RayCast Renderer Documentation

Silent0Wings

February 6, 2025

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.

Features and Functionality:

- **Representation:** Each `Vec3` instance encapsulates three floating-point values (`x`, `y`, `z`).
- **Mathematical Operations:** Includes addition, subtraction, dot product, cross product, and normalization.
- **Normalization:** Ensures vectors have unit length for accurate calculations.

Example:

- A normalized vector is expressed as:

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

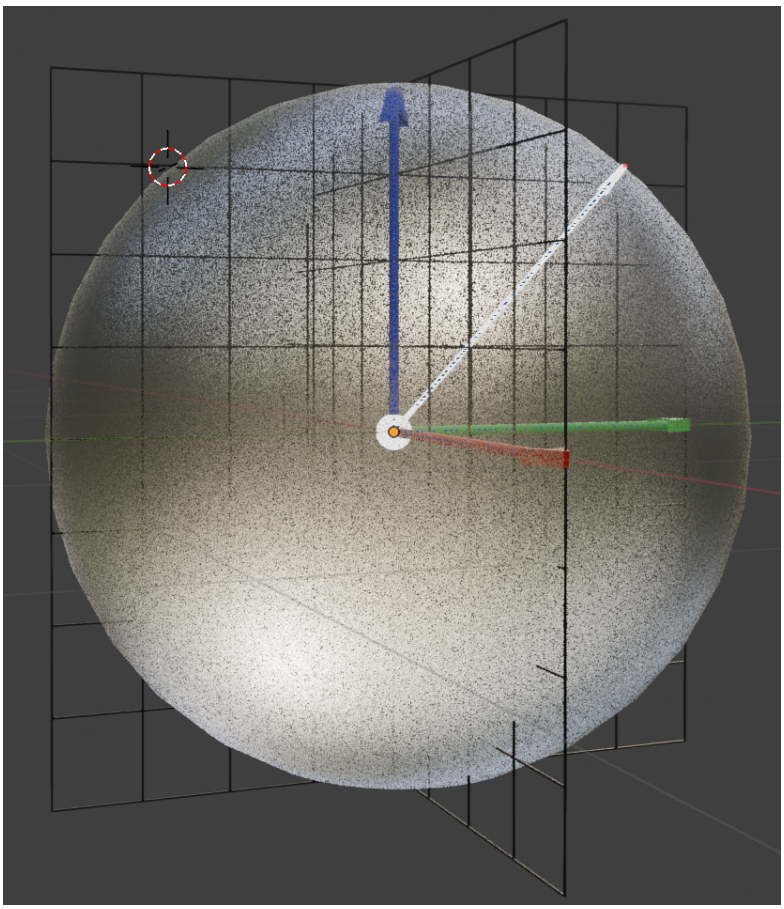

Figure 1: 3D Representation of the Vec3 Class

Importance: The `Vec3` class underpins geometric and directional calculations.

—

2.2 Point Class

The `Point` class defines specific spatial locations in 3D space, forming the fundamental building blocks of the RayCast renderer. These points are used to define triangle vertices, ray origins, and other geometric constructs.

Features and Functionality:

- **Representation:** Each `Point` instance encapsulates three floating-point values (`x`, `y`, `z`), representing a fixed position in a 3D space.
- **Vertex Specification:** All geometric objects are represented as a collection of triangles. The vertices of each triangle are defined using the `Point` class.
- **Ray Origins:** Rays emitted from the camera originate at specific points in the 3D space.
- **Mathematical Operations:** Supports addition, subtraction, and interpolation for ray-triangle intersection tests and transformations.

Logic and Implementation:

- A triangle is represented by three points:

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

- A ray is expressed as:

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

where t determines the position along the ray.

```
10
11  /**
12   * @class point
13   * @brief Represents a point in 3D space.
14   * The point class encapsulates a point defined by x, y, z coordinates.
15   * It provides methods to get and set these attributes, as well as to compute the distance
16   * to another point, translate the point by a vector, compute the midpoint between two points,
17   * and overload operators for addition, subtraction, and multiplication.
18   * Inherits from vec3 class.
19   */
20  class point : public vec3 {
21  public:
22      // Default constructor (origin point)
23      point() : vec3(0, 0, 0) {}
24
```

Figure 2: Point Class Code Representation

Visualization of Point Class Usage:

- **Point Definition:** Each vertex of a triangle is defined as a `Point`.

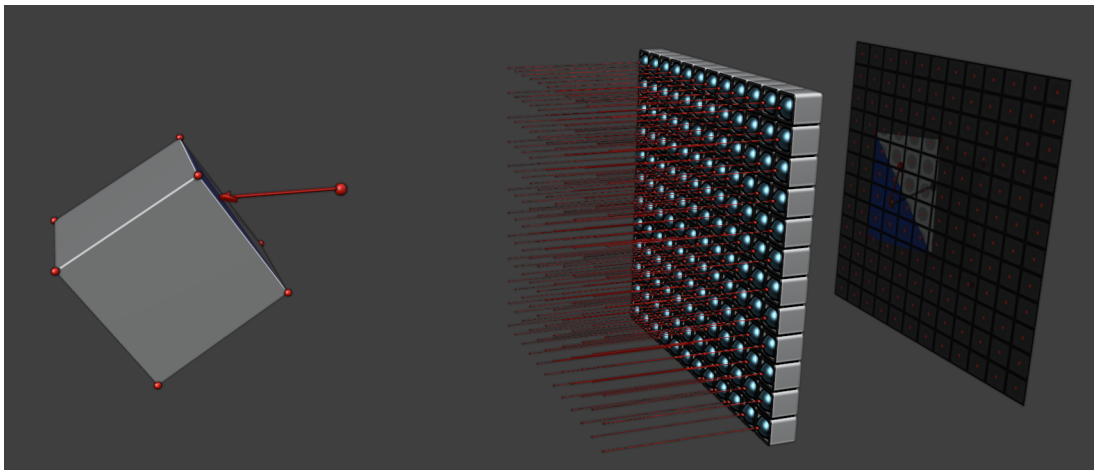

Figure 3: Spatial Layout of Points, Camera, and Objects

- **Ray Originating at a Point:** Points align ray origins with the camera's image plane.

```
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 4: Camera Emitting Rays from Specific Points

—

2.3 Color Class

The `Color` class is responsible for handling the RGB values used to define the surface colors of objects and the pixel colors in the rendered image. It plays a crucial role in producing visually accurate and appealing renders by enabling color assignment during ray-object intersections.

Features and Functionality:

- **RGB Representation:** Each `Color` instance stores three values (`R`, `G`, `B`) representing the red, green, and blue color intensities, typically ranging from 0 to 255.
- **Color Assignment:** During the rendering process, the `Color` class assigns specific colors to pixels based on the surfaces of intersected objects.
- **Blending and Interpolation:** Supports blending multiple colors to simulate effects such as shading or transitions between materials.

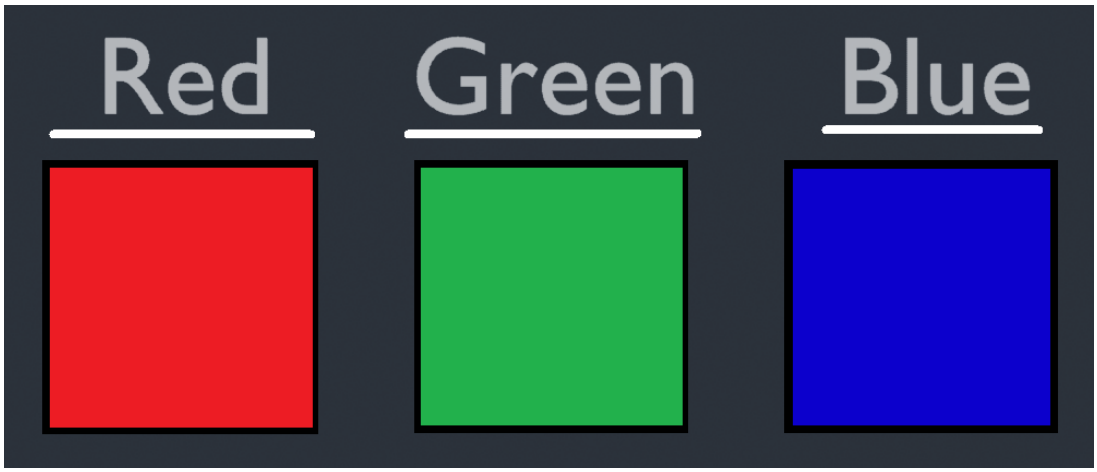

Figure 5: Representation of the Color Class Showing RGB Components

Visualization of Color Application:

- **Object Coloring:** Each triangle of an object is assigned a specific `Color`, creating a cohesive appearance when rendered.

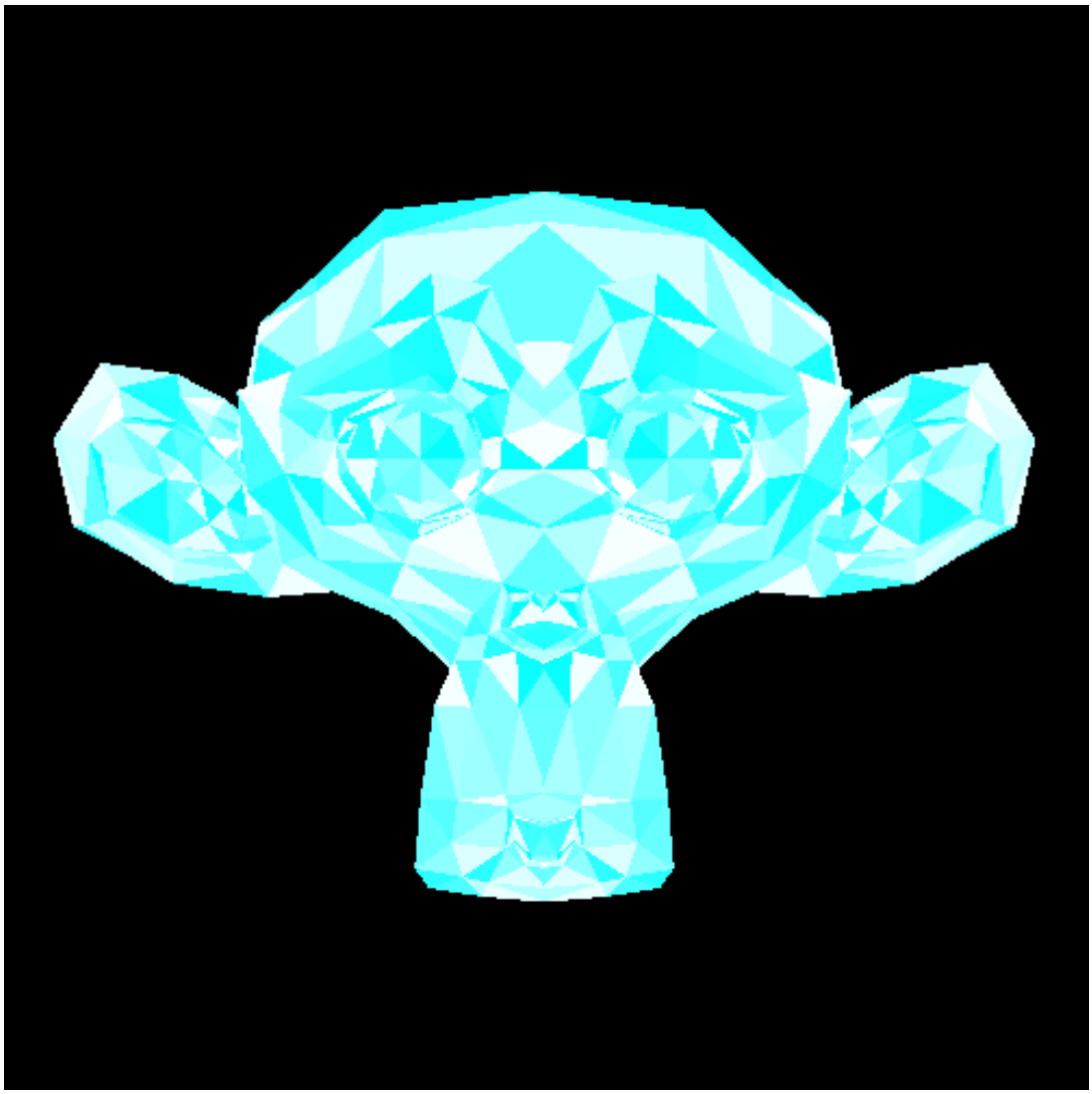

Figure 6: Suzanne Model Rendered with a Uniform Blue Color

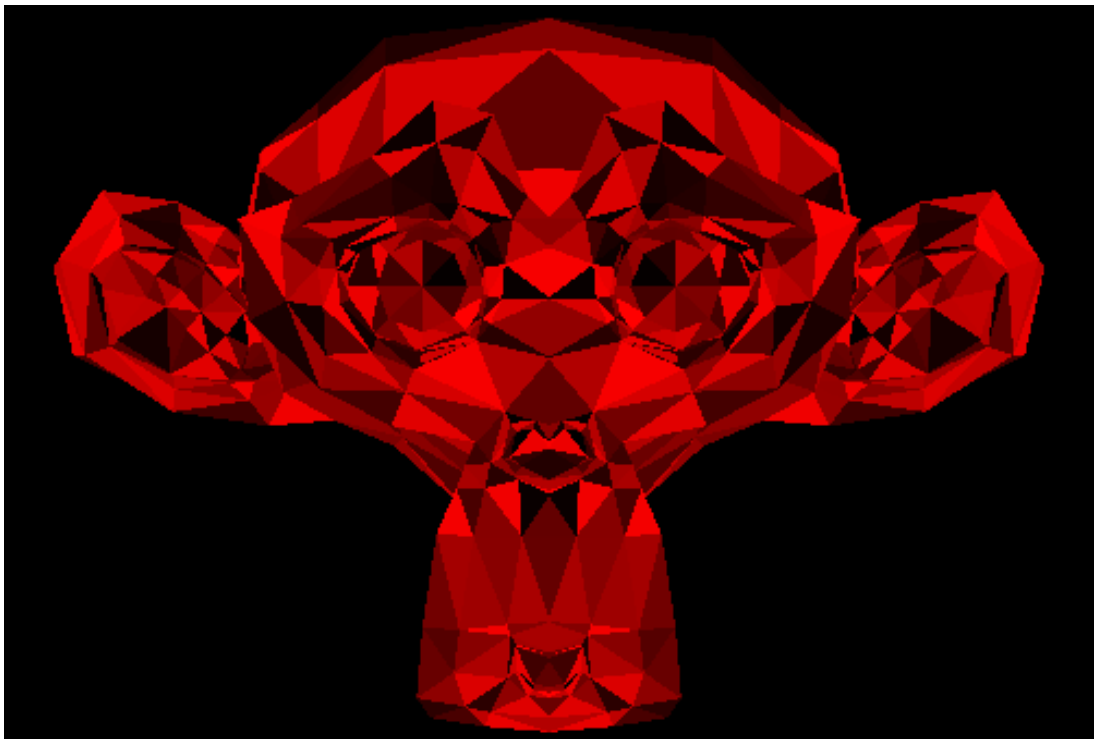

Figure 7: Suzanne Model Rendered with a Uniform Red Color

- **Corrected Render:** After fixing color assignment logic, objects are rendered with accurate colors.

Rendering Output with wrong Logic:

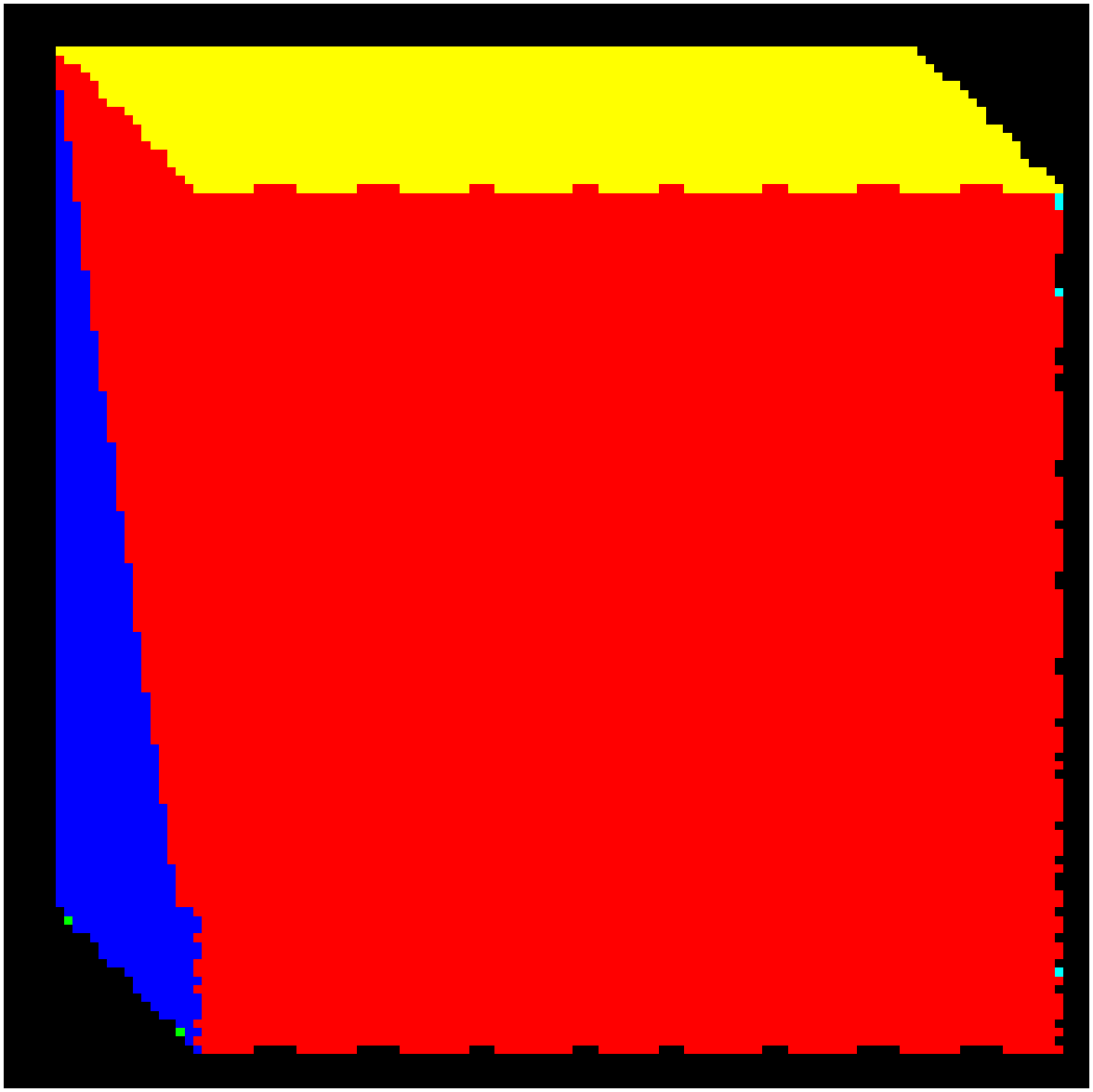

Figure 8: Final Render of Cube with wrong Color Assignments

Initial Rendering Error and Fix:

