CSE306 Assignment 1
Physically-Based Rendering

William Koch

May 9, 2021

1 Introduction

Physically-based rendering is a rendering approach in which a computer attempts to accurately simulate the basic physical properties of light, material (including reflections) and shadows. Other approaches (such as those frequently used in real-time rendering applications) use fast approximations to achieve effects that trick the viewer into believing the physical accuracy of the image.

In this assignment, I have implemented a path-tracer in CUDA using the C++ API with the following features:

- Diffuse, mirror and glass surfaces, including Fresnel
- Direct lighting and shadows for point light sources
- Indirect lighting for point light sources
- Antialiasing
- Ray-mesh intersection using the bounding-box optimization

On top of this, I paid attention to the usability of the code in more general settings. This includes, in particular, loading scenes from JSON, allowing you to render different or modified scenes without recompiling the code:

- Spheres (position, radius, material, color) can be specified in an array
- Objects (position, rotation, scale, material, color) can be specified in an array by providing a path to a Wavefront file for each object. Note that texture and normal maps are not supported yet.
- Point-lights (position, intensity) can be specified in an array
- Camera parameters allow you to easily change the position and rotation of the camera, as well as the resolution (non-square resolutions are supported), field of view and number of rays.

You can refer to `assets/scenes/default.json` for an example.

Animations can be easily achieved by changing `main.cu` to iterate over a list of scene files and writing the resulting images to a folder. If you only intend on moving the camera, the `Scene` class provides simple public transformation and rotation methods you can call from `main.cu`. An example of this can be found in the projects Github repository. When rendering multiple frames, change the seed in `Scene::render` on every frame (e.g. by setting the current frame number to be the seed). This ensures there is no static noise in the video.

2 Implementation Details

2.1 Pixel, Camera and World Space

In order to generalize my code to arbitrary camera rotations, translations, resolutions and field of view parameters, I added support for camera intrinsics and extrinsics. The camera intrinsic matrix K is commonly defined as

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} f_x &= \frac{\text{width}}{2 \cdot \tan(\pi * \text{fov}/360)} \\ f_y &= \frac{\text{height}}{2 \cdot \tan(\pi * \text{fov}/360)} \\ c_x &= \frac{\text{width}}{2} \\ c_y &= \frac{\text{height}}{2}. \end{aligned}$$

The camera extrinsic matrix is a 4×4 matrix, composed of the 3×3 rotation matrix R and translation vector t:

$$E = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

Converting a point from world to camera space is then as simple as computing $E \cdot p_{\text{world}}$. Converting a pixel to a 3D point in camera space (e.g. to obtain the ray direction for a specific pixel), we can use the inverse of the intrinsic matrix: $p_{\text{camera}} = K^{-1} p_{\text{pixel}}$.

I used camera intrinsic and extrinsic matrices a lot for a project during my exchange at TUM, so I re-used some of the code snippets for these projects (available in `include/projection_helpers.cuh`).

2.2 CUDA-Specific Modifications

2.2.1 `float` instead of `double`

Modern GPU's are optimized for 32-bit floating point operations. While working with 64-bit floating point numbers is certainly possible, it does affect performance noticeably, nearly doubling render times. I therefore decided to use `float` instead of `double`.

2.2.2 C++ Standard Library

Many classes from the C++ Standard Library cannot be used in device code (i.e. code that runs on the GPU), such as `std::vector`, `std::list` or `<random>` (however, there is a library called `thrust`). Instead, I used `std::vector` when loading the scene objects dynamically, and then copied a standard fixed-sized C++ array to the GPU.

Generating random numbers in particular requires storing states for every single thread individually.

2.2.3 Recursive get_color

Implementing the `get_color` function in a recursive manner was unfortunately not feasible in CUDA. This is theoretically possible with a workaround by creating a function template and creating different instances of the function with the current depth as a parameter. Since we call different functions at every recursion step, this is not a recursive function in the classic sense.

While this may be possible in theory, I noticed in practice that my code would start crashing at a certain depth, potentially due to GPU memory limitations. To avoid the limitation, I rewrote the provided code section to work in a non-recursive manner using a for-loop. I achieved this modifying `start` and `ray_dir` variables and recomputing the intersecting point at each iteration of the for loop. Updating the color value was not as straight-forward as this, but can easily be achieved, as shown below:

```
__device__ CU_Vector3f get_color(...) {
    CU_Vector3f L;
    CU_Vector3f albedo(1.f, 1.f, 1.f);

    for(int depth = 0; depth < MAX_RAY_DEPTH; depth++) {
        CU_Vector3f P = get_intersection(...);

        if(intersection) {
            ...
            if(material.type == DIFFUSE) {
                ...
                CU_Vector3f direct = lights[l].I / (4*M_PI*M_PI*d*d) * ...;

                L += albedo * direct;
                albedo *= material.color;

                start = P + 0.01*N;
                ray = random_cos(N);
            }
            ...
        }
        else { return L; }
    }
    return L;
}
```

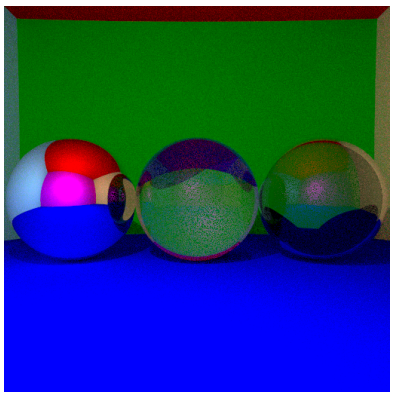
Correctness can be easily verified by expanding the lighting term of the recursive implementation.

