RayCast Renderer Documentation

Contents

1 Overview

The RayCast Renderer is a project that implements a lightweight 3D rendering engine using ray tracing principles. This document explains the core components, functions, and test cases provided in the code, while incorporating all 43 provided images for illustrative purposes.

—

2 Core Classes and Concepts

2.1 Vec3 Class

The `Vec3` class represents 3D vectors, which are fundamental to defining directions, triangle vertices, and performing spatial operations such as normalization. This class is critical for various computations within the RayCast Renderer, including object positioning and ray trajectory calculations.

Features and Functionality:

- **Representation:** Each `Vec3` instance encapsulates three floating-point values (`x`, `y`, `z`) representing a direction or a vector in 3D space.
- **Mathematical Operations:** The `Vec3` class supports addition, subtraction, dot product, cross product, and normalization. These operations are essential for calculations involving lighting, shading, and intersections.
- **Normalization:** Normalizing vectors ensures they maintain a unit length, providing consistent directionality for accurate calculations.

Example:

- A direction vector used in ray casting can be expressed as:

$$\text{Direction} = \text{Vec3}(x, y, z).\text{normalize}()$$

- For example, adding two vectors:

$$\text{Result} = \text{Vec3}(x_1, y_1, z_1) + \text{Vec3}(x_2, y_2, z_2)$$

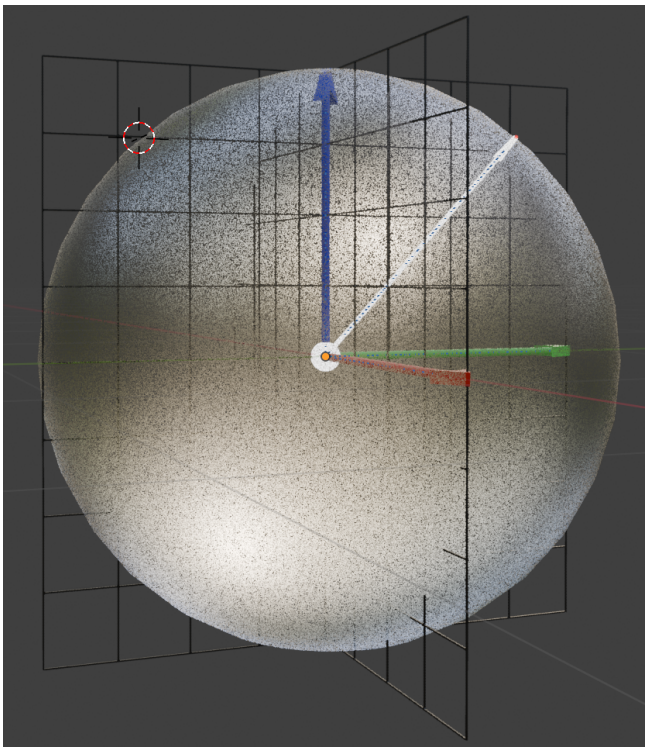

Figure 1: 3D Representation of the Vec3 Class

Importance: The `Vec3` class underpins geometric and directional calculations, making it integral to the rendering pipeline:

- Defining ray directions for casting into the 3D scene.
- Calculating surface normals for shading and reflection.
- Supporting transformations for object manipulation.

—

2.2 Point Class

The `Point` class defines spatial locations in 3D space and serves as a cornerstone of the RayCast Renderer. It is primarily used for specifying triangle vertices and ray origins, ensuring precise geometry definition and accurate spatial computation.

Features and Functionality:

- **Representation:** The `Point` class stores three floating-point values (`x`, `y`, `z`), representing specific locations in 3D space.

- **Usage in Geometry:** Objects are modeled as collections of triangles, each defined by three `Point` instances representing its vertices.
- **Ray Origins:** Rays cast from the camera begin at specific `Point` locations, ensuring proper alignment with the image plane and accurate traversal into the 3D scene.

