Assignment 3: Basic Ray Tracing

NAME: LIU ZHEN
STUDENT NUMBER: 2023533027
EMAIL: LIUZHEN2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment, we 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.

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

2 IMPLEMENTATION DETAILS

2.1 Compile the source code and configure the language-server environment

We use Visual Studio Code with clangd extension for this project. Since clangd needs *compile_commands.json* and mscv can't generate that, we use ninja as an alternative. We use the following command lines to compile and build:

```
cmake -B build -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
-DCMAKE_POLICY_VERSION_MINIMUM="3.5"
cmake –build build
```

2.2 Implement ray–triangle intersection functionality

My implementation decomposes the ray–triangle intersection test into three stages. First, we copmpute the ray's intersection with the triangle's supporting plane. The normal $\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$ and plane constant $d = \mathbf{n} \cdot \mathbf{v}_0$ are obtained from the triangle edges. If $\mathbf{n} \cdot \mathbf{dir} = 0$ the ray is parallel to the plane and the routine exits immediately; otherwise the parametric distance

$$t = \frac{d - \mathbf{n} \cdot \mathbf{origin}}{\mathbf{n} \cdot \mathbf{dir}}$$

is evaluated and clamped against the ray's valid interval $[t_{\min}, t_{\max}]$. Second, the three-dimensional containment problem is reduced to two dimensions by means of barycentric coordinates. The hit point $\mathbf{q} = \mathbf{origin} + t\,\mathbf{dir}$ is projected onto the plane and its barycentric weights are derived from signed-area ratios expressed as dot products with the constant normal $\mathbf{n}$:

$$u' = \frac{\big((\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{q} - \mathbf{v}_1)\big) \cdot \mathbf{n}}{(\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{n}}, \qquad v' = \frac{\big((\mathbf{v}_0 - \mathbf{v}_2) \times (\mathbf{q} - \mathbf{v}_2)\big) \cdot \mathbf{n}}{(\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{n}}.$$

Early exits guard the intervals $u', v' \in [0, 1]$ and the sum constraint $u' + v' \leq 1$. Finally, the weights are rearranged to match the assignment specification:

$$u = v', \qquad v = 1 - u' - v',$$

ensuring the barycentric interpolation $(1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$ coincides exactly with $\mathbf{q}$ while preserving the non-negativity and unit-sum conditions.

2.3 Implement ray–AABB intersection functionality

First we compute the inverse direction using the safe_inverse_direction method. Then we calculate two sets of t values for each dimension (x, y, z). These t values represent the parametric distances from the ray's origin to the AABB's boundaries along the ray's direction. The calculations are done as follows:
t1 is computed by subtracting the ray's origin from the AABB's lower boundary and multiplying by the inverse of the ray's direction. This gives the t values for when the ray would intersect the lower boundary of the AABB.
t2 is computed similarly but using the AABB's upper boundary. This gives the t values for when the ray would intersect the upper boundary of the AABB.
The tMin vector contains the maximum of the t1 and t2 values for each dimension, representing the latest possible entry point into the AABB. Conversely, tMax contains the minimum of the t1 and t2 values, representing the earliest possible exit point from the AABB.The ReduceMax and ReduceMin functions are used to find the maximum of the tMin values and the minimum of the tMax values, respectively. These values represent the entry and exit times of the ray with respect to the AABB.Finally we check if the entry time is greater than the exit time or if the exit time is less than zero. If either condition is true, it means the ray does not intersect the AABB, and the function returns false. If the ray does intersect, the entry and exit times are stored in the provided pointers t_in and t_out, and the function returns true.

2.4 Implement the BVH (Bounding Volume Hierarchy) construction

First, we compute the AABB. Then we check whether the current node's depth has reached the preset maximum depth. Additionally, we check whether the number of child nodes contained in the current node is less than or equal to one. If a node contains only one or no child nodes, it can no longer be subdivided and should therefore be created as a leaf node. If either of the above conditions is met, the code will create a leaf node. The leaf node includes the current node's index range (span_left to span_right), and is marked as a leaf

node . The leaf node's AABB is set to the pre-built AABB, then it is added to the list of internal nodes, and its index in the list is returned. Otherwise, we create an internal node and choose the dimension with the largest extent. To achieve median segmentation, we defined a lambda expression cmp as the comparison function, which compares two nodes based on the value of the center of their AABB in the specified dimension. This comparison function is used to determine the relative position of nodes during the sorting process.Then we use std::nth_element to split the set of nodes into two parts, such that all nodes on the left side have a center value in the specified dimension that is less than those on the right side.