2.2.4 Inherited Classes

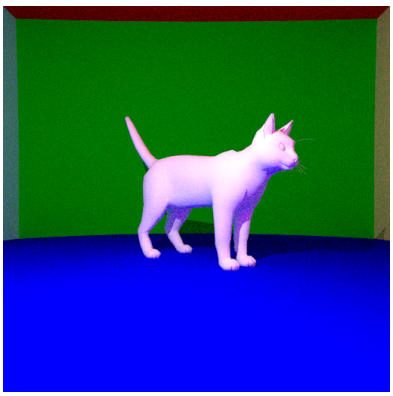
I refrained from using a `Geometry` parent class, since I need to create arrays of triangles and spheres separately. Otherwise, finding the exact size of the array when copying the objects from the host to the device (required for copying to the GPU) would be more complicated. As I am not that experienced with CUDA yet, I am not even sure if it is possible to copy an array of objects of different sizes to the device.

3 Performance

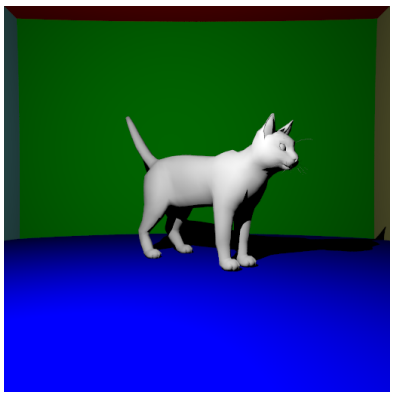
All performance benchmarks have been performed on an NVIDIA GeForce GTX 1050 mobile card. I only measured the performance of the GPU code (path-tracing), not however the host code (loading the scene and objects), which only causes a negligeable performance overhead.



Indirect Lighting + Antialiasing

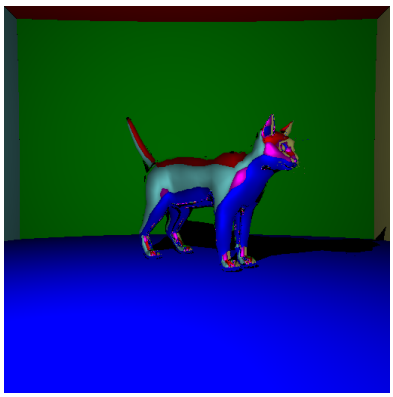


Indirect Lighting + Antialiasing

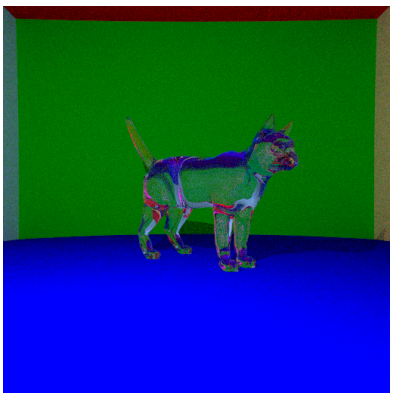


Antialiasing

Figure 1: Results of the benchmark tests.



Mirror, Simple



Glass, Indirect + Antialias.

Figure 2: Some more experimental renders.

Type	Default Scene	Cat Scene	Factor (cat / standard)
Simple	0.05 s	0.37 s	7.4
Simple + BBox	N/A	0.12 s	2.4
Antialiasing + BBox	0.12 s	4.14 s	18.0
Indirect Lighting + Antialiasing	0.23 s	59.38 s	258.2
Indirect Lighting + Antialiasing + BBox	N/A	53.77 s	233.8

We see an immediate performance drop when using indirect lighting and antialiasing with meshes. In fact, even when using just a single ray, it takes 2.54 seconds. This is caused by the fact that when we use indirect lighting, we cannot directly return from the `get_color` function when we hit a diffuse surface. This is an optimization we use when we don't use indirect lighting, so the performance gap diverges.

4 Conclusion

I unfortunately did not have time to implement BVH, textures, the pin-hole camera model or spherical light sources, and the performance improvements using the bounding box technique are questionable.

However, the performance on the default scene is very good, taking only 7.5 seconds to render with 1000 rays, indirect lighting and anti-aliasing. If I had spent more time optimizing my CUDA code for

meshes (in particular improving memory access for vertices and normals, as well as the bounding box technique), significant performance improvements on meshes would have still been possible.

5 Bibliography

(Bibtex was unfortunately causing issues)

- Lecture Notes CSE306 (Ecole Polytechnique)
- Lecture Notes "3D Scanning & Motion Capture" (TUM)
- CUDA Random Numbers, `http://ianfinlayson.net/class/cpsc425/notes/cuda-random`, last accessed: May 09, 2021
- Library tinyobjloader, `https://github.com/tinyobjloader/tinyobjloader`
- Library JSON, `https://github.com/nlohmann/json`