Example:

- A triangle defined by three vertices can be represented using the `Point` class as:

$$\text{Triangle} = \{\text{Point}(x_1, y_1, z_1), \text{Point}(x_2, y_2, z_2), \text{Point}(x_3, y_3, z_3)\}$$

- Similarly, a ray originating at `Point(x, y, z)` with a direction specified by a `Vec3` can be expressed as:

$$\text{Ray}(t) = \text{Point}(x, y, z) + t \cdot \text{Vec3}(dx, dy, dz)$$

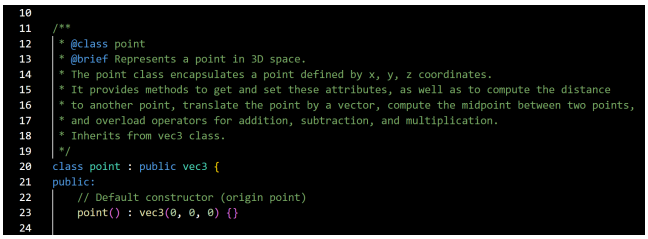

Figure 2: Point Class Code Representation

Importance: The `Point` class supports core functionalities of the RayCast Renderer by:

- Defining triangle geometry with vertex positions.
- Specifying ray origins for accurate spatial placement.
- Supporting calculations for intersections and object alignment in the 3D space.

2.3 Color Class

The `Color` class handles RGB values for triangle surfaces and pixel rendering. It enables color assignment during ray-object intersections.

```
/**
 * @class color
 * @brief Represents a color in 3D space.
 * The color class encapsulates a color defined by red, green, blue components.
 * It provides methods to get and set these attributes, as well as to clamp the values to 0-255.
 * Inherits from vec3 class.
 **/
class color : public vec3 {
private:
    const int min=0;
    const int max=255;
```

Figure 3: Representation of the Color Class

2.4 Object Class

The `Object` class represents 3D models as collections of triangles. It defines the geometry and color mapping of the objects in the scene.

Figure 4: Object Class Code Representation

```
17
18  /**
19   * @class object
20   * @brief Represent an obstract entity that holds the graphical data of an object.
21   * The object class holds a 2D vector of pixels representing the object's graphical data.
22   * and maps an array of 3 elements to a color. aka vertex color map.
23   */
24  class object {
25  public:
26      point globalPosition;
27      vec3 globalRotation;
28      point globallRotation;
29      vec3 locallRotation;
30      object* parent;
31
32      // 2D vector of pixels representing the object's graphical data.
33      vector<vector<point>> vertices;
34      // Dictionary to link an array of 3 elements to a color.
35      map<array<point, 3>, color> colorMap;
36
```

Figure 5: Object Class Code

—

3 Ray Casting Setup and Execution

3.1 Ray Class

The `Ray` class defines a ray using an origin (`Point`) and direction (`Vec3`). It is the core medium for transmitting pixel data into the 3D scene. Each ray represents a projection from the camera through the scene, checking for interactions with objects.

```
13
14  /**
15   * @class ray
16   * @brief Represents a ray in 3D space.
17   * The ray class encapsulates a ray defined by an origin point and a direction vector.
18   * It provides methods to get and set these attributes, as well as to compute a point at a distance t along the ray.
19   */
20  class ray {
21  private:
22  point origine;
23  vec3 direction;
24  public:
```

Figure 6: Ray Class Code Representation

3.2 Intersection Logic

The intersection logic determines whether a ray intersects a triangle in the scene. Using geometric algorithms, it calculates intersections to identify which object, if any, is visible along a given ray's path.