```
void getPixel(unsigned int i, unsigned int j, object obj, ray r1) {
    // Iterate through the color map vertices
    for (auto const& x : obj.colorMap) {
        // Get the vertices as an array
        array<point, 3> arr = {x.first[0], x.first[1], x.first[2]};

        // Check if the ray intersects with the current triangle
        point* val = gmath::intersect3d1(r1, arr.data());

        if (val != nullptr) {
            // Set the pixel in the image
            double leng = gmath::distance(r1.getOrigine(),*(val));
            cout << "Length: " << leng << endl;


            img.set(i, j, x.second);
            distance[i][j] = leng;
            // Debugging: Output the intersection and color
            //cout << "Intersection at Pixel (" << i << ", " << j << "): " << *val << endl;
            //cout << "Color: " << x.second << endl;


            // Clean up memory and exit the loop (first valid intersection found)
            delete val;
            return;
        }
    }
```

Figure 9: Original Rendering Output Highlighting Color Assignment Errors

Code Fix for Correct Color Assignments:

```
void getPixel(unsigned int i, unsigned int j, object obj, ray r1) {
    double distance=99999999999999999;
    // Iterate through the color map vertices
    for (auto const& x : obj.colorMap) {
        // Get the vertices as an array
        array<point, 3> arr = {x.first[0], x.first[1], x.first[2]};

        // Check if the ray intersects with the current triangle
        point* val = gmath::intersect3d1(r1, arr.data());

        if (val != nullptr) {
            // Set the pixel in the image
            double leng = gmath::distance(r1.getOrigine(),*(val));
            delete val;
            //cout << "Length: " << leng << endl;
            //cout << "Distance: " << distance[i][j] << endl;

            if(distance >= leng)
            {
                distance = leng;
                img.set(i, j, x.second);
                // Debugging: Output the intersection and color
                //cout << "Intersection at Pixel (" << i << ", " << j << "): " << *val << e
                //cout << "Color: " << x.second << endl;
            }

            //cout << "distance: " << distance[i][j] << endl;
            // Clean up memory and exit the loop (first valid intersection found)
        }
    }

// If no intersection is found, set a default color (e.g., black)
img.set(j, i, defaultColor); // Default: Black
// cout << "No intersection at Pixel (" << i << ", " << j << ")" << endl;
}
```

Figure 10: Code Implementation for Fixing Color Assignment Errors

- **Fixed Intersection Logic:** After correcting the intersection logic, the rendered result accurately displays the object's geometry.

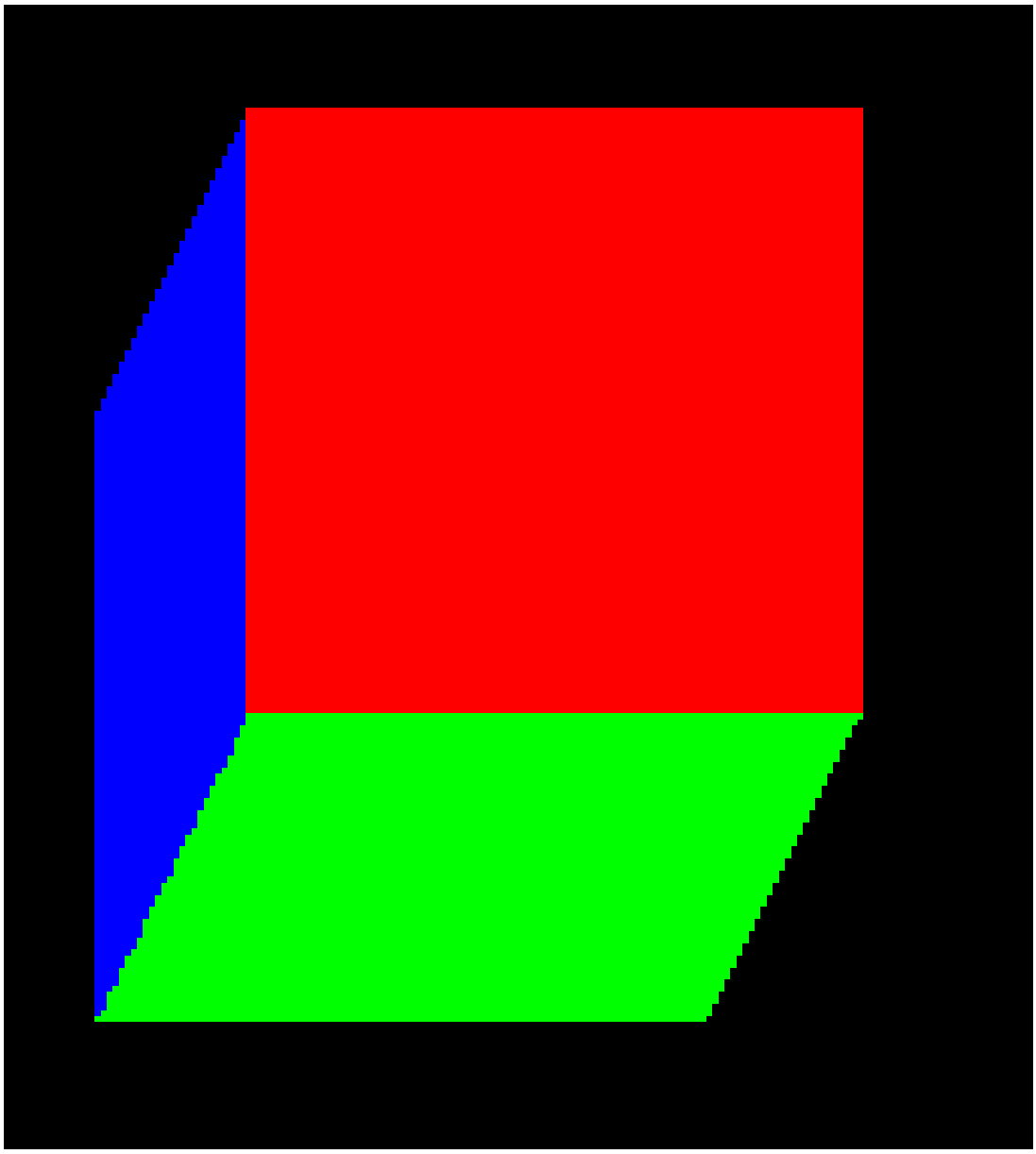

Figure 11: Corrected Rendering Result After Fixing Intersection Logic

Significance of the Fix:

- Accurate color assignments for each triangle in the scene.
- Improved visual fidelity and realism in the rendered image.
- A reliable framework for rendering more complex scenes without similar errors.

Importance of the Color Class:

- Enables realistic visualization of objects by assigning colors to surfaces.
- Ensures that rendering results are visually coherent and meet aesthetic goals.

—

2.4 Object Class

The `Object` class represents 3D models within the RayCast Renderer. It defines the geometry, color mapping, and properties of objects in the scene. Objects are composed of multiple triangles, which are the fundamental building blocks for 3D models.

Features and Functionality:

- **Triangle Representation:** Each `Object` is modeled as a collection of triangles, with each triangle defined by three vertices (`Point` instances).
- **Color Mapping:** The `Object` class associates each triangle with a specific color (`Color` instance), allowing precise control over the visual appearance of the object.

- **Intersection Testing:** The `Object` class supports efficient ray-triangle intersection testing, enabling the renderer to identify which parts of the object are visible in the scene.

Figure 12: Representation of the Object Class Code and Its Components

Logic and Implementation:

- Each object consists of a list of triangles, where each triangle stores:
  - Its three vertices (instances of the `Point` class).
  - Its associated color (instance of the `Color` class).
- During rendering, the ray-object intersection logic evaluates each triangle for potential intersections with the ray.

