CS171 Assignment 3: Basic Ray Tracing

Introduction

In this assignment, you will implement several core components of a ray-tracing renderer, including ray-triangle intersection, BVH tree construction and traversal. And rendering Cornell box with simple direct illumination.

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 source code and configure the language server environment. [5%]
- **[must]** Implement ray-triangle intersection functionality. [10%]
- **[must]** Implement ray-AABB intersection functionality. [10%]
- **[must]** Implement the BVH (Bounding Volume Hierarchy) construction. [25%]
- **[must]** Implement the `IntersectionTestIntegrator` and `PerfectRefraction` material for basic ray tracing validation, handing refractive and solid surface interactions [25%]
- **[must]** Implement a direct lighting function with diffuse BRDF and shadow testing. [20%]
- **[must]** Implement anti-aliasing via multi-ray sampling per pixel within a sub-pixel aperture. [5%]
- **[optional]** Implement support for multiple light sources. [5%]
- **[optional]** Implement rectangular area lights with soft shadow generation. [15%]
- **[optional]** Implement environment lighting via environment maps. [15%]
- **[optional]** Implement texture mapping support. [15%]
- **[optional]** Implement GPU-parallel BVH construction following the algorithm in Karras (2012). [30%]

Notes

- 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 256x256, and the samples per pixel (spp) should be at least 4 for handling antialiasing. 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. 21, 2025**

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*.

Also note that you can search `TODO(HW3)` in the codebase to quickly locate the places you need to implement.

Git Classroom

Accept the assignment in GitHub Classroom using this link or download the zip file to start your assignment.

Set Up Development Environment

After you accept the assignment in GitHub Classroom, clone the repository and set up your development environment. We recommend using VS Code with `clangd`. Please refer to this guide to get started.

Ray-Triangle Intersection

Complete the implementation of the ray-triangle intersection function `TriangleIntersect` in `src/accel.cpp`. You can use either the geometric approach, such as this or this, or the linear-equation-solving approach discussed in recitation.

Ray-AABB Intersection

Complete the AABB (Axis-Aligned Bounding Box) intersection in the function `AABB::intersect` in the file `src/accel.cpp`. You can refer to this article for details about the algorithm.

After implementing these two functions, you can test your implementation by running:

```
cmake -B build
cmake --build build
./build/tests/intersection_tests
```

Note that the location of the executable may vary depending on your build system.

BVH Construction

Implement the function `BVHTree<_>::build` in `bvh_tree.h`. Refer to the lecture slides, recitation materials, or the warmup assignment on BVH construction for guidance.

We recommend the following resources for a comprehensive understanding of BVH:

- Here is a detailed explanation of BVH with dynamic demonstrations.
- Here provides a practical guide covering the fundamentals of BVH.
- Here provides an article for a more in-depth and advanced topic of BVH structures.

After implementing this functions, you can test your implementation by running:

```
cmake -B build
cmake --build build
./build/tests/bvh_tests
```

Note that the location of the executable may vary depending on your build system.

Implement a Direct Illumination Integrator

Implement the `IntersectionTestIntegrator` class in `src/integrator.cpp`. This integrator should perform the following tasks:

- Cast multiple rays per pixel with small offsets for anti-aliasing.
- For each ray, find the closest intersection with the scene geometry using the BVH.
- Cast a shadow ray from the intersection point toward the light source to determine visibility.
- For each visible intersection, compute direct illumination from the light source.

You may use any simple shading model, such as Phong shading, to compute direct illumination.

Integrate with Refractive Materials

Modify the function `Li` in `src/integrator.cpp` and `PerfectRefraction::sample` in `src/bsdf.cpp` so that rays are perfectly refracted when they hit refractive materials.

- Cast multiple rays per pixel with small offsets for anti-aliasing.
- For each ray, **trace the ray through transparent objects until it first intersects a non-transparent (solid) object**.

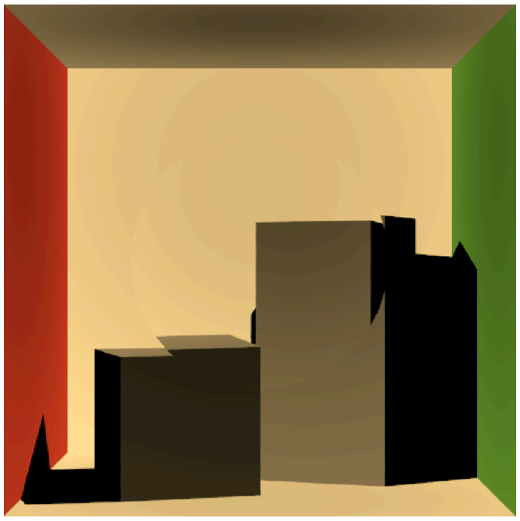
- Cast a shadow ray from that intersection point toward the light source to determine visibility.
- For each visible intersection, compute direct illumination from the light source.

Final Results

After completing these tasks, you should be able to render images similar to the following:

```
cmake --build build
./build/src/renderer data/cbox_no_light_refract.json -o cbox_no_light_refract.exr
# or the following if you want to debug without refraction
./build/src/renderer data/cbox_no_light.json -o cbox_no_light.exr
```

You should see an output image saved in the `data/cbox_no_light_refract.exr`.



The image shows the glass pane refracting camera rays while the boxes cast shadows on the floor.