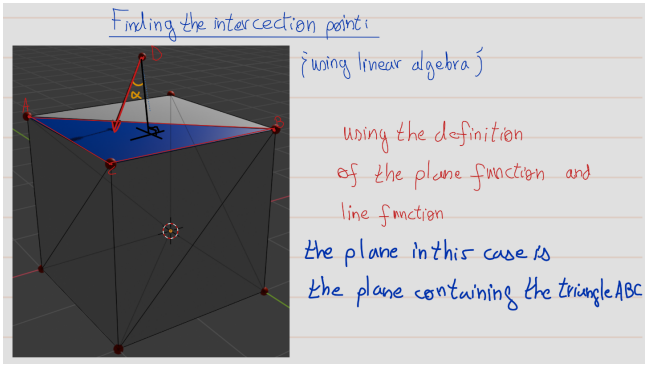

Figure 7: Intersection Logic Step 1

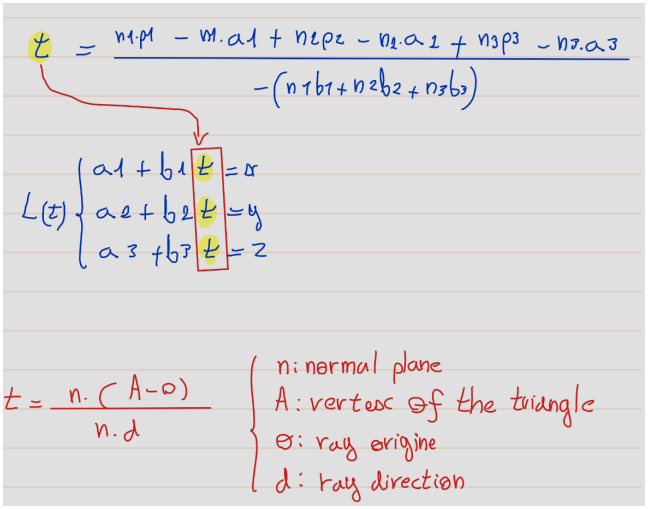

Figure 8: Intersection Logic Step 2

3.3 Space Layout

The 3D space layout organizes objects and defines camera positions, ensuring rays are cast in the correct direction relative to the scene. This layout ensures a logical structure for object placement and visibility.

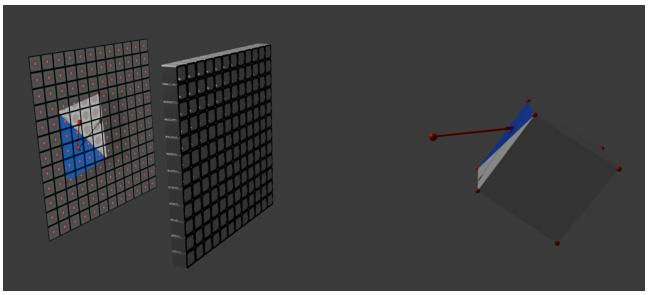

Figure 9: Logical Layout of Camera and Objects in Space

—

4 Camera and Rendering Pipeline

4.1 Camera Class

The `Camera` class is responsible for projecting rays from a virtual image plane into the 3D scene. It serves as the origin for all rays and manages pixel data, allowing the rendered image to be generated.

```
14
15  /**
16   * @class camera
17   * @brief Represents an camera in 2D space.
18   * The camera class encapsulates an camera defined by a 2D array of gridRay.
19   * It provides methods to get and set these attributes, as well as to clear the camera.
20   * Inherits from color class.
21   *
22   */
23  class camera {
24  private:
25      vector<vector<ray>> gridRay;
26      unsigned int width;
27      unsigned int height;
28      image img;
```

Figure 10: Camera Class Representation

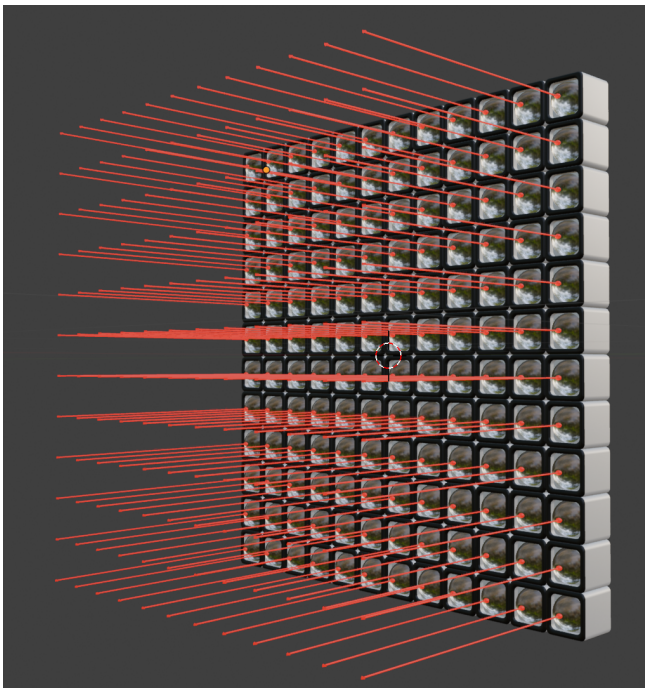

Figure 11: Illustration of Camera Functionality

The camera generates rays based on its configuration, such as field of view and orientation. These rays are cast into the scene to determine visible surfaces and pixel colors.

