DH2323 Computer Graphics and Interaction
Recursive Backwards Ray Tracing
Project Report

Ramona Häuselmann
KTH Royal Institute of Technology
ramonaha@kth.se

ABSTRACT

We present the implementation of a simple recursive backwards ray tracing algorithm based on the algorithm introduced by John Turner Whitted in 1980. In the scope of the project we look into details and improvements of a basic ray tracing algorithm to explore its capabilities and limitations.

The system is capable of rendering triangles and spheres with glossy, diffuse, specular and transmissive as well as textured materials. It uses a bounding volume hierarchy for increased performance, sub sampling to reduce aliasing effects and an area lighting model for soft shadows.

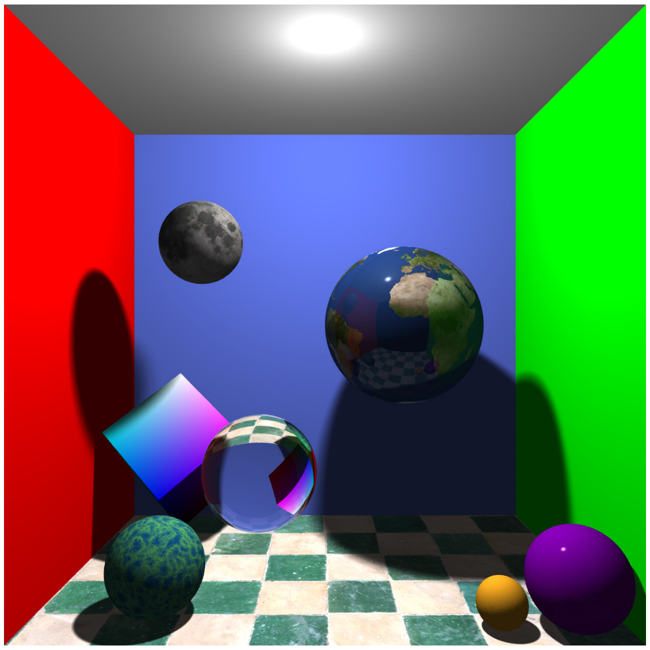


Figure 1: Cornell box scene. Rendered with 4 super samples per pixel and a rectangle area light for soft shadows.

Figure 1 shows the final result. The scene consists of a glossy, specular sphere with earth map texture, a diffuse sphere with moon texture, a diffuse cube with rainbow texture, a glass sphere, a purple, specular sphere, a yellow, diffuse sphere and a diffuse sphere with procedural texture (green and blue)

Source Code:
The source code and data can be found at https://github.com/rhaeus/basic_raytracer.

1 INTRODUCTION

In the field of computer graphics there are many different techniques for generating realistic images. In this project we explore one early ray tracing approach that was introduced by John Turner Whitted in 1980. We based our implementation on his ideas and added several improvements.

The implemented raytracer is a so called backwards raytracer. In nature light is emitted from every light source and bounces infinitely on objects until it finally reaches our eye. Given that most of the rays never reach our eye it would be wasteful and computationally not feasible to trace each ray starting from the light source. A backwards raytracer follows the inverse approach: It follows the rays backwards, starting from the eye into the scene. This reduces the computational complexity but also has the disadvantage that it can not account for global lighting effects (e.g. indirect lighting, caustics) since only rays that reach the eye are considered in the calculations. In this project we want to explore the basic techniques to build a foundation for extending the raytracer in future work with one of the many approaches that take global lighting into account, e.g. Photon Mapping, Path Tracing or Radiosity.

2 GENERAL ALGORITHM

The raytracer calculates a primary ray for each image pixel starting at the camera position. It then calculates if the ray intersects any of the objects in the scene. If an intersection occurs, the algorithm calculates the color at this position, depending on the object material by using the Phong illumination model. For shadow calculation the raytracer casts a shadow ray to each light source to check if the surface point receives direct lighting. If the surface of the intersected object is reflective and/or refractive the raytracer calculates the reflection and refraction rays using Snell's law as well as the amount of reflection and refraction based on the material properties by using the Fresnel equations [4]. This process is repeated recursively until the ray hits a diffuse surface without reflection or refraction, misses the scene or the max recursion depth is reached.

3 LIGHTING AND COLOR CALCULATION

For illumination and coloring we use the Phong illumination model [6]. The color of an object surface is calculated as

$$\begin{aligned} color = {} & ambientColor \\ & + diffuseAndSpecularColor \\ & + reflectiveAndRefractiveColor \end{aligned} \tag{1}$$

The ambient color term adds a small constant illumination to every object. Without this term surfaces without direct lighting would be pitch black since we don't use global illumination.

