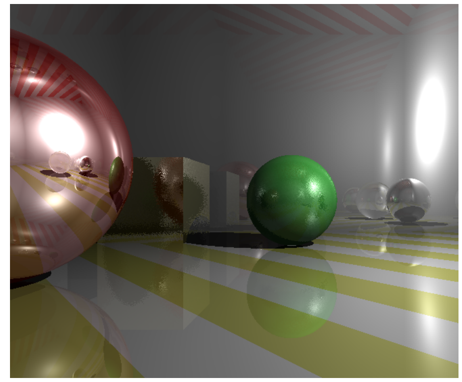

Figure 5: A noisy (glossy) surface

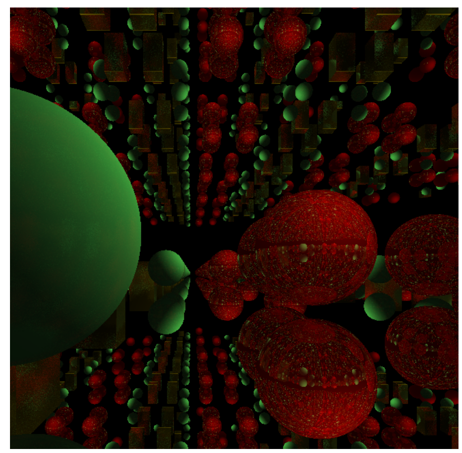

Figure 6: Every plane given maximum possible reflection, with maximum ray bounce depth at 15

hoped for, with a demo containing several objects that runs very smoothly (>100fps) on modern hardware (we used a Nvidia GTX 1070).

Our biggest hurdle was the lack of recursion in GLSL, which made the shader code messier, but in the end probably didn't have a great effect on the final rendering, and forced us to think about performance and the exact number of rays fired all the time.

It seems obvious that a backwards ray tracer is the only feasible way to make it run in real-time, at least for now, and because of that we were not able to use forward ray tracing / photon mapping, which would certainly have been interesting.

Optimizations are mainly limited to better ways of tracing the geometry in the scene, with more effective methods that may exist, and avoiding testing every ray against every piece of geometry in the scene as we do now. Obviously it's possible to lower the number of rays fired from the glossy surface, but not without significant visual degradation.

Compared to rasterization we get some very realistic looking materials and the reflections are better then using cube mapping. If a lot of objects and more complex objects or materials was to be used it might not have been good enough performance wise to render in real-time. In cases where complex scenes are to be rendered in real-time, rasterization would be to prefer with its rendering speed, for offline rendering ray tracing is a relevant method since its lack of speed won't be such a big disadvantage.

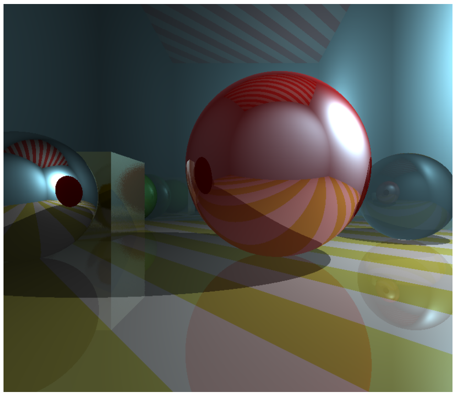

Figure 7: maximum ray bounce depth limited to 2

References

LAWLOR, D. 2012. Recursive raytracing, on gpu hardware without recursion.

WHITTED, T. 1980.