```
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 13: Detailed Code Representation of the Object Class

Visualization of Object Class Usage:

- **Simple Geometry:**
  Objects such as cubes or planes are created using a small number of triangles. For example:

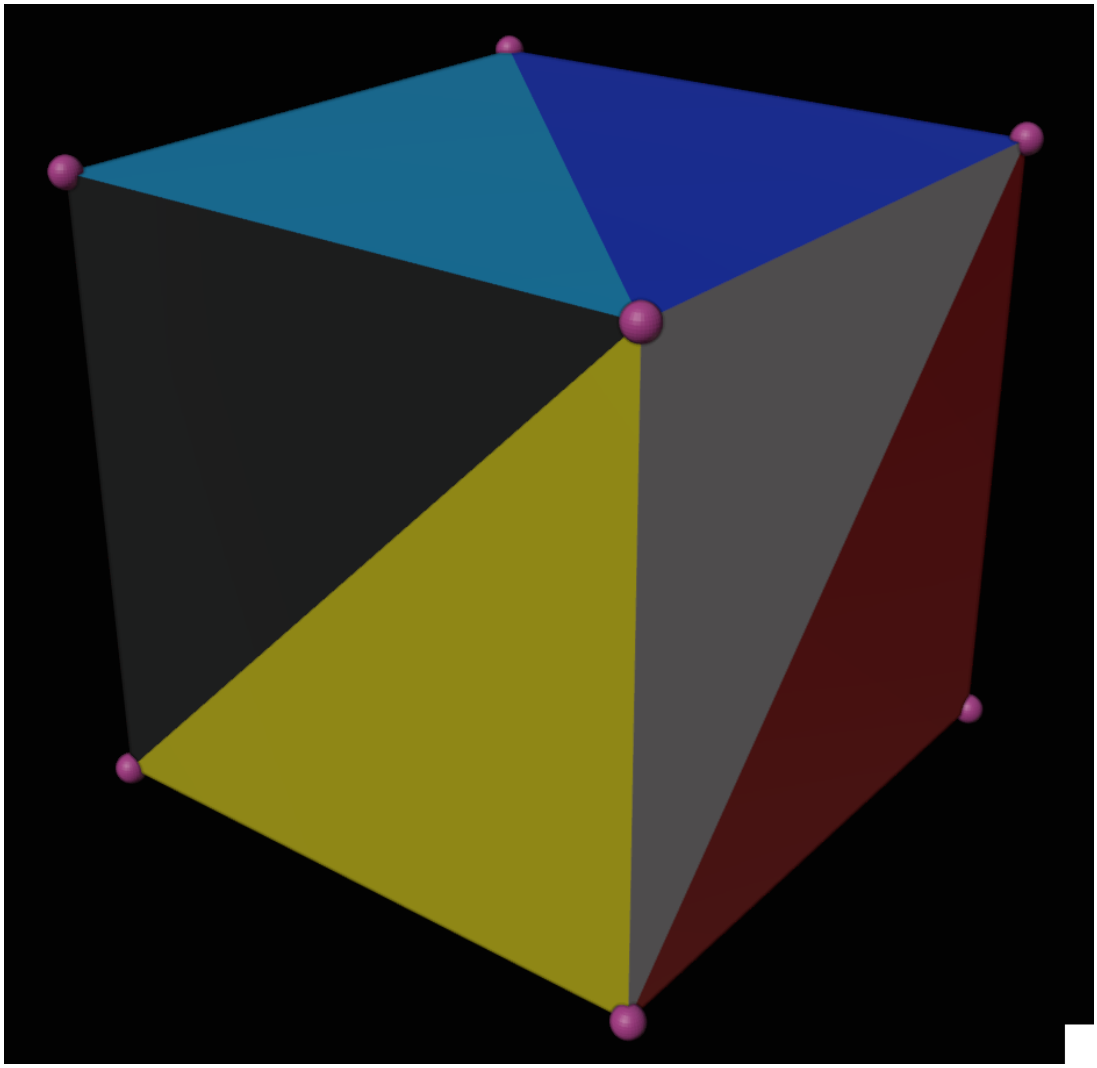

Figure 14: Low-Resolution Cube Render Composed of Several Triangles

- **Complex Geometry:**
  More detailed models, such as the Suzanne model, are created using a large number of smaller triangles.

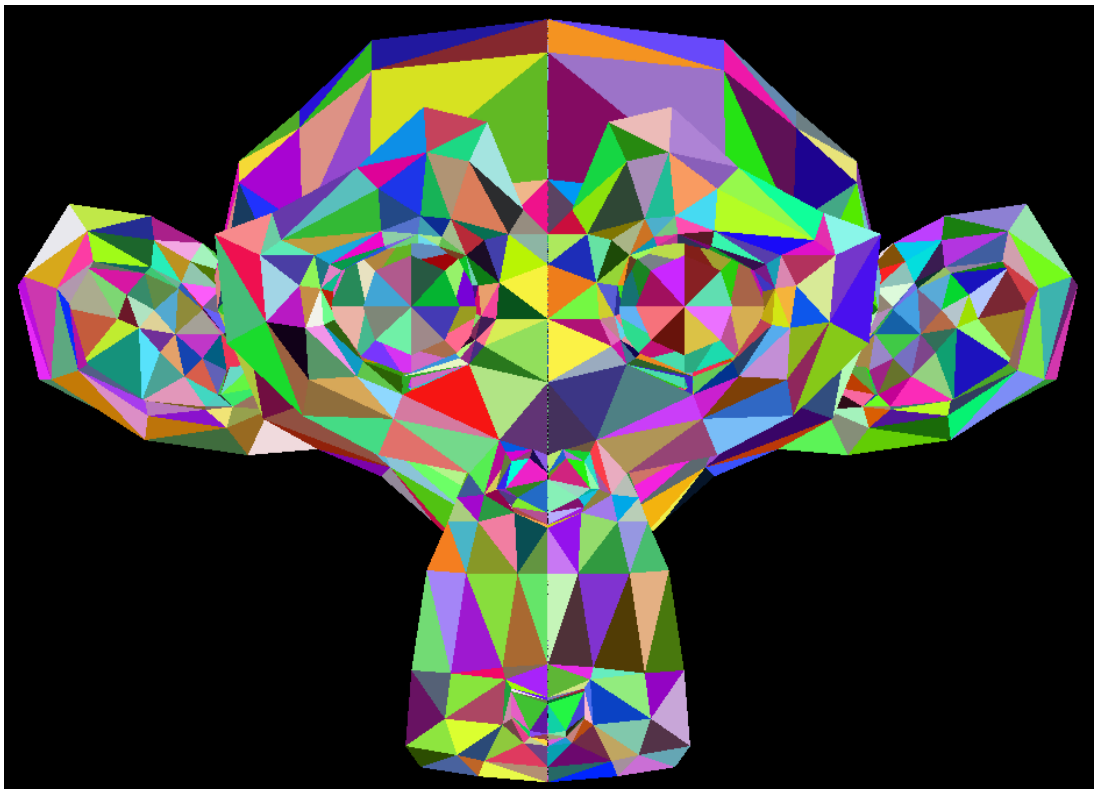

Figure 15: Suzanne Model Rendered with Triangular Mesh Representation

- **Wire frame Representation:**
  The `Object` class supports visualization of the object's wire frame, showing how triangles form the overall structure.

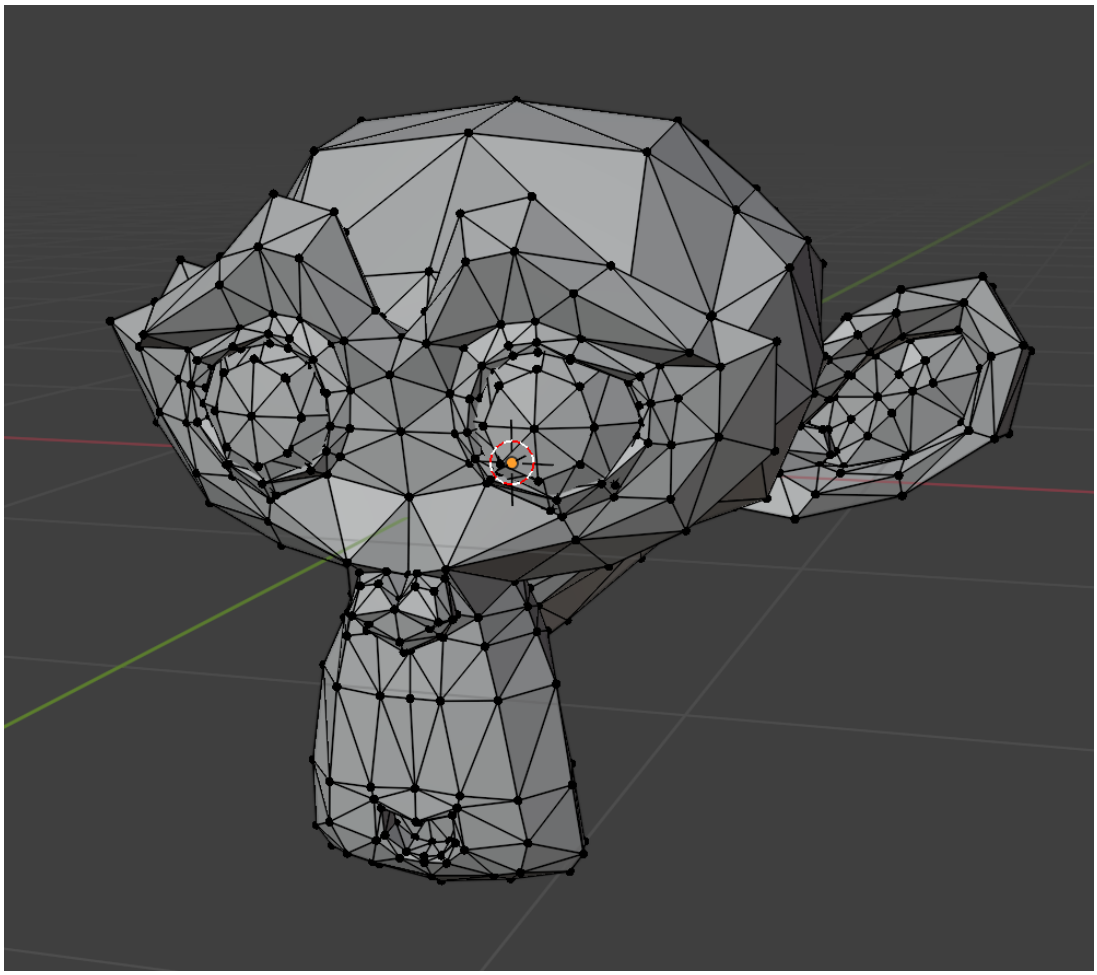

Figure 16: Wireframe View of the Suzanne Model

- **Multiple Objects:**
  The RayCast Renderer supports scenes with multiple objects, each defined as an instance of the `Object` class.

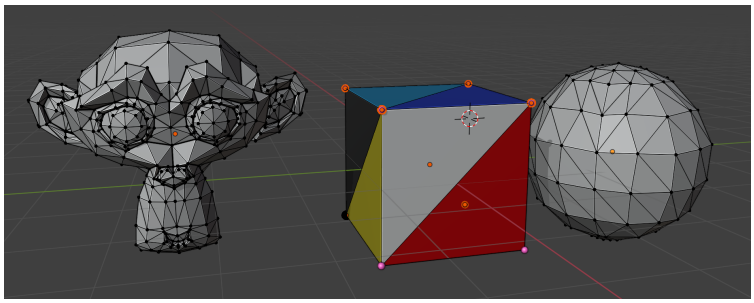

Figure 17: Scene with Multiple Objects Defined Using the Object Class

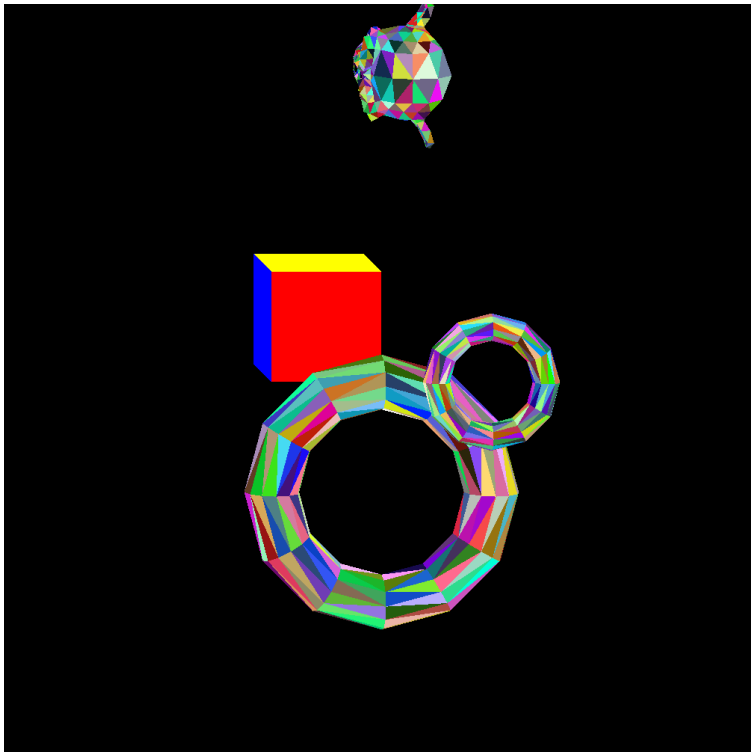

Figure 18: Actuall render of multiple object with accurate overlay and render order

Importance of the Object Class:

- Provides the foundation for creating 3D geometry in the scene.
- Supports efficient ray-triangle intersection testing for accurate rendering.
- Enables detailed color mapping, wireframe visualization, and rendering of complex shapes.

—

2.5 Ray Class

The `Ray` class is the core medium for transmitting information from the camera to the scene in the RayCast Renderer. It defines a ray as an origin point and a direction vector. Each ray corresponds to a single pixel in the final image and determines visibility, color, and shading by interacting with objects in the 3D space.

Features and Functionality:

- **Definition:** A `Ray` is defined by:
  - `Point`: The origin of the ray, typically corresponding to the camera's position or the image plane.
  - `Vec3`: The direction of the ray, indicating its trajectory in the 3D space.
- **Mathematical Representation:** A ray is mathematically expressed as:

$$\text{Ray}(t) = \text{Origin} + t \cdot \text{Direction},$$

  where t is a scalar that determines the position along the ray.
- **Ray Casting:** Rays are cast from the camera through the image plane into the 3D scene to determine intersections with objects.