Let N be the surface normal, L the light incidence direction, R the direction of light reflection, V the viewing direction and p represents the shininess of the surface. Then the color of any surface point can be calculated as follows:

$$\begin{aligned} ambientColor &= surfaceColor * 0.2 \\ diffuseColor &= surfaceColor * lightColor * \langle N, L\rangle \\ specularColor &= lightColor * \langle R, V\rangle^p \end{aligned} \tag{2}$$

The surface color, N and p are determined by the object and its material. The reflective and refractive color components are calculated by casting rays in reflection and refraction direction.

4 TEXTURE MAPPING

The system supports three different ways of texturing:

- texture color mapping using a texture file
- procedural texture color generation using Perlin noise
- procedural normal generation using Perlin noise

4.1 UV Mapping

Sphere mapping: To texture a spherical object we need to map the 3D surface coordinate (x,y,z) to a 2D texture coordinate (u,v). To achieve a sherical mapping that does not suffer from distortions we use spherical UV mapping [11] and the following equations.

$$u = 0.5 + \frac{\arctan(d_x, d_z)}{2\pi}$$

$$v = 0.5 - \frac{\arcsin(d_y)}{\pi}$$

d represents the surface normal at point (x,y,z) on the sphere. We use those coordinates to lookup the color value in the color map file.

Triangle mapping: Each triangle vertex has a corresponding texture coordinate. To calculate a texture coordinate at any point on the triangle we have to interpolate between the vertices. For interpolation we use barycentric coordinates [5].

Let v_0, v_1, v_2 be the vertices of the triangle and pos the 3D position of which we want to calculate the color value. t_0, t_1, t_2 are the 2D texture coordinates of each vertex. We first express pos in terms of v_0, v_1, v_2:

$$pos = \alpha * v_0 + \beta * v_1 + \gamma * v_2$$

Then we can use those coefficients for interpolating between the vertex texture coordinates to obtain the 2D texture coordinate tex as

$$tex = \alpha * t_0 + \beta * t_1 + \gamma * t_2$$

4.2 Procedural texture generation

Perlin noise [7] is a gradient noise that produces very natural looking structures. It is therefore widely used in computer graphics. By using variations of this noise function we can procedurally generate various forms of textures [9].

Figure 2 a) shows the pure Perlin noise. With the Perlin noise function we generate a noise coefficient that we can use to mix two colors (black and white here) together.

$$color = coef * color1 + (1 - coef) * color2$$

If we add multiple noise patterns of different scale together, we can generate a turbulent structure as shown in Figure 2 b).

$$coef = \sum_{l=1}^{L} \frac{1}{l} * perlin(l * a * x, l * b * y, l * c * z)$$

$a = b = c = 0.05, L = 10.$
By modifying L, a, b, c we can generate marble- or wood-like structures as seen in Figure 2 c) and d).

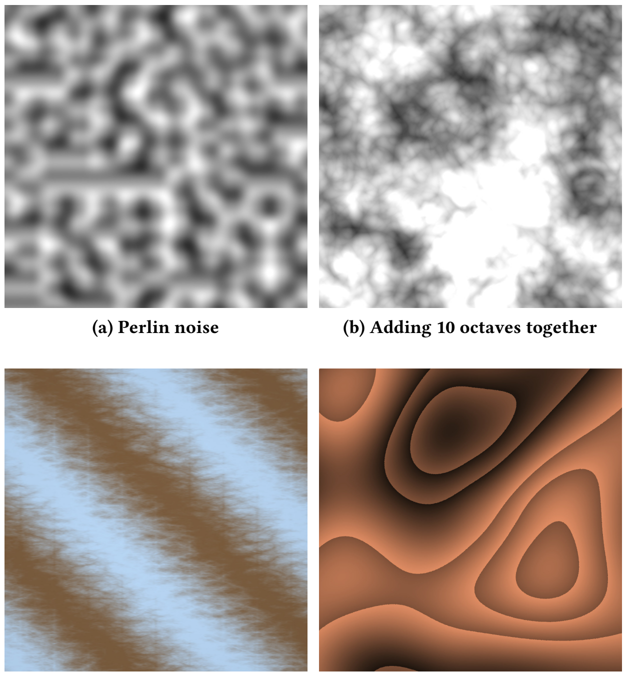

(a) Perlin noise (b) Adding 10 octaves together
(c) Marble-like structure (d) Wood-like structure

Figure 2: Perlin noise variations

4.3 Procedural normal mapping

We can use Perlin noise to procedurally modify the surface normals of the objects. This makes it possible to generate a look of bumpy surfaces even though the mesh of the object is not changed. But this effect also has its limits. Since it is only affecting the surface normals while calculating and lighting the surface, it has only an effect if there is direct lighting. Also the bumpy surface is not visible in the cast shadow. To achieve this we would have to change the mesh and/or use displacement mapping. Figure 3 a) shows a bumpy, matt surface. If we use a shiny material the modified surface normal affect the reflected light as seen in Figure 3 b).

