CS171 Assignment 3: Realistic Rendering with Ray Tracing

Introduction

In this assignment, you are required to implement the basic ray tracing framework as well as a Monte-Carlo path tracing for global illumination. To perform a ray-mesh intersection, you are also required to build a BVH tree as the acceleration structure. To implement this homework, make sure you have the basic knowledge of Monte-Carlo integration and rendering equation.

In the following, we will give you the specifics about what you need to accomplish in this assignment, as well as full documentation in order to assist your programming.

Programming Requirements

- **[must]** Compile the code and setup the language server environment. [5%]
- **[must]** Run `src/conventions.cpp`, implement `UniformSampleDisk`, `UniformSampleHemisphere` and `UniformSampleSphere` functions. [20%]
- **[must]** Implement ray-triangle intersection. [20%]
- **[must]** Implement the BVH tree with tree building and intersection. [25%]
- **[must]** Implement ray-generation in Integrator. [10%]
- **[must]** Implement path radiance estimate. [20%]
- **[optional]** Implement Microfacet BRDF. [10%]
- **[optional]** Implement Glass Material. [5%]
- **[optional]** Implement ThinGlass Material. [10%]
- **[optional]** Implement Rough Plastic Material. [15%]
- **[optional]** Implement Surface Area Heuristic. [5%]
- **[optional]** Implement Parallel BVH build (see "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees") also construct a large scene to verify the parallelism. [20%]
- **[optional]** Implement the Bidirectional Path Tracing (can be O(n^4) in path length without handling glass material). [25%]
- **[optional]** Implement PhotonMapping. [25%]
- **[optional]** Implement the Metropolis Light Transport (PSSMLT or MMLT). [25%]

Notes

- We provide a comprehensive documentation, you can optionally (and recommended to) follow it. The sections where it would help if you practiced are marked in purple in the document; the remainders are for your reading and comprehension. The purple sections partially corresponds to the programming requirements above.
- It is strongly recommended to read PBRT's corresponding section as a reference.
- Please note that you are **NOT** allowed to use OpenGL in this assignment and **NOT** allowed to use third-party libraries except for the libraries we give in the framework.
- As the computation in this assignment is quite heavy, we encourage you to use OpenMP (inside Phase 5.1) for acceleration. You may apply for an account on the SIST HPC cluster if necessary.
- The resulting image in your report needs to be clear enough, i.e. the image resolution should be at least 512x512, and the samples per pixel (spp) should be at least 128 for handling global illumination. You are free to change the scene settings inside, for example, `data/cbox.json`.

- To verify your implemented algorithms, you can choose or build your own scene settings in the `data/` folder.

Submission

You are required to submit the following things through the GitHub repository:

- Project code in the Coding folder.
- A PDF-formatted report which describes what you have done in the Report folder.

Submission deadline: **22:00, Nov. 30, 2024**

Grading Rules

- You can choose to do the **[optional]** item(s), and if you choose to do it/them; you will get an additional score(s) based on the additional work you have done. But the maximum additional score will not exceed 30% of the entire score of this assignment.
- **NO CHEATING!** If found, your score for the entire assignment is zero. You are required to work **INDEPENDENTLY**. We fully understand that implementations could be similar somewhere, but they cannot be identical. To avoid being evaluated inappropriately, please show your understanding of the code to TAs.
- Late submission of your assignment will be subject to a score deduction.

Skeleton Project/ Report Template

- The skeleton program and report template will be provided once you accept the assignment link of GitHub Classroom which we published on the Piazza. If you accept the assignment through the link properly, a repository which contains the skeleton project and report template will be created under your GitHub account.
- Please **follow the template** to prepare your Report.

Implementation Guide

The framework is large and complex, and you are not expected (and not recommended) to understand the whole framework before starting programming. Again, we recommend that you check out our documentation for a more detailed explanation of the framework and tasks. *Please resort to TAs for any difficulties you meet in understanding the code framework*.

Git Classroom

Accept the assignment in this link through Git classroom to start your assignment.

1. Compile the Code and Setup the Language Server Environment

In this section, you should compile our code and setup the language server with either existing IDEs or your own setup, e.g., `clangd` with generated `compile_commands.json`. You should demonstrate your setup in the Report and the features that it can possibly achieve. Since the external lib is managed by CPM, in order to help you mange your lib management, set the environment variable CPM_SOURCE_CACHE first; otherwise the compilation will be slow.

Some of the features are required in your environment to complete this assignment, you should never write code in a larger codebase with only a text editor. The features required are, but not

limited to "jump to definition", "find all references" and "switch between header and source". Report your setup inside the Report. Its completeness will be used to score.

An example using `vscode`(you can use the IDE you prefer) would be to install the extension `clangd` and generate `compile_commands.json` by yourself. `clangd` will find the `compile_commands.json` and use it to provide the functionalities, specially in windows you need to use Ninja as the generator and builder to generate the compile_commands, try to install Ninja.

2. Warmup with Math Conventions

Go and find the file `src/conventions.cpp`. It contains some examples of the math conventions we use in this assignment. Execute the corresponding executable `build/src/conventions`(possibly elsewhere) to see the results so that you can understand the basic use of the math library.

To get your hands wet, you are required to implement the `UniformSampleDisk`, `UniformSampleHemisphere` and `UniformSampleSphere` functions inside `include/rdr/math_utils.h`. You might be able to complete this task within 10 lines of code.

3. Implement Ray-triangle Intersection

As an integral part of a full-fledged renderer, the task in this section is to implement a simple ray-triangle intersection algorithm inside the `TriangleIntersect` function.