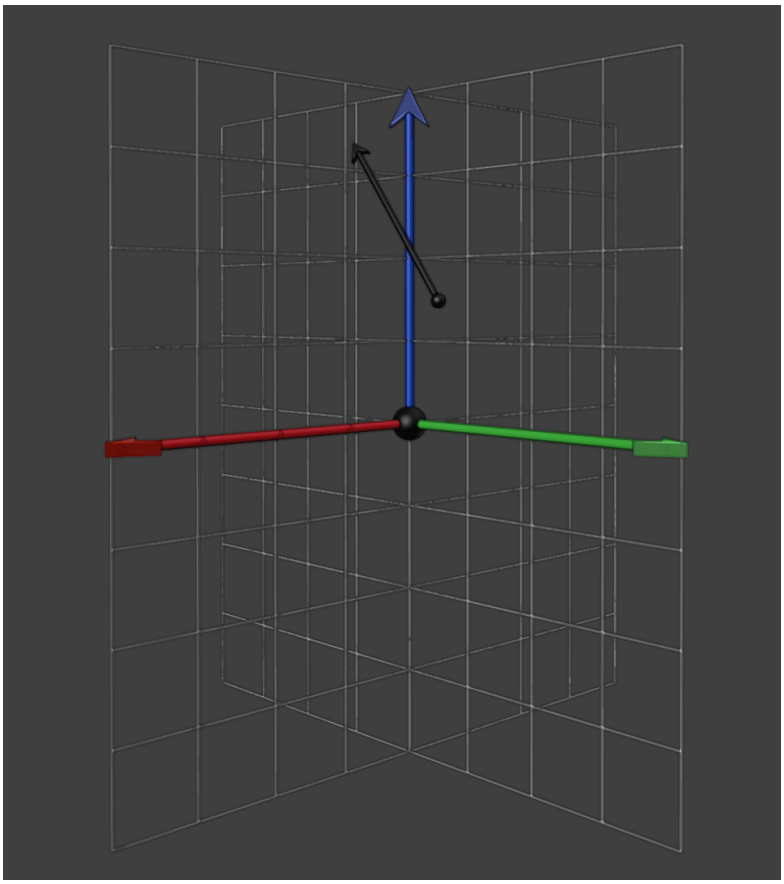

Figure 19: Representation of the Ray Class Showing Its Components

Logic and Implementation:

- **Ray Generation:** The camera generates a grid of rays, with each ray corresponding to a pixel in the image. These rays are then tested against the scene to determine which objects are visible.
- **Ray-Object Interactions:** Each ray is checked for intersections with objects in the scene. The intersection logic determines:
  - Which object the ray hits first.
  - The exact point of intersection.
  - The color or shading of the intersected object at the intersection point.

```
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 20: Detailed Code Implementation of the Ray Class

Visualization of Ray Class Usage:

- **Realistic Projection:** Demonstrates how rays interact with objects in the scene to produce accurate render results.

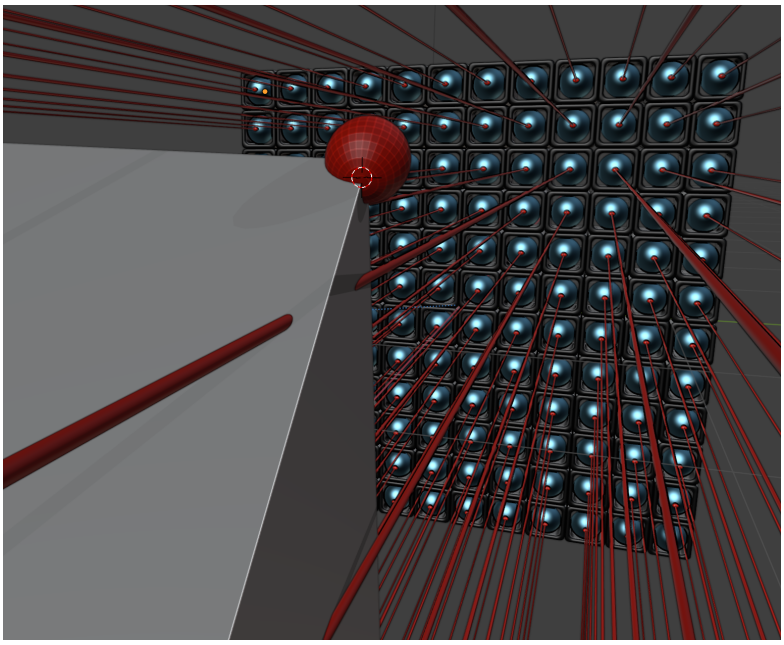

Figure 21: Realistic Ray Projection Respecting the 12x12 Resolution

Importance of the Ray Class:

- Forms the backbone of the rendering process by transmitting visual information from the camera to the scene.
- Enables accurate visibility and shading calculations through ray-object intersections.
- Serves as a foundational element for more advanced rendering techniques, such as reflection and refraction.

—

2.6 Ray Projection

Ray projection is the process by which the camera generates a grid of rays corresponding to pixels on the image plane. These rays traverse the 3D scene, determining which objects are visible and their corresponding pixel colors.

Features and Functionality:

- **Ray Generation:** The camera creates rays for each pixel in the image. The direction of each ray is calculated based on the camera's orientation and field of view.
- **Pixel Mapping:** Each ray corresponds to a specific pixel in the final image, ensuring accurate rendering.
- **Depth Perspective:** Ray projection incorporates perspective distortion, making closer objects appear larger in the rendered image.

Logic and Implementation:

- **Grid-Based Projection:** Rays are cast through a grid representing the image plane. Each grid cell corresponds to one pixel.
- **Direction Calculation:** The direction of each ray is determined by mapping pixel coordinates to 3D space using the camera's position, orientation, and field of view.
- **Pixel-Color Mapping:** The color of each pixel is updated based on the intersection of its corresponding ray with objects in the scene.

Importance of Ray Projection:

- Maps the 2D image plane to the 3D scene, enabling accurate pixel rendering.
- Captures depth and perspective, essential for lifelike rendering.
- Forms the basis for advanced effects like reflections and refractions by extending the ray projection logic.

—

2.7 Camera Class

The `Camera` class represents the virtual viewpoint in the 3D space. It serves as the origin for all rays, projecting them into the scene to determine visibility, shading, and pixel colors. The camera's position, orientation, and field of view control how the scene is captured and rendered.

Features and Functionality:

- **Positioning:** The camera is defined by its position in the 3D space, represented as a `Point`.
- **Orientation:** The camera's orientation is controlled by a directional vector (`Vec3`), which determines where the camera is looking.
- **Field of View (FOV):** Defines the angular extent of the scene captured by the camera. A wider FOV results in more of the scene being visible but may introduce distortion.
- **Ray Generation:** The camera generates rays corresponding to each pixel in the virtual image plane, which are cast into the 3D scene.

```
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 22: Representation of the Camera Class

Logic and Implementation:

- **Ray Casting:** The camera projects rays through a grid corresponding to the image resolution. Each ray is defined by:

$$\text{Ray}(t) = \text{Origin} + t \cdot \text{Direction}.$$

- **Pixel Mapping:** Each ray corresponds to a specific pixel on the virtual image plane. The color of the pixel is determined based on the intersection of the ray with objects in the scene.
- **Projection:** The camera's FOV, combined with its resolution, determines the spacing and direction of the rays, ensuring accurate perspective projection.

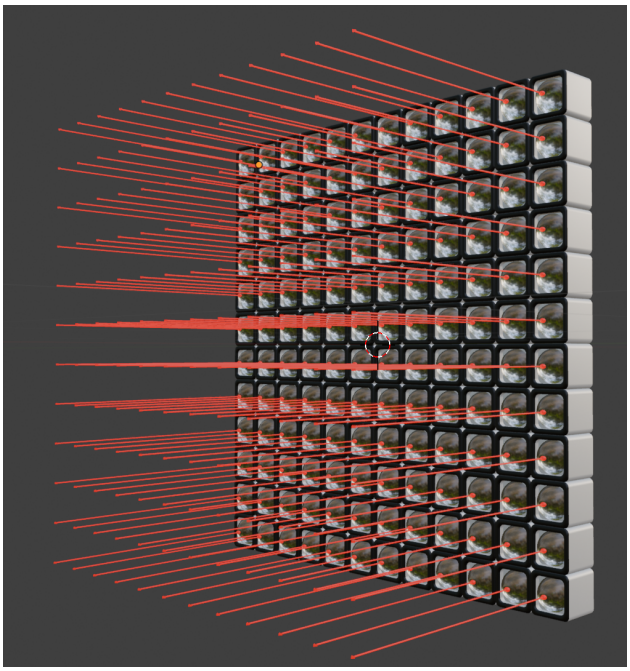

Figure 23: Illustration of Camera Functionality and Ray Generation

Visualization of Camera Class Usage:

– **Front View of Ray Casting:** A front view of the camera showing rays originating from its position and projecting into the 3D space.

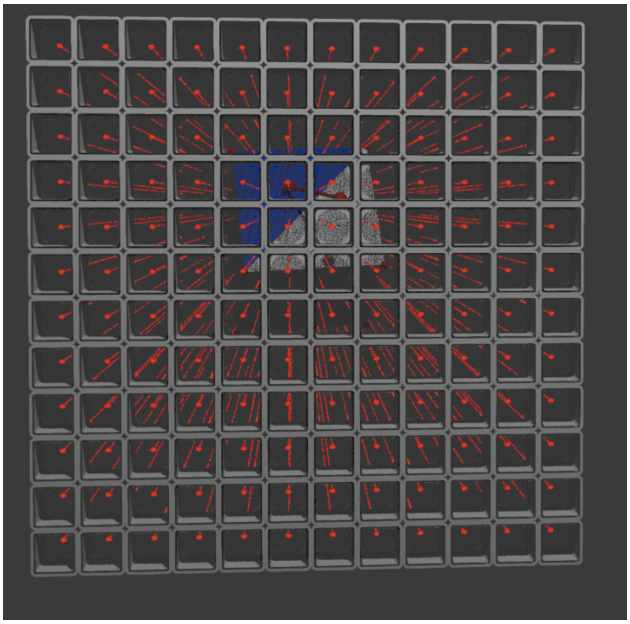

Figure 24: Front View of Camera Projecting Rays into the Scene

– **Side View of Ray Paths:** Shows how rays pass through the image plane and intersect with objects in the 3D scene.

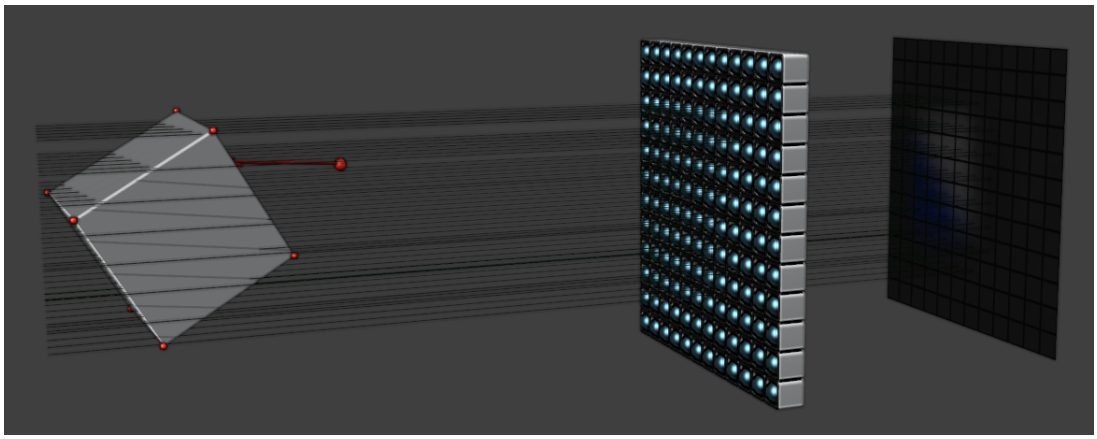

Figure 25: Side View of Ray Paths Projecting Pixel Colors

Importance of the Camera Class:

- Defines the viewpoint and perspective for the rendered scene.
- Controls how rays are cast into the 3D space, determining pixel visibility and shading.
- Serves as the foundation for advanced techniques such as depth of field and camera motion effects.

—

2.8 Intersection Logic

The intersection logic is a crucial computational process used to determine whether a ray intersects an object (triangle) in the 3D scene. This process identifies visible objects, calculates shading, and generates the final image.

Features and Functionality:

- **Intersection Tests:** The renderer evaluates whether and where a ray intersects each triangle using geometric algorithms.
- **Closest Intersection:** For each ray, only the closest intersection (if any) is recorded to ensure correct visibility.
- **Ray-Triangle Algorithm:** The logic determines ray-triangle intersections by:
  - Solving for t in the ray equation:
    $$\text{Ray}(t) = \text{Origin} + t \cdot \text{Direction}.$$
  - Ensuring $t > 0$, so the intersection is in front of the ray origin.
  - Verifying that the intersection lies within the triangle's bounds using barycentric coordinates.

Mathematical Basis:

- **Plane Definition:** The plane containing the triangle ABC is defined as:
  $$ax + by + cz + d = 0,$$
  where a, b, c are the components of the plane's normal vector, and d is the plane's offset from the origin.
- **Ray Definition:** The ray $D(t)$ is parameterized as:
  $$D(t) = D_0 + t \cdot \vec{D},$$
  where D_0 is the ray's origin, $\vec{D}$ is its direction vector, and t is a scalar representing the position along the ray.
- **Intersection Calculation:** To compute the intersection:
  1. Substitute the ray equation $D(t)$ into the plane equation.
  2. Solve for t, ensuring $t > 0$.
  3. Verify that the intersection lies within the triangle using barycentric coordinates or edge-based checks.

Intersection Parameter Calculation:

$$t = \frac{\vec{n} \cdot (\vec{A} - \vec{O})}{\vec{n} \cdot \vec{d}}$$

where:

- $\vec{n}$: Normal vector of the plane.

- $\vec{A}$: A vertex of the triangle.
- $\vec{O}$: Ray origin.
- $\vec{d}$: Ray direction.

Once t is determined, the ray $L(t)$ can be parameterized as:

$$L(t) = \begin{cases} a_1 + b_1 \cdot t = x \\ a_2 + b_2 \cdot t = y \\ a_3 + b_3 \cdot t = z \end{cases}$$

Here, a_1, a_2, a_3 represent the initial coordinates, and b_1, b_2, b_3 correspond to the direction vector components scaled by t.

Logic and Implementation:

- Iterate through all triangles in the scene for each ray.
- For each triangle:
  - Calculate the intersection point using the ray-triangle algorithm.
  - Verify the intersection is inside the triangle.
  - Compare the t value with previously recorded intersections to find the closest point.
- If an intersection is found, update the pixel color with the triangle's color.