4.4 Material collection

Figure 4 shows a collection of all materials the system offers.

First row:

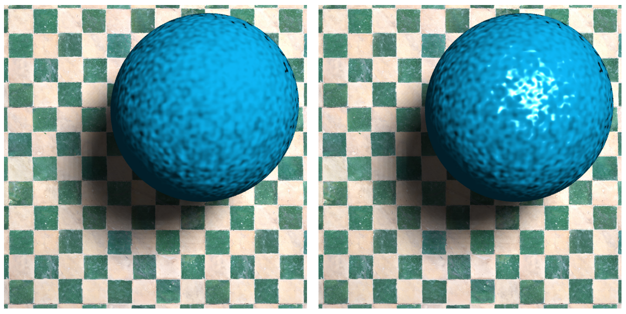

(a) Bumpy, matt surface (b) Bumpy, shiny surface

Figure 3: Bumpy surface

- flat, purely diffuse sphere
- flat, purely diffuse sphere with earth texture mapping
- bumpy, purely diffuse sphere

Second row:

- glass sphere with a slight red tint
- colorless glass sphere
- bumpy, colorless glass sphere

Third row:

- shiny, reflective sphere
- shiny sphere with moon texture mapping
- bumpy, shiny sphere

Forth row:

- shiny, bumpy sphere with marble like procedural texture
- flat, purely diffuse sphere with procedurally generated texture
- flat, purely diffuse sphere with wood like procedural texture

5 IMPROVEMENTS

5.1 Bounding Volume Hierarchy (BVH)

In order to render the scene, the ray tracing algorithm must calculate if a ray intersects with an object. The performance of those intersection tests limit the speed of the ray tracer.

A naive approach is to iterate over all primitives (triangles„ spheres) for each ray and perform the intersection test. This results in a lot of unnecessary intersection tests because most rays only intersect with a few primitives. To improve performance we use a **B**ounding **V**olume **H**ierarchy data structure [8]. This data structure hierarchically divides the space into smaller sub spaces, represented by an axis-aligned bounding box (AABB). Each primitive is stored in a corresponding container. The root container of this tree represents the entire scene space. To perform intersection tests we recursively check intersections with the AABB of the sub spaces and only descend further if this bounding box is hit by a ray. This way we can entirely discard many intersection tests with the primitives stored in the sub space if the bounding volume is not hit by the ray.

We use a BVH with a max depth of 16. Each space can be divided into 8 sub spaces and we allow for a maximum of 4 primitives in each space before it is split up. The achieved speed up is shown in

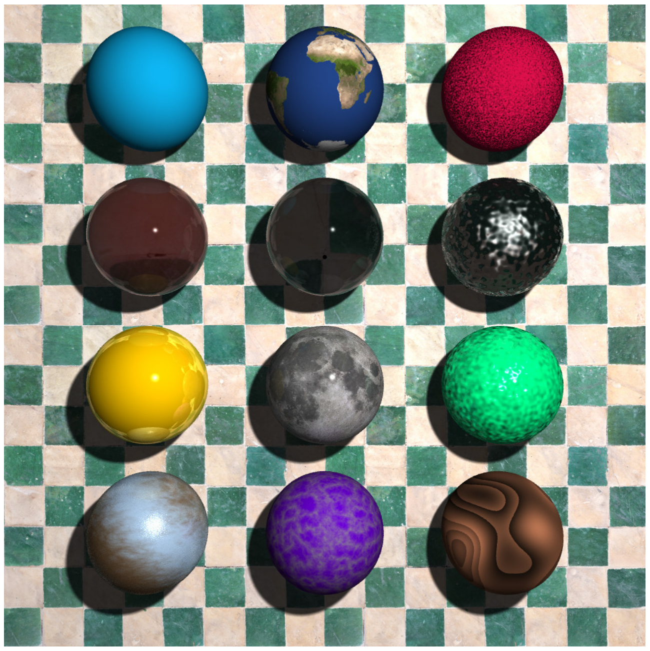


Figure 4: Material showcase. Rendered with 4 super samples per pixel and a rectangle area light for soft shadows.