You might find the derivation in our documentation and this page helpful.

4. Implement the BVH Tree with Tree Construction and Ray-Mesh Intersection

In this section, your task is to implement the BVH tree construction and ray-mesh intersection algorithms. You are recommended to checkout `std::nth_element` for the tree construction, and if you find it difficult, you can also use `std::sort` instead. As for the tree traversal, we employ the recursive method inside `BVHTree<>::intersect()`.

`BVHTree<>::intersect()` accepts a callback function with a signature documented in the comments. You can also leverage the "find all references" functionality of your IDE to find the usage of the callback function. You can refer to our documentation and this page for reference.

5. Implement the Ray-generation in Integrator

This is a warmup section for the path-tracing algorithm. You are required to fill the implementation inside `PathIntegrator::render`, traverse the pixels and spawn the rays. You might find our documentation helpful for this section.

6. Implement the Path Radiance Estimate

Here comes the boss. Your task is to transform the math derivation of path tracing into approximately 70 lines of code estimating the radiance of a given path. The paths themselves are pre-built for you, and you are free to assume that they are all valid.

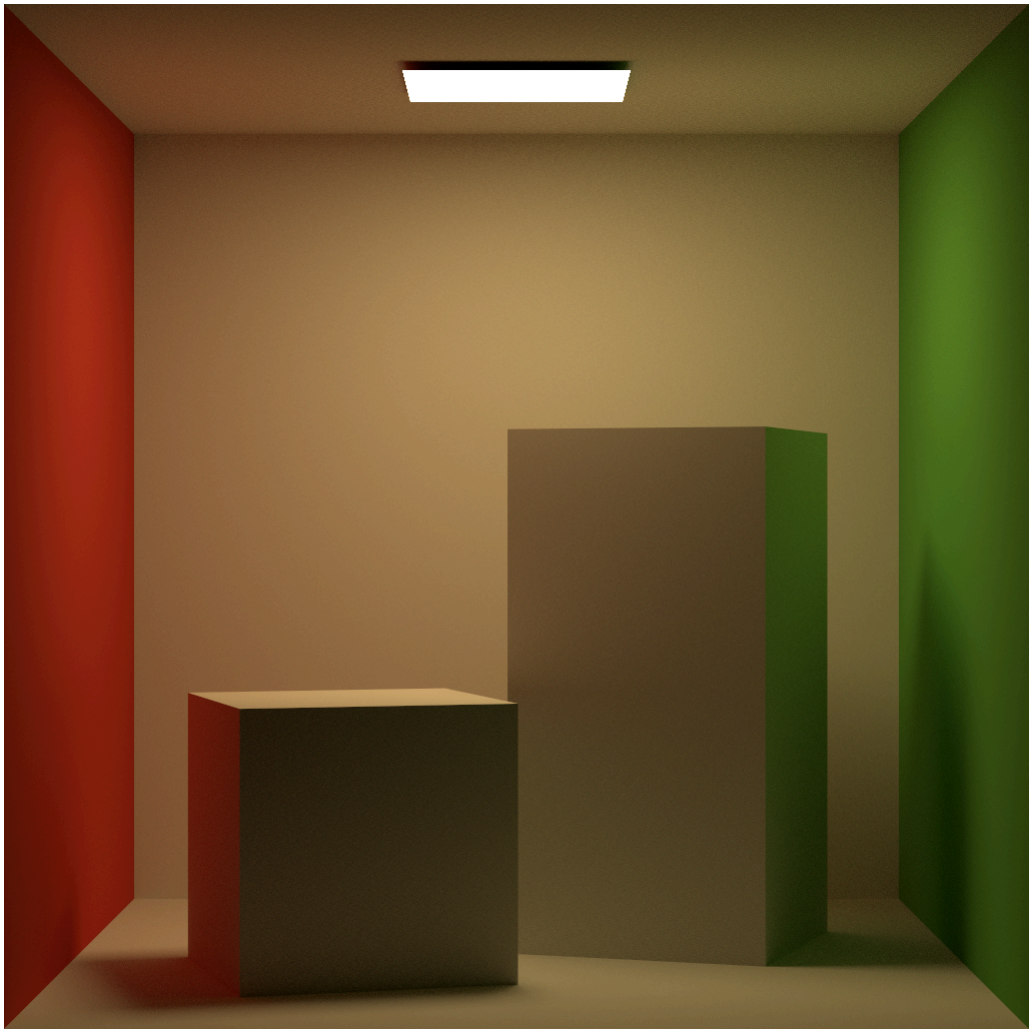
You are able to find the derivation in both of our documentation and PBRT page (you'll find the sections before and after useful).

Your implementation goes inside the `Path::estimate` function in `src/path.cpp`. Possible helper functions are already provided in the comment. Although a for-loop is sufficient to complete this task, it requires a deeper understanding of path tracing, so please read the references we provide, actively ask questions and think deeply before you start writing code.

Expected Results

The rendering result can be obtained by invoking the `renderer` executable with the scene file inside `data/` as terminal argument. For example, you can run the cornell box scene with the `build/src/renderer data/cbox.json`. A simple demonstration of the rendering result of `cbox.json` is sufficient and shown below. You are free to replace the mesh with other scenes in the `data/` folder (please explore it).

You can get a png output with, for example `renderer data/cbox.json -o output.png`. But the output might not be precise enough for you to validate correctness. So it is generally recommended to install tev or VERIV to view the exr result.



You might also find the exr reference images helpful.