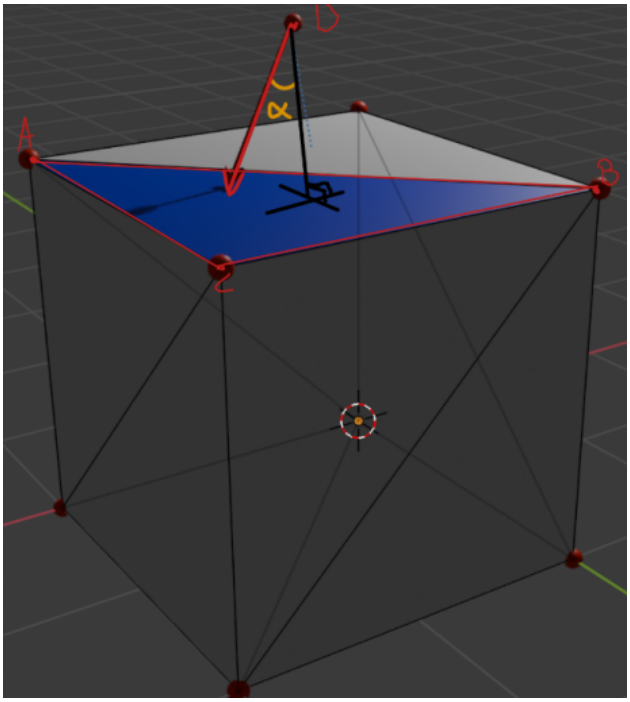

Figure 26: Step 1: Ray-Triangle Intersection Logic Overview

Visualization of Intersection Logic in Action:

- **Error Due to Incorrect Logic:** Mistakes in intersection calculations can result in rendering errors:

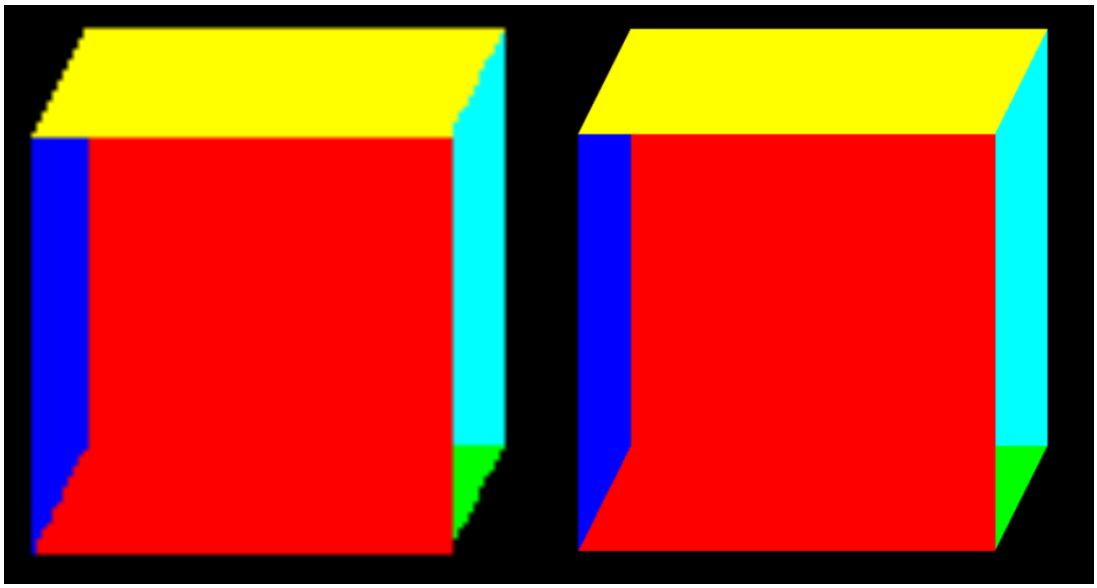

Figure 27: Rendering Error Caused by Incorrect Intersection Logic

Importance of Intersection Logic:

- Identifies visible objects and surfaces in the scene.
- Ensures correct layering and depth by selecting the closest intersection.
- Provides the foundation for advanced rendering techniques, such as reflections and shadows.

—

2.9 Space Layout

The 3D space layout defines the spatial organization of objects, the camera, and their relationships. It ensures that rays are cast in the correct direction relative to the scene, enabling accurate rendering.

Features and Functionality:

- **Object Placement:** Objects are positioned within a coordinate system that represents the 3D space.
- **Camera Positioning:** The camera is placed in the 3D space, with its orientation and field of view determining the direction of rays.
- **Coordinate System:** The space layout uses a Cartesian coordinate system (x, y, z) to define object positions, camera orientation, and ray directions.
- **Scene Composition:** The space layout facilitates the composition of scenes with multiple objects, ensuring that relationships like depth and perspective are accurately represented.

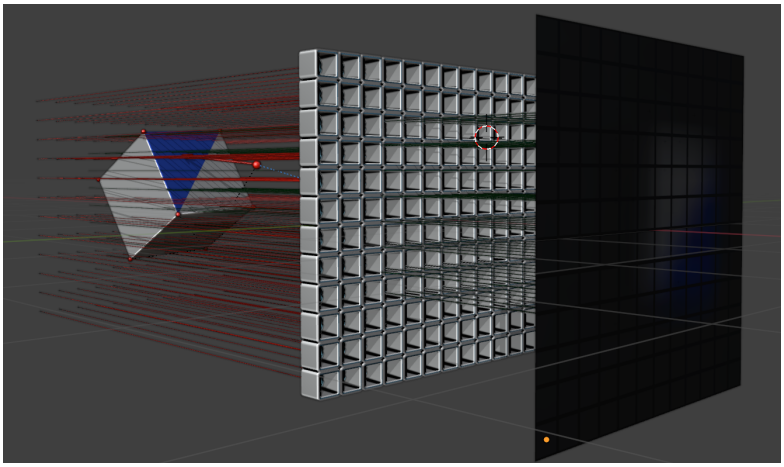

Figure 28: Logical Layout of Camera, Objects, and Rays in 3D Space

Logic and Implementation:

– **Object Placement:** Objects are placed using their position vectors (`Point` instances), which define their location in the 3D space.
– **Camera Orientation:** The camera's position and orientation are defined using a position vector and direction vector (`Vec3`), ensuring that rays are cast toward the intended objects.
– **Ray Casting:** Rays are generated from the camera's position and projected into the scene based on the image plane grid. These rays traverse the 3D space and test for intersections with objects.

```
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 29: Camera Positioned in Space, Emitting Rays Toward Objects

Visualization of Space Layout:

– **Scene Composition:** The layout organizes multiple objects within the same 3D space, ensuring proper alignment and depth perception.

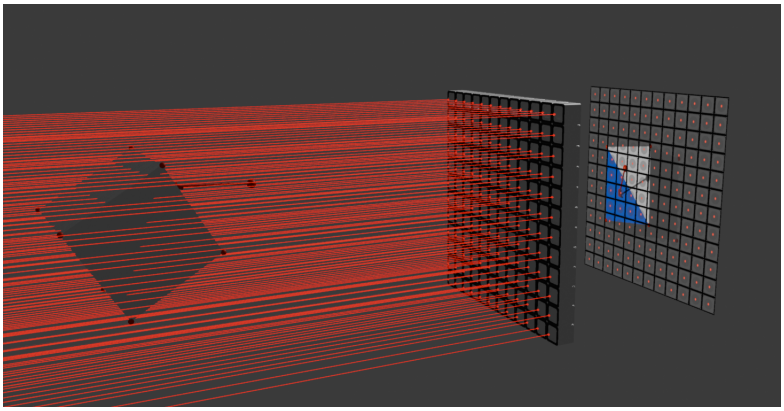

Figure 30: Camera Casting Rays into a Scene with Multiple Objects

– **Ray Interaction with Objects:** Demonstrates how rays traverse the 3D space and intersect with objects to determine visibility and shading.

Importance of Space Layout:

– Ensures that objects, camera, and rays are correctly aligned for accurate rendering.
– Facilitates realistic depth and perspective, enabling lifelike scene composition.
– Provides a logical framework for organizing complex scenes with multiple objects.

—

3 Rendering Results and Examples

The rendering results showcase the effectiveness of the RayCast Renderer in capturing object geometry, perspective, and shading. By demonstrating renders of simple and complex models, this section highlights the renderer's capabilities and the impact of parameters such as resolution and ray density.

—

3.1 Simple Rendering

Simple rendering involves basic geometric objects, such as cubes, rendered at varying resolutions. These examples illustrate how the renderer translates 3D geometry into 2D pixels with increasing levels of detail.

Examples:

– **Low Resolution:** Demonstrates the basic structure of a cube with minimal detail.