Table 1. We rendered the scene shown in Figure 1 with and without the use of the BVH. The scene consists of 27 primitives and was rendered with a resolution of 1200x1200 pixels. We use 4 sub samples per pixel and 100 samples for the area light. A total of 6,637,052 rays were cast. We can see that the use of the BVH speeds up the algorithm by a factor of ca. 1.8.

Table 1: BVH Performance

	Without BVH	With BVH
ray triangle tests	13,264,817,544	1,835,448,170
ray triangle hits	565,645,299	565,645,281
ray sphere tests	3,617,677,512	1,240,731,817
ray sphere hits	97,583,742	97,583,742
AABB tests	-	9,037,706,495
abort early (AABB miss)	-	6,296,976,881
render time	540s	286s

5.2 Soft shadows (area light)

When lighting a scene the simplest approach is to use a point light source, meaning the light is emitted at a singular point in space. This is of course not very realistic since every light source in the real world has some expansion. Point light sources lead to very sharp shadows as seen in Figure 6 a) and are not very realistic. To render softer shadows we can simulate a light source that has extents. In this project we used a rectangle light source. Instead of sampling a single point for the lighting calculation, for an area light we would now have to integrate over the entire light source

area [3]. Since this is computationally very intensive we can use Monte Carlo integration to solve the problem:
We can approximate the light intensity at every point in the scene by sampling N points on the light source area. We then calculate if this point casts a shadow on the current scene point. Every sample point is weighted by 1/N. The final color of the scene point is then given by

$$resultingColor = color * lightIntensity$$

$$lightIntensity = \frac{\#pointsHitByLightRay}{N}$$

color is calculated using the Phong illumination model. The pattern in which we choose to sample the points on the light source impacts the look of the shadows. We studied two approaches:

- Uniform grid: divide the area of the light source with a uniform grid and sample a point at the center of each grid cell. This leads to a visible pattern in the shadow as shown in Figure 6 b).
- Jittered grid: divide the area of the light source with a uniform grid but instead of sampling at the center we choose the position randomly. This eliminates the pattern but also leads to a noisier result as shown in Figure 6 c). By increasing the number of samples the noise can be reduced and a smooth shadow is obtained (Figure 6 d))

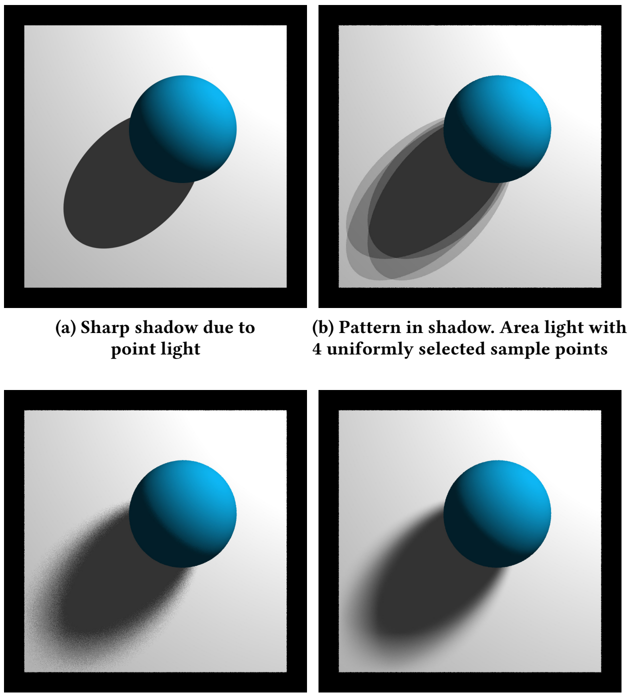

(a) Sharp shadow due to point light
(b) Pattern in shadow. Area light with 4 uniformly selected sample points
(c) Softer but noisy shadow. Area light with 4 randomly selected sample points
(d) Soft shadow without visible noise. Area light with 100 randomly selected sample points

Figure 5: Comparison of different light sampling approaches

5.3 Supersampling (anti-aliasing)

The basic attempt to raytrace a scene is to shoot a primary ray starting from the camera through each pixel center. But this approach leads to pixelated edges in the scene as seen in Figure 6 a). To overcome this problem we can sample the scene in a much higher resolution by shooting more than one ray per pixel and average those color results to determine the color of the pixel. Let N be the number of sub samples. Then the pixel color is calculated as follows [10].

$$pixelColor = \frac{1}{N} * \sum_{n=1}^{N} subPixelColor$$

Figure 6 b) uses 16 samples per pixel.