2.5 Implement the IntersectionTestIntegrator and PerfectRefraction material for basic ray-tracing validation, handling refractive and solid-surface interactions

First, we define a color vector and set to black (0, 0, 0) to store the final illumination color. The code uses a loop to recursively cast rays until a maximum depth is reached or a diffuse surface is encountered. In each iteration, the intersection of the ray with the scene is detected using the method scene->intersect(ray, interaction). Through runtime type identification (RTTI), the code determines the type of surface at the intersection point. It checks for two types: ideal diffuse surfaces and perfect refraction surfaces. If the surface at the intersection point is a perfect refraction surface, the code needs to call the method interaction.bsdf->sample to obtain the new direction of the ray. This method calculates the scattering direction of the ray at the intersection point based on the BSDF of the refractive material. Then, the ray is updated using the method interaction.spawnRay(interaction.wi), where interaction.wi is the new direction obtained from interaction.bsdf->sample. If the surface at the intersection point is an ideal diffuse surface, the code sets diffuse_found to true and exits the loop. If a diffuse surface is found, the directLighting method is called to compute the contribution of lighting to the intersection point's color, and the result is returned. Finally, the function returns the computed illumination color.

2.6 Implement a direct-lighting function with diffuse BRDF and shadow testing

First, we define a color vector and initialize to black (0, 0, 0) to store the final lighting color.Then we calculate the vector between the light source positionand the intersection point to obtain the light direction and normalize it. At the same time, the distance between these two points is calculated. A ray is constructed using the intersection point position and the normalized light direction for subsequent shadow testing. The scene's intersect method is called to detect whether this ray intersects with any geometry in the scene. If shadow_ray intersects with the scene and the distance from the intersection point to is less than the distance from the light to the intersection point minus a small tolerance value eps, it is considered occluded, and black is returned. If the intersection point is not occluded, the contribution of the lighting to the intersection point's color is then calculated. First, a runtime type identification (RTTI) check is performed to determine whether the surface's BRDF is an ideal diffuse model. If it is an ideal diffuse, the lighting contribution is calculated. A simplified Phong shading model is used here to approximate the lighting contribution. First, the cosine of the angle between the light direction and the surface normal is calculated. The final color is computed by multiplying the incident radiance ($Li = point\_light\_flux/(4\pi r^2)$) with the BRDF value and the cosine of the incident angle Finally, the function returns the computed lighting color.

2.7 Implement anti-aliasing via multi-ray sampling per pixel within a sub-pixel aperture

For each pixel, a sampler object is instantiated, and its pixel index is set accordingly. The function then iterates over the number of samples per pixel, generating multiple rays to perform Monte Carlo integration and compute the radiance for each sample. For each sample, we first retrieve a pixel sample position using the Sampler's getPixelSample method. It then generates a differential ray using the camera's generateDifferentialRay method, which takes the sample position as input. The differential ray is used to simulate the variation in ray direction due to pixel sampling, which helps in achieving anti-aliasing effects. After generating the ray, the function asserts that the sample position is within the bounds of the pixel. It then calls the Li function to compute the radiance at the sample position by tracing the ray through the scene. The computed radiance is accumulated into the film. By iterating over all pixels and samples, the function accumulates the radiance values to produce a final rendered image. The use of Monte Carlo integration allows for the estimation of complex lighting effects and the simulation of realistic scenes.

2.8 Implement support for multiple light sources

When implementing a system with two light sources, we initially redefine an additional light source within the IntersectionTestIntegrator class in the integrator.h file, adjusting the position of the existing light source as necessary. Following this, in the directLighting function within integrator.cpp, we conduct shadow tests for each of the two light sources. If neither light source can reach the point in question, we return black to signify complete occlusion. Otherwise, we compute the color contribution for each light source that successfully illuminates the point and aggregate these contributions to ascertain the final color. This methodology ensures that our lighting model accurately accounts for the presence of multiple light sources and their interactions with the scene's geometry.

2.9 Implement rectangular area lights with soft-shadow generation

We defined the relevant parameters of a rectangular area light source, such as center, length, width, and normal vector, in IntersectionTestIntegrator, and implemented a direct illumination integrator based on multiple importance sampling in the corresponding direct illumination section, which is used to calculate the direct illumination contribution of a certain point on the object surface in path tracking. We first traverse all the light sources in the scene, sample each light source num * num times, and simulate the test light emitted from surface points towards the light source. Each time a sample is taken, a sampling point on the light source and its corresponding probability density are obtained through light ->sample(), and the distance and direction from the surface point to the light source are calculated. Subsequently, a shadow test ray is constructed to determine whether it intersects with other objects in the scene. If there is a closer intersection point, it indicates that the light source is obstructed and the current sampling is invalid, skipping this contribution. For unobstructed lighting, further determine whether the surface material is ideal diffuse reflection. If so, calculate the incident radiance in that direction, and combine the evaluation value of the bidirectional scattering distribution function, the cosine value of the normal angle between the surface and the light source, and the distance squared decay term to accumulate the lighting contribution. All sampling results are finally averaged to obtain the direct illumination color value of the point. Besides, we define a rectangle class in shape. h and add a registration function. At the same time, uniform sampling of rectangles was implemented in shape.cpp to obtain the range of AABB and intersection of rectangles. We have removed the point light source in the configuration file cbox _no-light _ fract _ realight. exr, replaced it with a rectangle, and declared the light attribute as the light

source. In order to display the light source, we modified the Li function to detect whether the light directly hits the emitting surface, and if it does, return the radiance directly.