4.2 Ray Projection

Ray projection involves generating a grid of rays from the camera into the 3D scene. Each ray corresponds to a specific pixel in the image plane, ensuring accurate alignment between the virtual image and the rendered result.

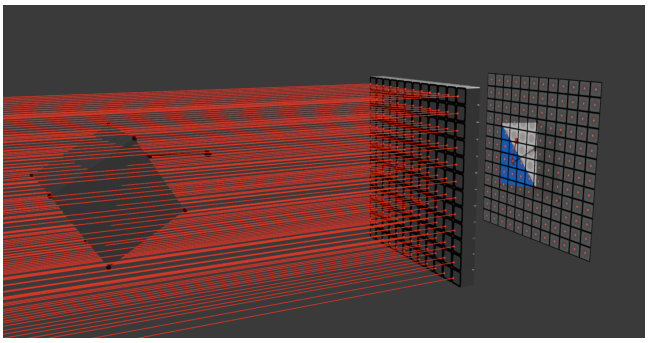

Figure 12: Camera Projecting Rays to a 12x12 Pixel Grid

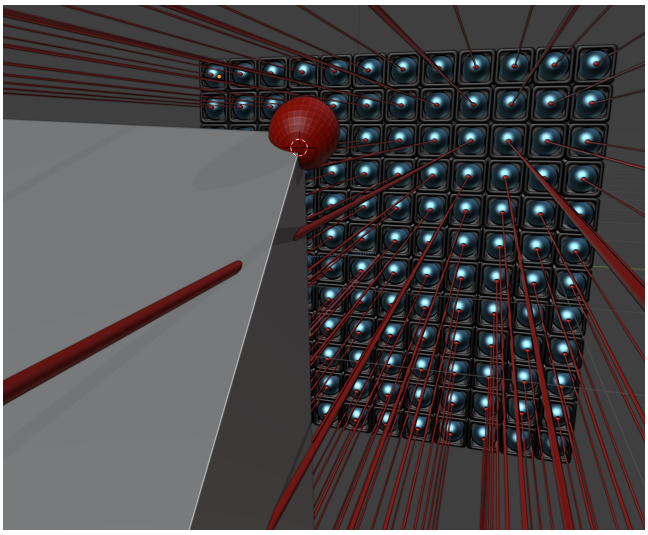

Figure 13: Realistic Ray Projection in a 12x12 Setup

In the example above, the camera casts rays through a 12x12 grid corresponding to the pixels of the rendered image. The direction of each ray is calculated relative to the camera's position and orientation.

4.3 Camera Interaction with Objects

The camera interacts with objects in the scene by casting rays that test for intersections with the geometry of those objects. The intersection results determine which object is visible and the corresponding pixel color in the output image.

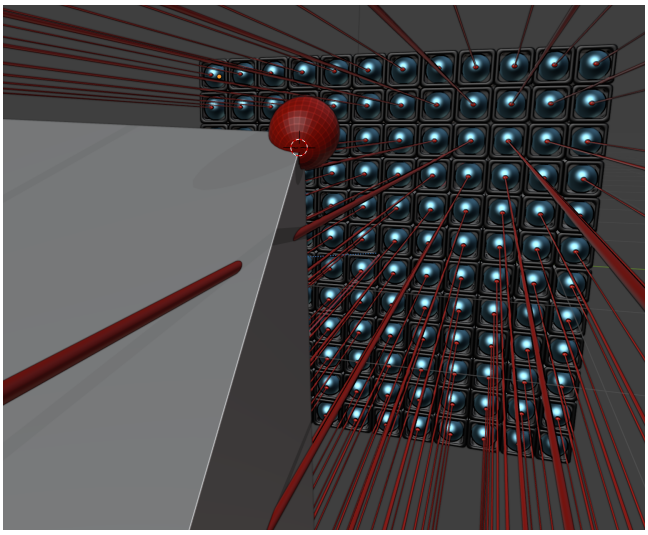

Figure 14: Interaction Between Rays, Camera, and Objects

When a ray intersects an object, the object's surface properties (such as color) are mapped to the corresponding pixel. This process ensures that the rendered image reflects the correct scene geometry and visual attributes.