```
// Create object vertices for multiple triangles in the z = 0 plane
std::vector<std::vector<point>> vertices = {
    {point(0, 0, 0), point(1, 0, 0), point(0, 1, 0)},  // Triangle 1
    {point(1, 0, 0), point(1, 1, 0), point(0, 1, 0)},  // Triangle 2
    {point(1, 2, 0), point(1, 1, 2), point(0, 1, 0)},  // Triangle 3
    {point(0, 0, 0), point(1, 0, 0), point(0, 1, 0)},  // Triangle 4
    {point(0, 0, 0), point(1, 0, 0), point(0, 1, 0)},  // Triangle 5
    {point(0, 0, 0), point(1, 0, 0), point(0, 1, 0)},  // Triangle 6
    {point(2, 2, 0), point(3, 2, 0), point(2, 3, 0)},  // Triangle 7
    {point(3, 2, 0), point(3, 3, 0), point(2, 3, 0)},  // Triangle 8
    {point(4, 4, 0), point(5, 4, 0), point(4, 5, 0)},  // Triangle 9
    {point(5, 4, 0), point(5, 5, 0), point(4, 5, 0)},  // Triangle 10
    {point(6, 6, 0), point(7, 6, 0), point(6, 7, 0)},  // Triangle 11
    {point(7, 6, 0), point(7, 7, 0), point(6, 7, 0)},  // Triangle 12
    {point(8, 8, 0), point(9, 8, 0), point(8, 9, 0)},  // Triangle 13
    {point(9, 8, 0), point(9, 9, 0), point(8, 9, 0)},  // Triangle 14
    {point(10, 10, 0), point(11, 10, 0), point(10, 11, 0)},  // Triangle 15
    {point(11, 10, 0), point(11, 11, 0), point(10, 11, 0)},  // Triangle 16
    {point(12, 12, 0), point(13, 12, 0), point(12, 13, 0)},  // Triangle 17
    {point(13, 12, 0), point(13, 13, 0), point(12, 13, 0)},  // Triangle 18
    {point(14, 14, 0), point(15, 14, 0), point(14, 15, 0)},  // Triangle 19
    {point(15, 14, 0), point(15, 15, 0), point(14, 15, 0)},  // Triangle 20
    {point(16, 16, 0), point(17, 16, 0), point(16, 17, 0)},  // Triangle 21
    {point(17, 16, 0), point(17, 17, 0), point(16, 17, 0)},  // Triangle 22
    {point(18, 18, 0), point(19, 18, 0), point(18, 19, 0)},  // Triangle 23
    {point(19, 18, 0), point(19, 19, 0), point(18, 19, 0)},  // Triangle 24
    {point(20, 20, 0), point(21, 20, 0), point(20, 21, 0)},  // Triangle 25
    {point(21, 20, 0), point(21, 21, 0), point(20, 21, 0)},  // Triangle 26
    {point(22, 22, 0), point(23, 22, 0), point(22, 23, 0)},  // Triangle 27
    {point(23, 22, 0), point(23, 23, 0), point(22, 23, 0)},  // Triangle 28
    {point(24, 24, 0), point(25, 24, 0), point(24, 25, 0)},  // Triangle 29
    {point(25, 24, 0), point(25, 25, 0), point(24, 25, 0)}   // Triangle 30
};
```

Figure 31: Cube Rendered at Low Resolution

– **Detail Comparison:** Highlights the differences in quality between low, medium, and high-resolution renders.

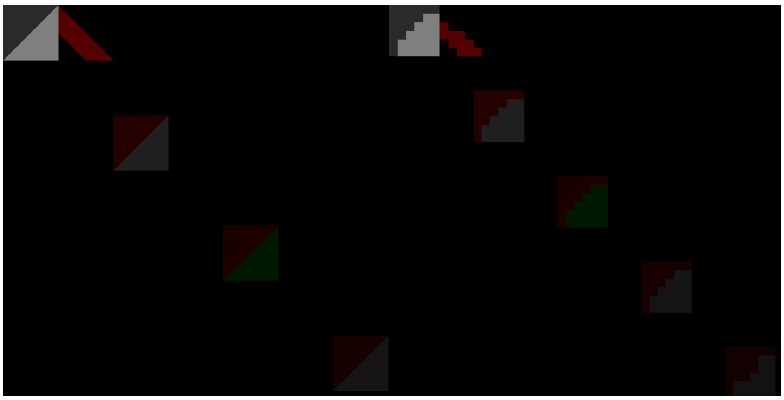

Figure 32: Comparison of Cube Renders at Different Resolutions

– **High Resolution:** Captures detailed edges and smooth surfaces, showcasing the renderer's precision. Increasing the resolution size and decreasing the step intervals produces higher detail and denser pixel projections.

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

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

Figure 33: Low-resolution cube render highlighting basic structure.

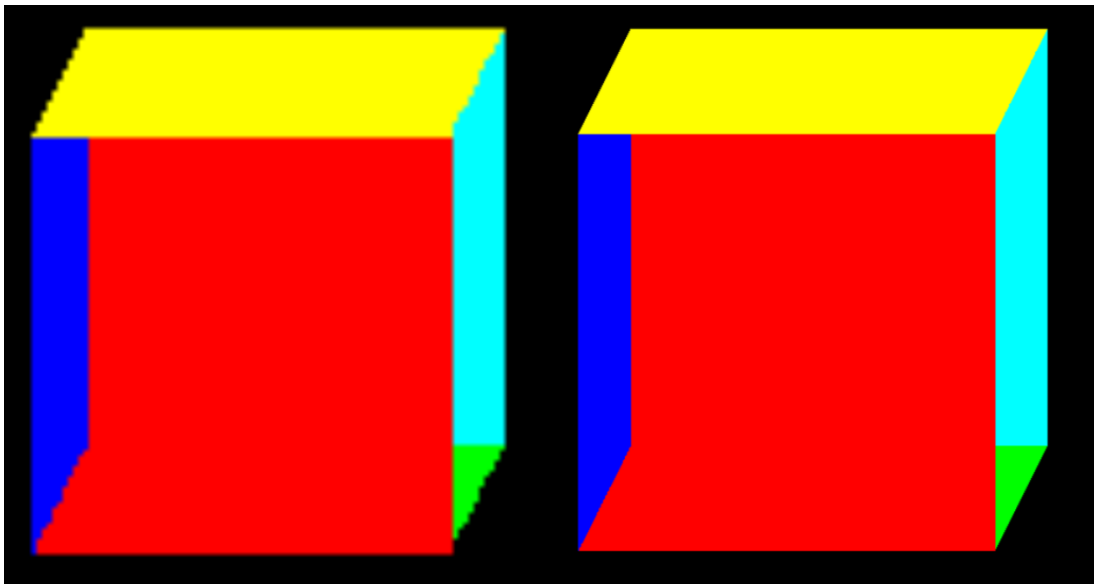

Figure 34: High-resolution cube render with enhanced detail and smooth surfaces.

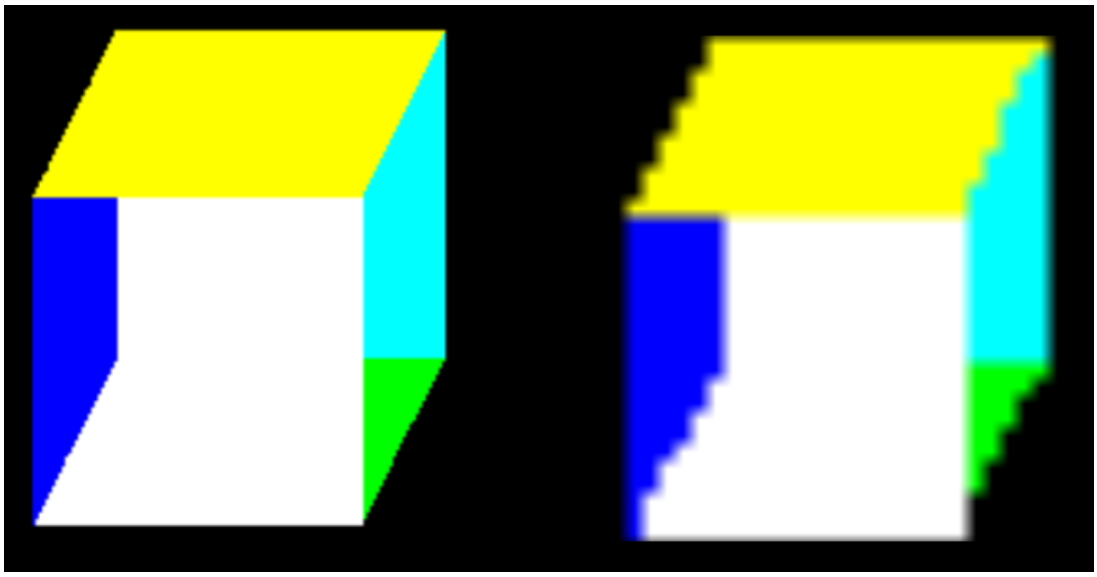

Figure 35: Comparison of renders showcasing detail improvements across resolutions.

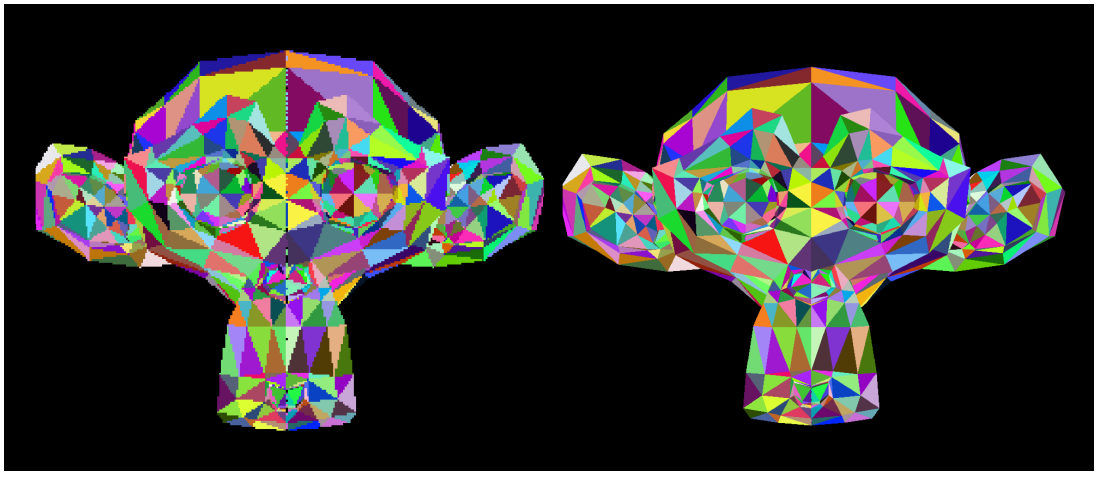

Figure 36: Comparison of renders showcasing detail improvements across resolutions.

3.2 Complex Models

The renderer's ability to handle intricate geometry is demonstrated through renders of complex models, such as a Dahlia flower and the Suzanne model.

– **Dahlia Flower:** The render captures the delicate geometry of the Dahlia flower, highlighting the renderer's ability to process fine details.

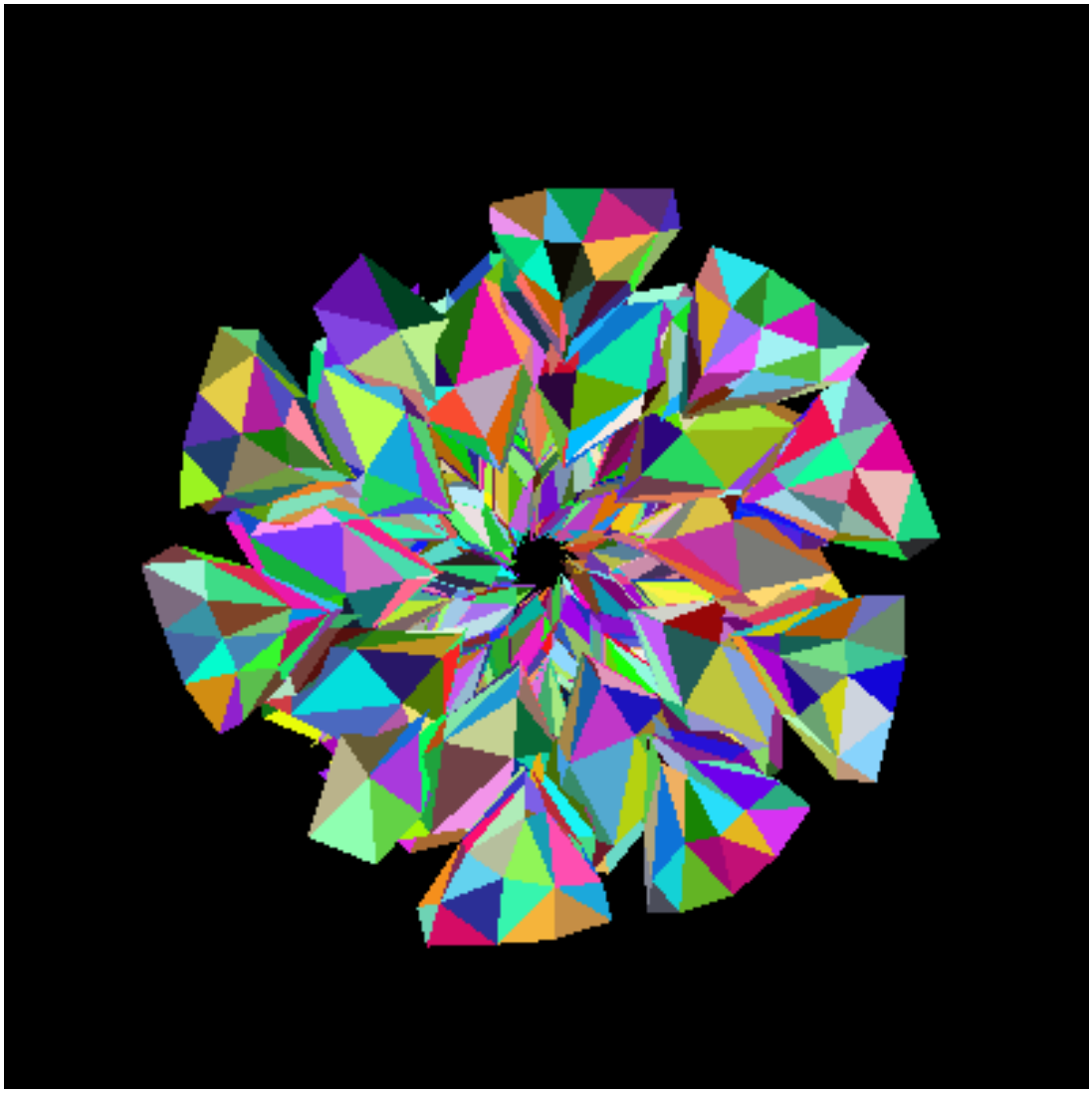

Figure 37: Rendered Dahlia Flower with Fine Detail

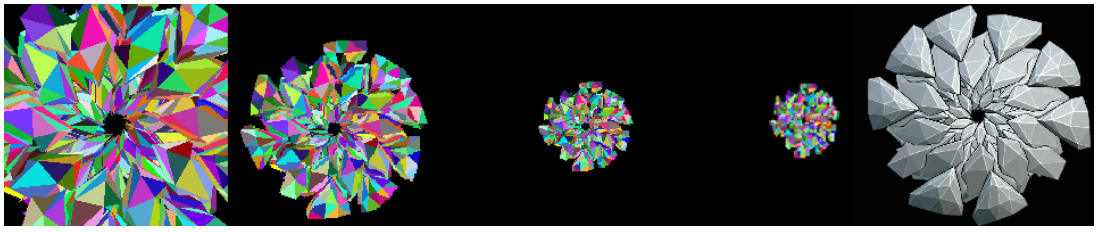

Figure 38: Rendered Dahlia Flower with Fine Detail full step

– **Suzanne Model:** Suzanne, a standard rendering benchmark, showcases the renderer's capability to handle complex shapes and shading.

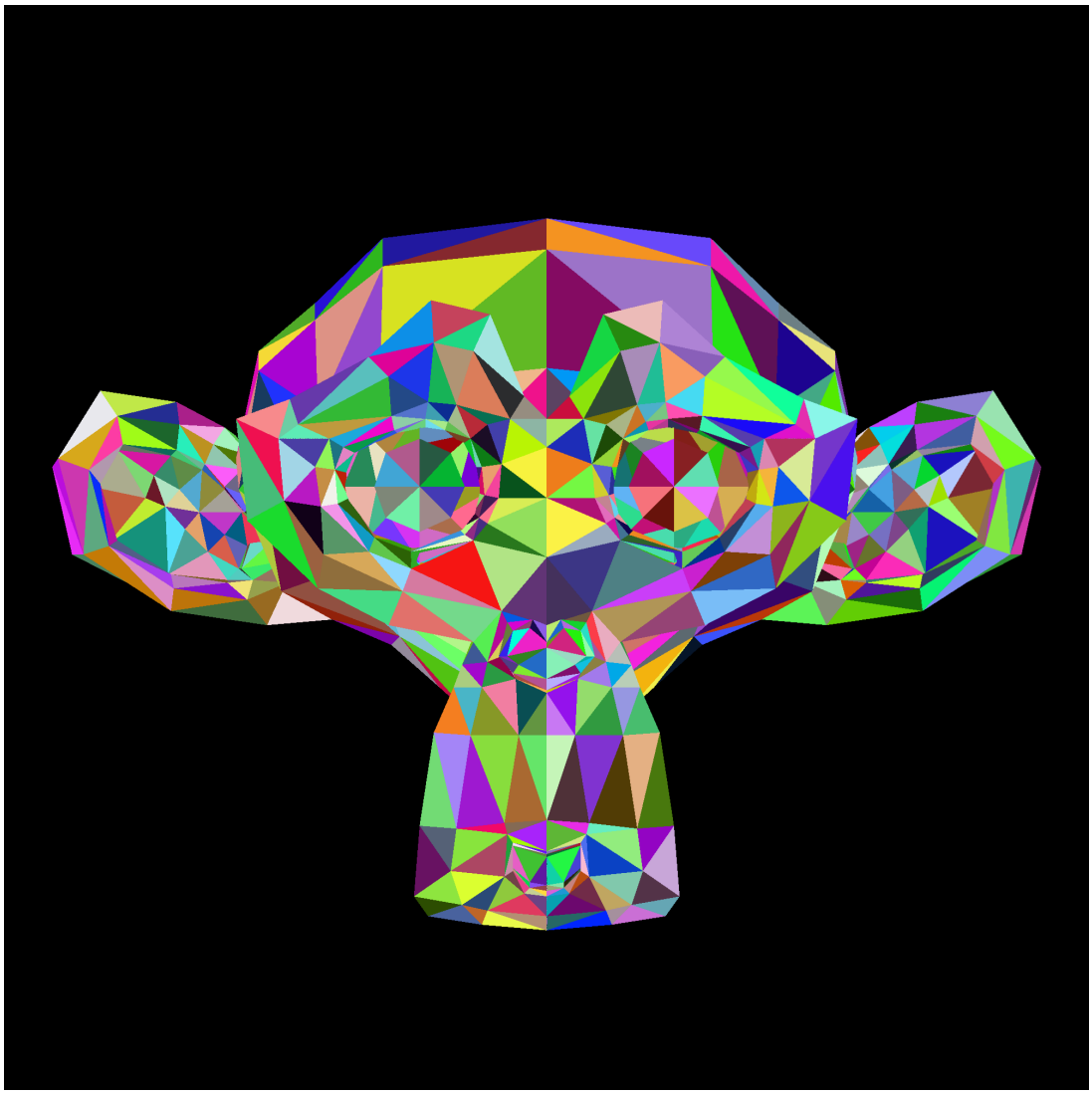

Figure 39: Rendered Suzanne Model with Complex Geometry

– **Wireframe Visualization:** A wireframe render of the Suzanne model shows how triangles form the underlying structure.

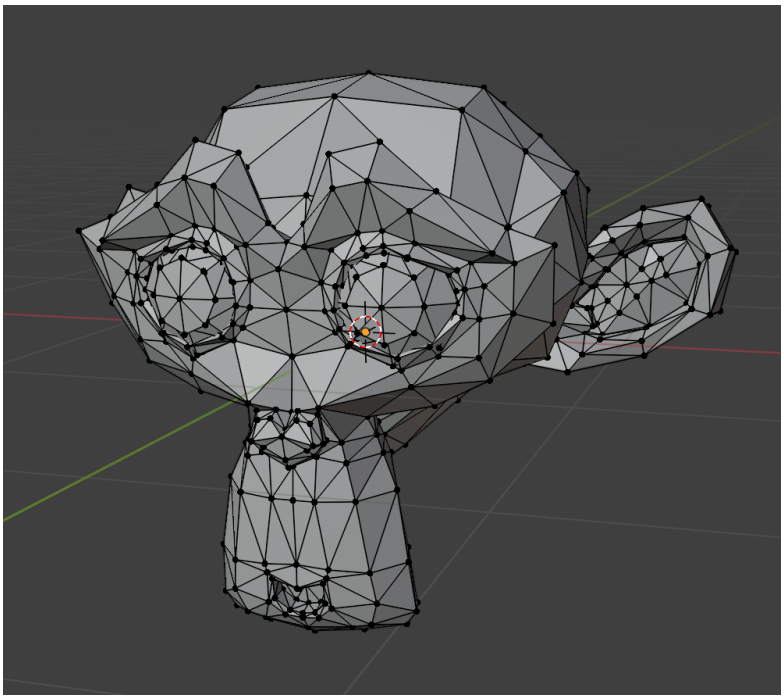

Figure 40: Wireframe View of Suzanne Model Showing Triangular Structure

—

3.3 Rendering Errors and Fixes

This section highlights common rendering issues, such as incorrect order or logic errors, and their resolution.

– **Rendering Error:** An example of a rendering error caused by incorrect logic, resulting in misplaced geometry or colors.

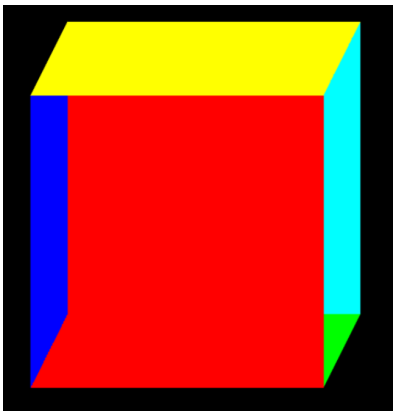

Figure 41: Rendering Error Due to Incorrect Logic

– **Fixed Rendering:** After resolving the error, the renderer produces accurate and visually correct results.

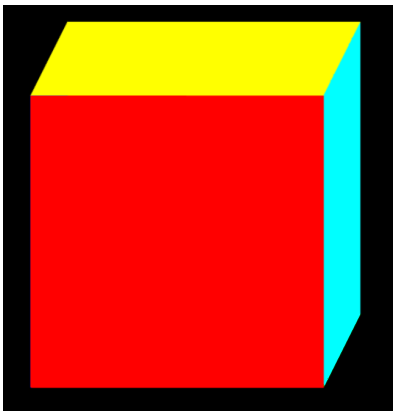

Figure 42: Corrected Rendering After Fixing Logic Errors

—

3.4 Low Ray Density Impact

The density of rays affects the detail and accuracy of the rendered image. Low ray density can result in missing or distorted details, as shown below.

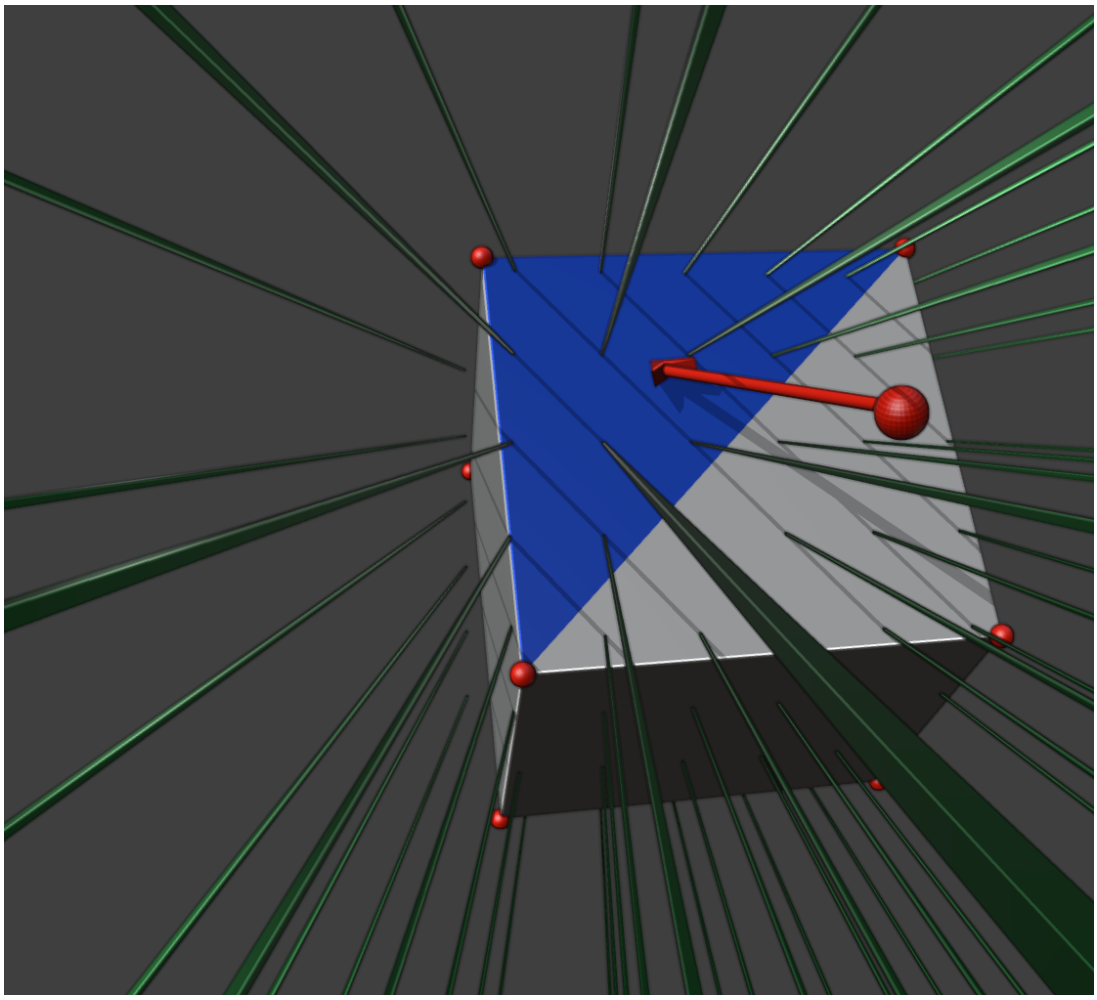

Figure 43: Effect of Low Ray Density on Detail Accuracy

Importance of Ray Density:

- Ensures accurate representation of fine details.
- Reduces artifacts caused by insufficient sampling.
- Improves overall rendering quality and realism.

—

4 Supporting Visualizations and Code Representations

4.1 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 44: Image Class Code Representation

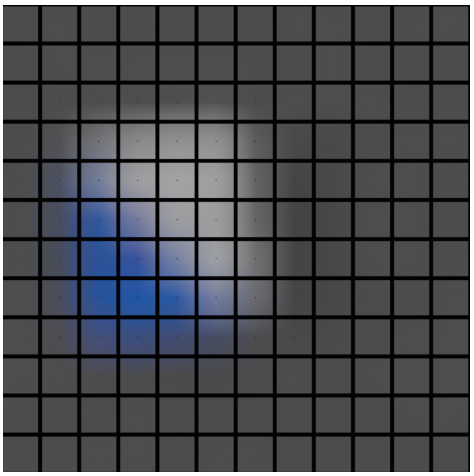

Figure 45: Detailed Code Representation of Image Class

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

—

4.2 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