To subsample a pixel we use a jittered grid: we divide each pixel with a uniform grid and sample a random position in each grid cell.

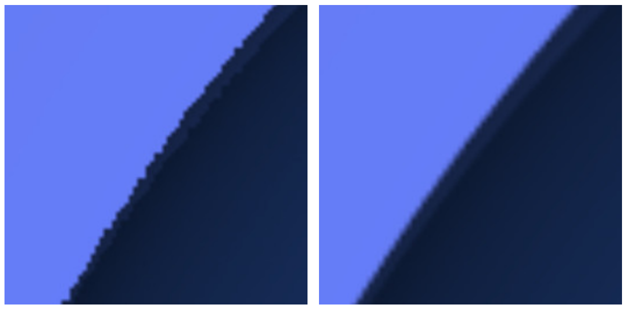

(a) No super sampling: pixelated edge
(b) 16xsuper sampling: smoother edge

Figure 6: Super sampling

6 PROBLEMS AND ACCIDENTAL ART

The biggest challenge in this project was the correct calculation of the reflective and refractive rays using Snell's law and the Fresnel equations.

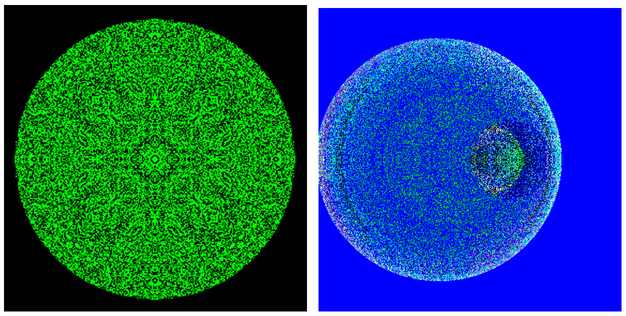
Also it was very challenging to assess the rendered scene in terms of correctness. Since the implemented technique does not take indirect illumination into account it was sometimes hard to judge if the rendered scene is correct given the constrains of the algorithm. This was especially true for the glass sphere.

Errors in the code sometimes resulted in very nice accidental art. Figure 7 shows the most beautiful accidents.

7 LIMITATIONS AND FUTURE WORK

The biggest limitation of this implementation is the simulation of global illumination. The used technique can not account for indirect lighting or caustic effects caused by lighting objects via reflection or refraction.

By tracing the ray backwards from the eye into the scene the algorithm assumes that only those rays that reach the eye contribute to the lighting of the scene. Lighting produced from reflection or refraction of light on surfaces are not taken into account. This is a large limitation in the attempt to render a realistic image.
Many different techniques exist to attempt to solve the rendering


(a) **Fractal pattern on sphere surface due to self occlusion**

(b) **Interesting looking death star formation what was supposed to be a smaller glass sphere in front of a bigger sphere**

Figure 7: Accidental art

equation [2] and produce a realistic image, e.g Photon Mapping, Path Tracing, Radiosity. In future work we want to implement and integrate one of those techniques into our project to overcome its limitations.

REFERENCES

[1] An improved illumination model for shaded display, `https://dl.acm.org/doi/10.1145/358876.358882`, Turner Whitted, Bell Laboratories, 1980

[2] The Rendering Equation, `http://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf`, James T. Kajiya, California Institute of Technology, 1986

[3] Rendering Soft Shadows, `http://raytracerchallenge.com/bonus/area-light.html`

[4] Rendering Soft Shadows, `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel`

[5] Barycentric coordinates, Lecture notes of course DD2258 Introduction to Visualization and Computer Graphics, *Grids and Interpolation*, p.61ff. Tino Weinkauf, KTH Stockholm, 2021.

[6] Phong Illumination Model, Lecture notes of course DD2258 Introduction to Visualization and Computer Graphics, *Visibility Shading*, p.59ff. Tino Weinkauf, KTH Stockholm, 2021.

[7] Perlin Noise, `https://mrl.cs.nyu.edu/ perlin/noise/`

[8] Introduction to Acceleration Structures, `https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1`

[9] Procedural Textures, `https://sebastiandang.github.io/docs/cse168/RayTracing.pdf`

[10] Supersampling, `https://en.wikipedia.org/wiki/Supersampling`

[11] UV Mapping, `https://en.wikipedia.org/wiki/UV_mapping`