4.4 Side View of Ray Projections

A side view of the ray projection process shows how rays pass through the image plane and interact with objects in the 3D space. This perspective highlights which rays contribute to pixel data in the rendered image.

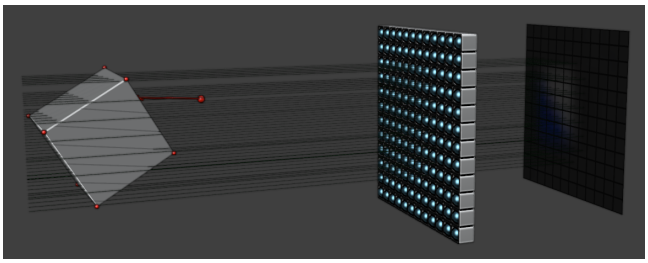

Figure 15: Side View of Ray Projections

—

5 Rendering Results and Examples

5.1 Simple Rendering

Rendering simple objects demonstrates the effectiveness of the RayCast Renderer in capturing basic geometry and lighting at varying resolutions. In this example, a cube is rendered at two different resolutions to showcase the level of detail achievable.

```
void testSpaceCamera() {
    std::cout << "_________Space Test_____________" << std::endl;

    // Define the grid size
    unsigned int size = 100;
    double step = 0.2;
```

Figure 16: Cube Rendered at Low Resolution

```
void testSpaceCamera() {
    std::cout << "_________Space Test_____________" << std::endl;

    // Define the grid size
    unsigned int size = 1000;
    double step = 0.01;
```

Figure 17: Cube Rendered at Medium Resolution

The low-resolution render demonstrates the basic structure of the cube, while the medium-resolution render adds clarity and detail to its edges and surfaces.

—

5.2 Complex Models

The RayCast Renderer can handle complex models with intricate details. Here, a Dahlia flower and the Suzanne model (a common 3D rendering benchmark) are rendered to showcase the renderer's ability to process detailed geometry and surface properties.