```
1   (0.468750,-0.757812,0.242188);(0.500000,-0.687500,0.093750);(0.562500,-0.671875,0.242188)
2   (-0.500000,-0.687500,0.093750);(-0.468750,-0.757812,0.242188);(-0.562500,-0.671875,0.242188)
3   (0.562500,-0.671875,0.242188);(0.546875,-0.578125,0.054688);(0.625000,-0.562500,0.242188)
4   (-0.546875,-0.578125,0.054688);(-0.562500,-0.671875,0.242188);(-0.625000,-0.562500,0.242188)
5   (0.500000,-0.687500,0.093750);(0.351562,-0.617188,-0.023438);(0.546875,-0.578125,0.054688)
6   (-0.351562,-0.617188,-0.023438);(-0.500000,-0.687500,0.093750);(-0.546875,-0.578125,0.054688)
7   (0.437500,-0.765625,0.164062);(0.351562,-0.718750,0.031250);(0.500000,-0.687500,0.093750)
8   (-0.351562,-0.718750,0.031250);(-0.437500,-0.765625,0.164062);(-0.500000,-0.687500,0.093750)
9   (0.351562,-0.781250,0.132812);(0.203125,-0.742188,0.093750);(0.351562,-0.718750,0.031250)
10  (-0.203125,-0.742188,0.093750);(-0.351562,-0.781250,0.132812);(-0.351562,-0.718750,0.031250)
11  (0.351562,-0.718750,0.031250);(0.156250,-0.648438,0.054688);(0.351562,-0.617188,-0.023438)
12  (-0.156250,-0.648438,0.054688);(-0.351562,-0.718750,0.031250);(-0.351562,-0.617188,-0.023438)
13  (0.140625,-0.742188,0.242188);(0.156250,-0.648438,0.054688);(0.203125,-0.742188,0.093750)
14  (-0.140625,-0.742188,0.242188);(-0.156250,-0.648438,0.054688);(-0.078125,-0.656250,0.242188)
15  (0.273438,-0.796875,0.164062);(0.140625,-0.742188,0.242188);(0.203125,-0.742188,0.093750)
16  (-0.140625,-0.742188,0.242188);(-0.273438,-0.796875,0.164062);(-0.203125,-0.742188,0.093750)
17  (0.242188,-0.796875,0.242188);(0.203125,-0.742188,0.390625);(0.140625,-0.742188,0.242188)
18  (-0.203125,-0.742188,0.390625);(-0.242188,-0.796875,0.242188);(-0.140625,-0.742188,0.242188)
19  (0.203125,-0.742188,0.390625);(0.078125,-0.656250,0.242188);(0.140625,-0.742188,0.242188)
20  (-0.203125,-0.742188,0.390625);(-0.078125,-0.656250,0.242188);(-0.156250,-0.648438,0.437500)
21  (0.351562,-0.718750,0.453125);(0.156250,-0.648438,0.437500);(0.203125,-0.742188,0.390625)
22  (-0.351562,-0.718750,0.453125);(-0.156250,-0.648438,0.437500);(-0.351562,-0.617188,0.515625)
```

Figure 46: Example of a Supported 3D File Format