2.10 Implement GPU-parallel BVH construction following the algorithm in Karras (2012)

We first define the BHVnode structure represents each node in the BVH. It contains attributes such as the Axis-Aligned Bounding Box coordinates, child indices, and flags indicating whether the node and its children is a leaf node or not. Several CUDA kernel functions are defined to perform parallel computations on the GPU. We compute Morton codes for the bounding box centers and stores them along with their original indices. We sort the Morton codes and indices using Thrust, a CUDA template library that provides standard algorithms and data structures, facilitating efficient sorting on the GPU. The algorithm function recursively constructs the BVH by finding the best split for each node based on Morton codes. The calculateBoudingBox function computes the AABB for each node in the BVH by combining the AABBs of its children, ensuring that each node's bounding box accurately encapsulates its descendants. The initNodesKernel function initialize the BVH nodes with default values, setting up the structure before the actual construction process begins. The build function orchestrates the BVH construction process. It allocates device memory, calls the initialization kernel, computes centers, sorts the Morton codes, constructs the BVH, and copies the results back to the host. This function serves as the main interface between the host CPU and the device GPU. For memory management, we manage device memory using cudaMalloc for allocation and cudaFree for deallocation. Data transfer between the host and device is handled using cudaMemcpy, ensuring that all necessary data is correctly moved to and from the GPU. Throughout the implementation, CUDA errors are checked after each CUDA API call. If an error occurs, an error message is printed, and the program can take appropriate action or terminate gracefully. Gnerally, we input the min and max information of AABBs and normalize it in every dimension before calculating and sorting the Morton codes. Then we call algorithm to get internalNodes arrays and leafNodes arrays which store the topological relationship of the tree. Finally, we call calculateBoudingBox to fill the AABB information in the nodes.

3 RESULTS

Figure 1 is a image showing the glass pane refracting camera rays while the boxes cast shadows on the floor.
Figure 2 is a image with multiple lights.
Figure 3 is a image with a rectangular area light.
Figure 4 is the outcome of a BVH tree with four nodes.

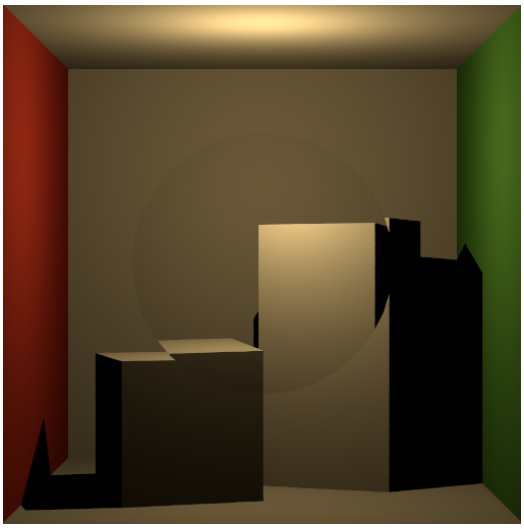

Fig. 1

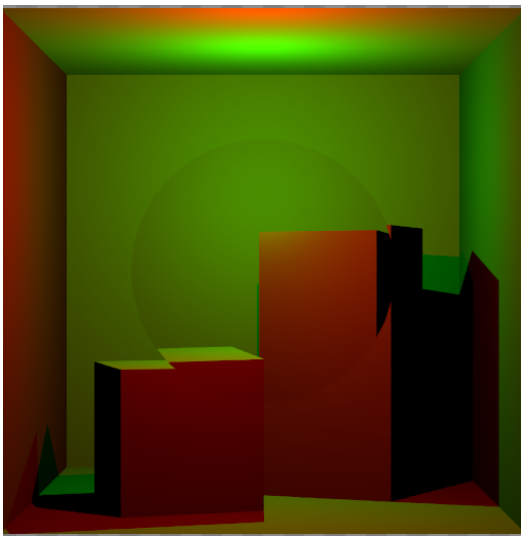

Fig. 2

1:4 • Name: Liu Zhen
student number: 2023533027
email: liuzhen2023@shanghaitech.edu.cn

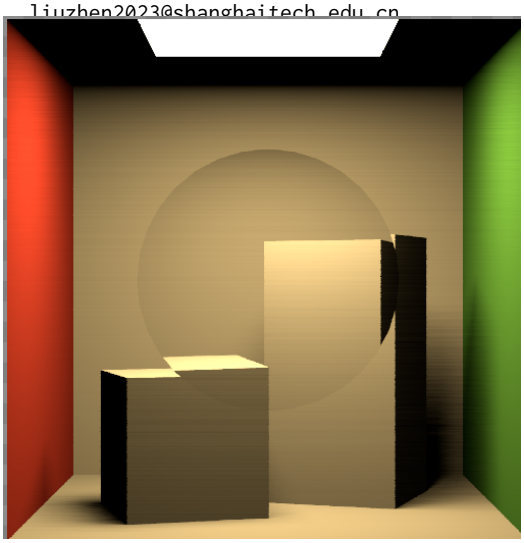

Fig. 3

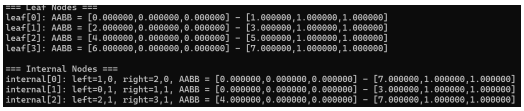

Fig. 4