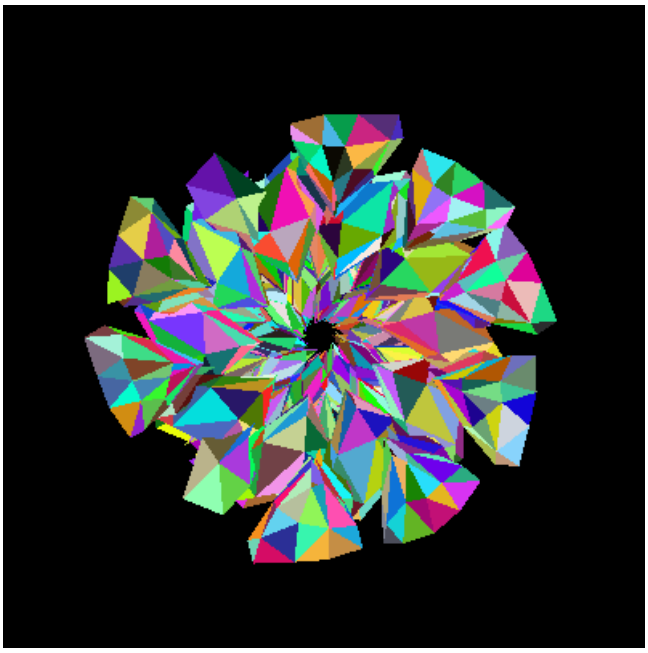

Figure 18: Rendered Dahlia Flower

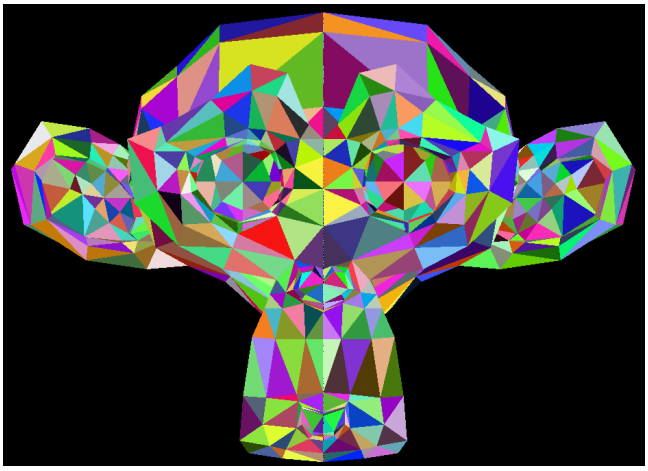

Figure 19: Rendered Suzanne Model

These examples highlight the renderer's capability to capture the intricate geometry of natural and synthetic objects.

—

5.3 Error Demonstrations and Fixes

Rendering errors can arise from various issues, such as incorrect render order. Below, an example of an order-related rendering error is shown, followed by the corrected result after fixing the underlying logic.

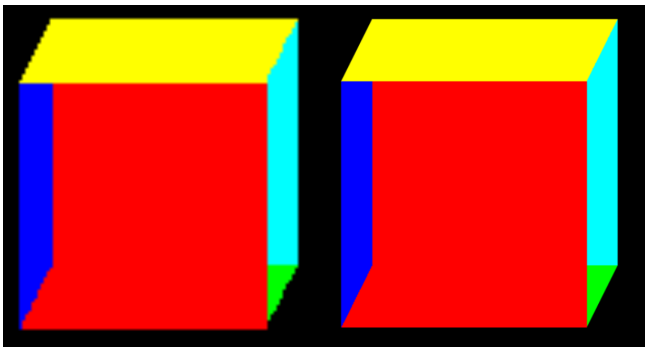

Figure 20: Rendering Error Due to Incorrect Order

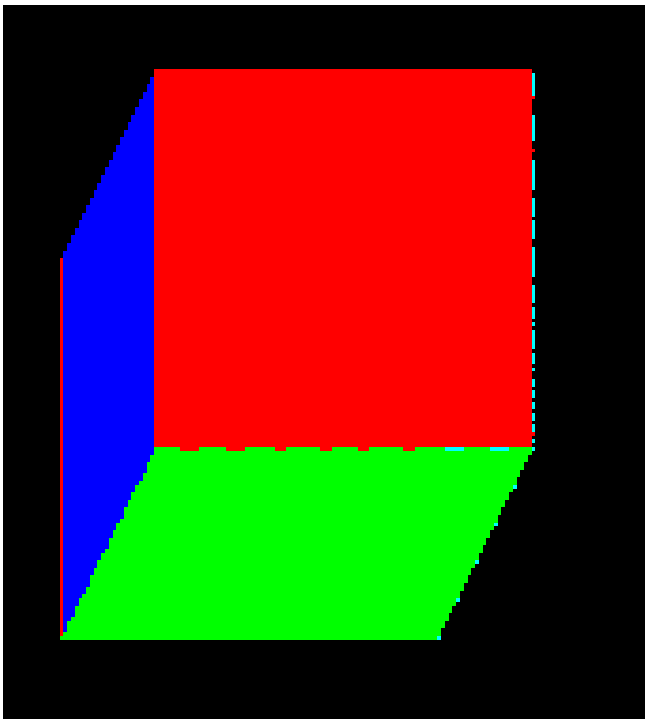

Figure 21: Fixed Rendering After Correcting Render Order

The correction ensures accurate layering of objects in the 3D space, preventing visual artifacts.

—

5.4 Low Ray Density Impact

Ray density plays a significant role in the detail and accuracy of the rendered image. Using a low ray density results in noticeable loss of details, as shown below.

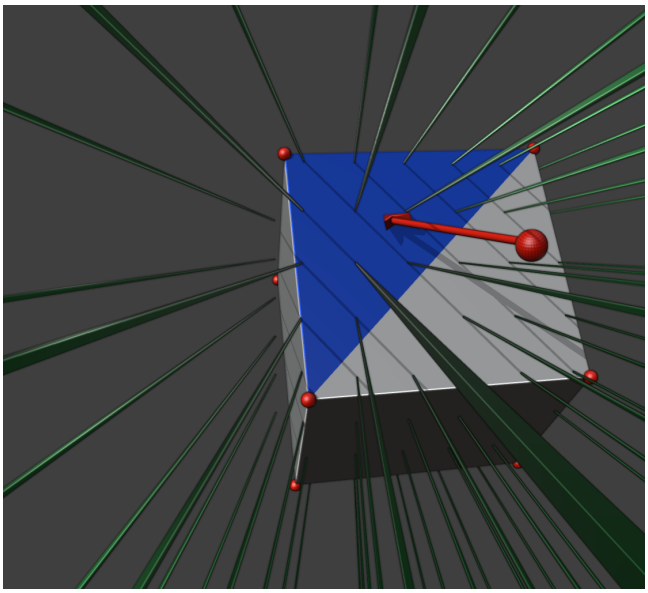

Figure 22: Effect of Low Ray Density on Detail Accuracy

This example illustrates how a low ray density can fail to capture fine details, such as edges or small objects, which are essential for a high-quality render.

—

6 Supporting Visualizations and Code Representations

6.1 Image Class and Pixel Mapping

The `Image` class is responsible for managing the pixel-by-pixel mapping of ray intersection results to the output image. It acts as the container for pixel data, storing color information for each ray intersection and rendering the final image.

Features: 1. **Pixel Storage**: - Each pixel corresponds to a specific ray's intersection result, containing color and intensity data. 2. **Pixel Mapping**:

- The `Image` class ensures accurate mapping of computed ray data to its corresponding pixel on the output grid.

```
13
14  /**
15   * @class image
16   * @brief Represents an image in 2D space.
17   * The image class encapsulates an image defined by a 2D array of pixels.
18   * It provides methods to get and set these attributes, as well as to clear the image.
19   * Inherits from color class.
20   *
21   */
22  class image {
23  private:
24      vector<vector<color>> pixels;
25      unsigned int width;
26      unsigned int height;
```

Figure 23: Image Class Code Representation

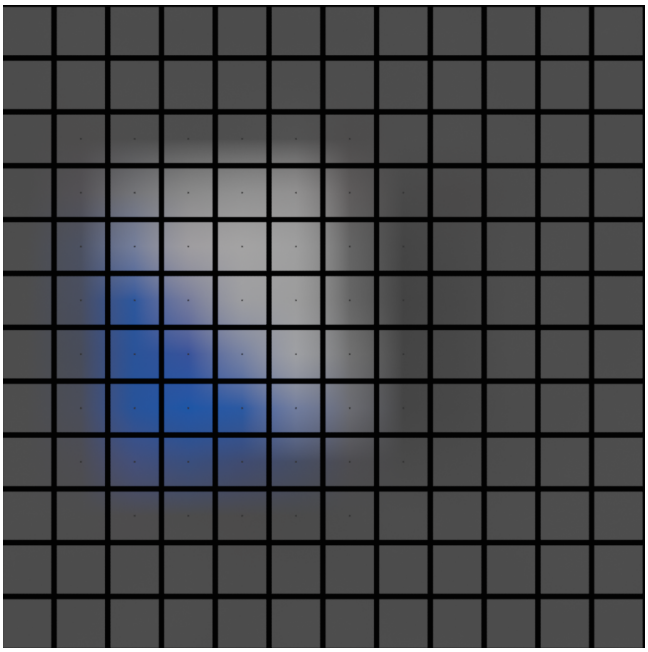

Figure 24: Detailed Code Representation of Image Class

The above figures illustrate the structure and logic behind the `Image` class, showcasing how it facilitates the storage and rendering of pixel data.

—

6.2 MeshReader and 3D File Format Handling

The `MeshReader` class is designed to load and parse external 3D model files. It converts the vertices, faces, and other geometric information into a format that the renderer can use for ray intersection tests.

Features: 1. **File Parsing**: - Supports common 3D file formats, extracting vertex and face data for use in the rendering process. 2. **Data Conversion**: - Converts 3D model data into internal data structures like triangles and points, which the RayCast Renderer can process.

face_coordinates.txt

```
(0.468750,-0.757812,0.242188);(0.500000,-0.687500,0.093750);(0.562500,-0.671875,0.242188)
(-0.500000,-0.687500,0.093750);(-0.468750,-0.757812,0.242188);(-0.562500,-0.671875,0.242188)
(0.562500,-0.671875,0.242188);(0.546875,-0.578125,0.054688);(0.625000,-0.562500,0.242188)
(-0.546875,-0.578125,0.054688);(-0.562500,-0.671875,0.242188);(-0.625000,-0.562500,0.242188)
(0.500000,-0.687500,0.093750);(0.351562,-0.617188,-0.023438);(0.546875,-0.578125,0.054688)
(-0.351562,-0.617188,-0.023438);(-0.500000,-0.687500,0.093750);(-0.546875,-0.578125,0.054688)
(0.437500,-0.765625,0.164062);(0.351562,-0.718750,0.031250);(0.500000,-0.687500,0.093750)
(-0.351562,-0.718750,0.031250);(-0.437500,-0.765625,0.164062);(-0.500000,-0.687500,0.093750)
(0.351562,-0.781250,0.132812);(0.203125,-0.742188,0.093750);(0.351562,-0.718750,0.031250)
(-0.203125,-0.742188,0.093750);(-0.351562,-0.781250,0.132812);(-0.351562,-0.718750,0.031250)
(0.351562,-0.718750,0.031250);(0.156250,-0.648438,0.054688);(0.351562,-0.617188,-0.023438)
(-0.156250,-0.648438,0.054688);(-0.351562,-0.718750,0.031250);(-0.351562,-0.617188,-0.023438)
(0.140625,-0.742188,0.242188);(0.156250,-0.648438,0.054688);(0.203125,-0.742188,0.093750)
(-0.140625,-0.742188,0.242188);(-0.156250,-0.648438,0.054688);(-0.078125,-0.656250,0.242188)
(0.273438,-0.796875,0.164062);(0.140625,-0.742188,0.242188);(0.203125,-0.742188,0.093750)
(-0.140625,-0.742188,0.242188);(-0.273438,-0.796875,0.164062);(-0.203125,-0.742188,0.093750)
(0.242188,-0.796875,0.242188);(0.203125,-0.742188,0.390625);(0.140625,-0.742188,0.242188)
(-0.203125,-0.742188,0.390625);(-0.242188,-0.796875,0.242188);(-0.140625,-0.742188,0.242188)
(0.203125,-0.742188,0.390625);(0.078125,-0.656250,0.242188);(0.140625,-0.742188,0.242188)
(-0.203125,-0.742188,0.390625);(-0.078125,-0.656250,0.242188);(-0.156250,-0.648438,0.437500)
(0.351562,-0.718750,0.453125);(0.156250,-0.648438,0.437500);(0.203125,-0.742188,0.390625)
(-0.351562,-0.718750,0.453125);(-0.156250,-0.648438,0.437500);(-0.351562,-0.617188,0.515625)
```

Figure 25: Example of a Supported 3D File Format

```
/**
 * @file MeshReader.h
 * @brief Defines the MeshReader class for reading text files to extract mesh data.
 */

#ifndef MESH_READER_H
#define MESH_READER_H

#include <string>
#include <vector>
#include <fstream>

#include "point.h"

using namespace std;
/**
 * @class MeshReader
 * @brief Represents a mesh reader for reading text files to extract mesh data.
 * The MeshReader reads a text file that has lines each line contains a set of vertices that
 * represent a face each vector position is seperated by a semicolone and each coordinate is seperated by a comma.
 */
class MeshReader {
public:
    vector<vector<string>> verticesString;
    MeshReader(string filename)
```

Figure 26: MeshReader Class Representation

These visualizations highlight how the `MeshReader` class facilitates the integration of complex 3D models into the rendering pipeline, enabling accurate and efficient handling of external assets.

—