```
1   /**
2    * @file MeshReader.h
3    * @brief Defines the MeshReader class for reading text files to extract mesh data.
4    */
5
6   #ifndef MESH_READER_H
7   #define MESH_READER_H
8
9   #include <string>
10  #include <vector>
11  #include <fstream>
12
13  #include "point.h"
14
15  using namespace std;
16  /**
17   * @class MeshReader
18   * @brief Represents a mesh reader for reading text files to extract mesh data.
19   * The MeshReader reads a text file that has lines each line contains a set of vertices that
20   * represent a face each vector position is seperated by a semicolone and each coordinate is seperated by a comma.
21   */
22  class MeshReader {
23  public:
24      vector<vector<string>> verticesString;
25      MeshReader(string filename)
```

Figure 47: 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.

—

5 Splitting a Camera into Sub-Cameras

To achieve high-resolution rendering or wide-angle coverage, a single virtual camera can be divided into multiple sub-cameras. These sub-cameras work collaboratively, each rendering a portion of the scene. The outputs from these sub-cameras are then stitched together to form the final image that represents the original camera's view.

5.1 Overview of Sub-Camera Splitting

The splitting process divides the original camera's image plane into smaller regions, assigning each region to a sub-camera. Each sub-camera:

- Shares the same focal length and resolution properties as the original camera.
- Adjusts its field of view to cover its specific region of the image plane.
- Produces a partial render that contributes to the final stitched image.

5.2 Splitting and Stitching Process

- **Splitting the Image Plane:** The image plane of the original camera is divided into a grid, where each grid cell corresponds to the view of a sub-camera.

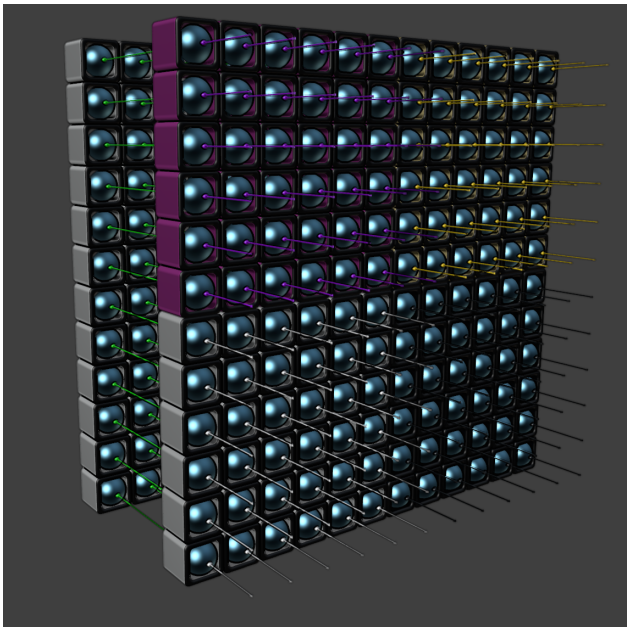

Figure 48: Division of the Camera's Image Plane into Sub-Cameras

- **Rendering with Sub-Cameras:** Each sub-camera independently renders its assigned region, using the same ray-casting logic as the original camera but restricted to its sub-image.

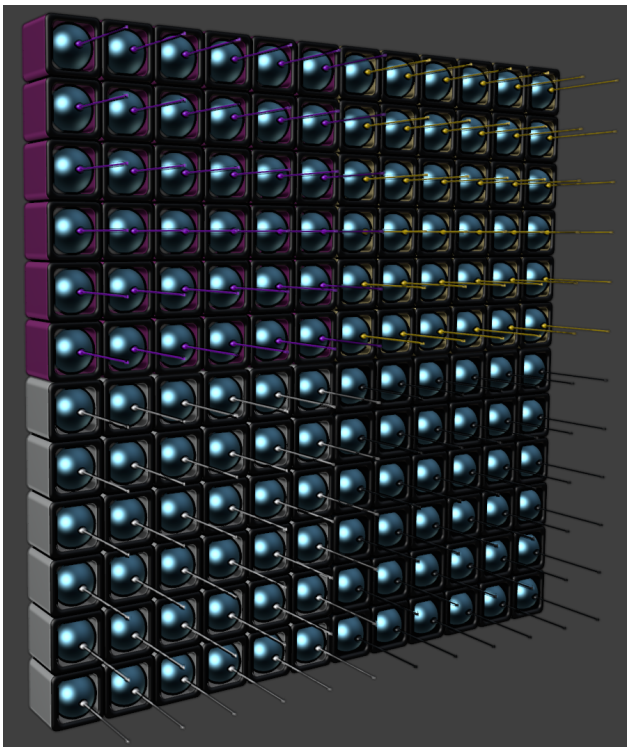

Figure 49: Rendering Outputs from Sub-Cameras

– **Visualization of the Sub-Cameras:** Each sub-camera occupies a defined space within the main camera's view, which is disabled for direct rendering. The sub-camera is responsible for handling its designated segment, ensuring efficient workload distribution and detailed rendering outputs.

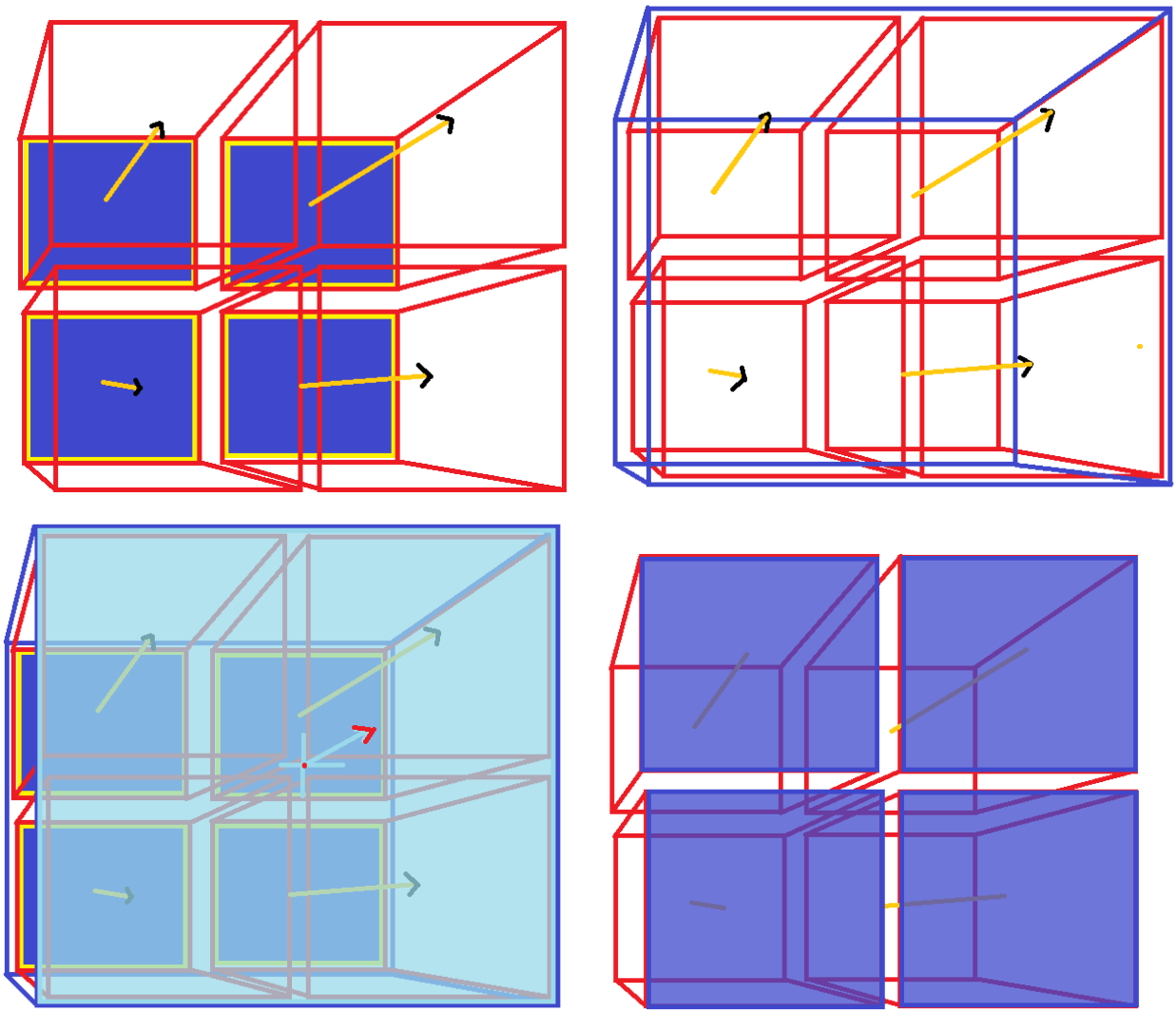

Figure 50: Examples of sub-camera layouts and their representation.

– **Stitching the Outputs:** The rendered outputs from the sub-cameras are stitched together by aligning their edges seamlessly. This process recreates the full image that the original camera would have produced.

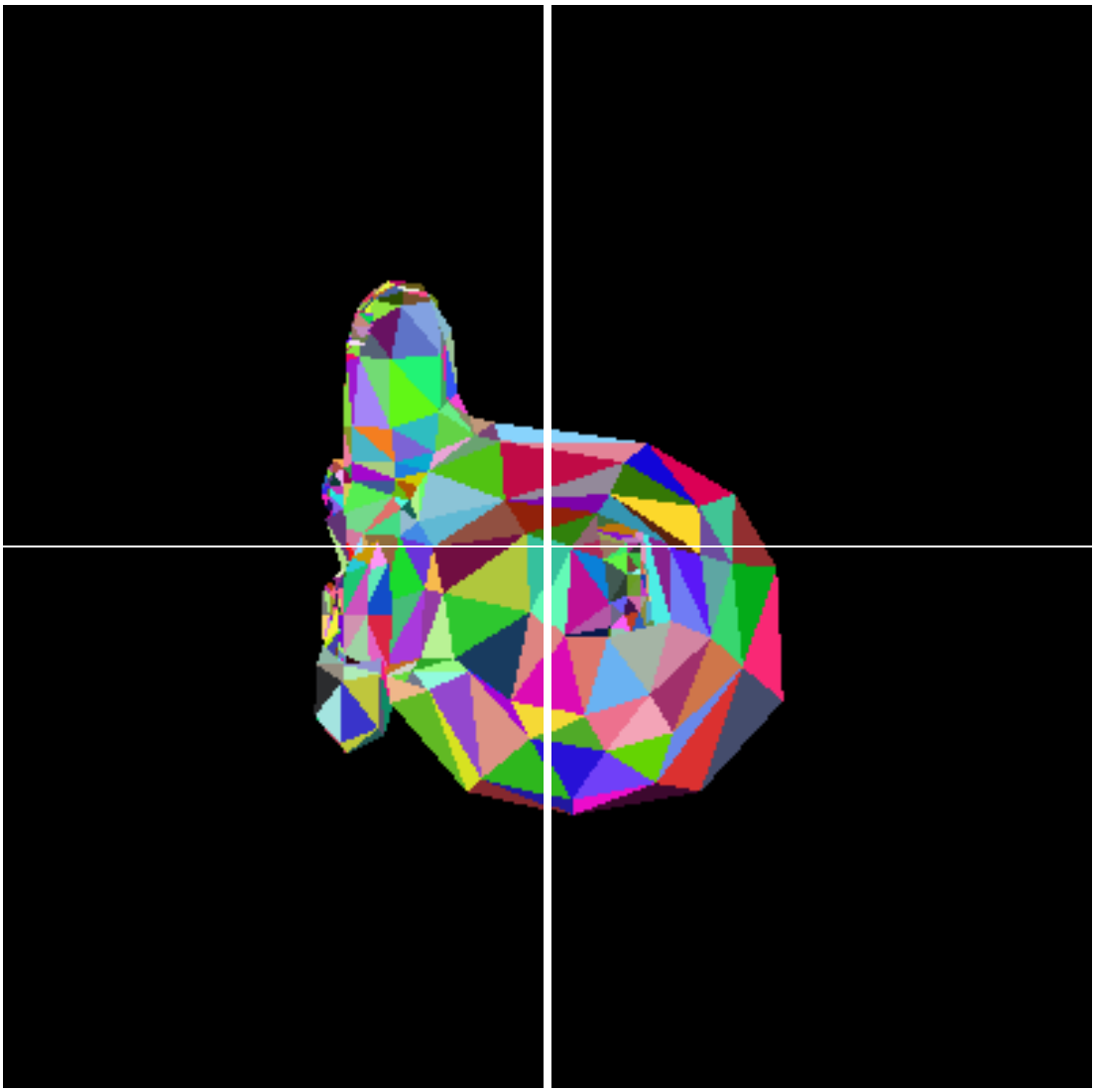

Figure 51: Fragments Produced by Sub-Cameras

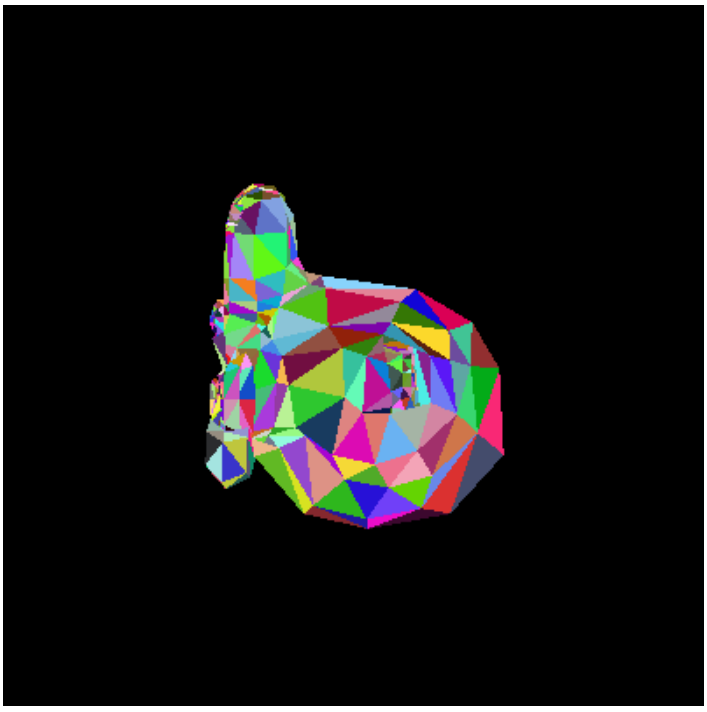

Figure 52: Final Image Reconstructed from Sub-Camera Fragments

5.3 Advantages of Sub-Camera Splitting

- **Increased Resolution:** Splitting allows rendering at higher resolutions by focusing computational resources on smaller sections of the image plane.
- **Efficient Parallel Processing:** Sub-cameras can be rendered in parallel, reducing overall rendering time.
- **Enhanced Coverage:** Splitting can handle larger fields of view by distributing the rendering workload among sub-cameras.

5.4 Applications of Sub-Camera Splitting

- **High-Resolution Renders:** Used in scenarios requiring detailed images, such as architectural visualization or medical imaging.
- **Panoramic and Wide-Angle Views:** Enables rendering of panoramic scenes or wide-angle views by splitting the camera into regions.
- **Efficient Load Distribution:** Distributes rendering workloads in systems with limited computational resources.

5.5 Challenges in Sub-Camera Splitting

- **Seamless Stitching:** Ensuring that the edges of sub-camera renders align perfectly without visible seams.
- **Calibration:** Correctly calibrating the field of view and orientation of each sub-camera to avoid distortions.
- **Performance Overhead:** Managing additional computational overhead for rendering and stitching.
- **Threading and Async Behavior:** Introducing threading using asynchronous programming (`async`) significantly improves performance by parallelizing sub-camera rendering. This approach optimizes resource utilization and reduces rendering time.

5.6 Visualization of Threading and Async Implementation

As shown in Figure 51, the sub-camera splitting process utilizes asynchronous behavior to produce individual fragments. By leveraging async programming, these fragments are rendered in parallel, significantly reducing processing time. The approach ensures efficient load distribution, seamless integration, and the generation of high-resolution outputs.

To better understand the implementation of threading and async behavior in sub-camera splitting, the following images are provided:

- **Calling Threads:** Refer to Figure 53.
- **Function Using Async Behavior:** Refer to Figure 54.

```
// Declare a vector to store futures for asynchronous tasks
std::vector<std::future<void>> futures;
// call the function containing the async behavior for now we are calling on async for each object within each camera
// for our case we only have 1 object and 4 cameras so its 1 thread but we should change it to 1 thread per camera it makes more
// TO-DO XXX : switch thread logic
s.threadedCameraRay(futures);
```

Figure 53: Illustration of threading in sub-camera splitting.

```
97
98  void threadedCameraRay(std::vector<std::future<void>>& futures) {
99      for (auto& cam : cameras) {
100         size_t camIndex = &cam - &cameras[0];
101         std::cout << "Thread N*= " << camIndex << " |started!" << std::endl;
102         for (auto& o : obj) {
103             // Launch asynchronous task for processing each camera-object pair
104             futures.push_back(std::async(std::launch::async, [&cam, &o, &camIndex]() {
105                 // Calculate thread identifier for logging
106
107                 // Log thread start
108
109                 // Clear screen (optional, might not work in all consoles)
110                 std::cout << "\033[2J\033[H";
111
112                 // Process the object with the camera
113                 cam.cameraToImage(o);
114
115                 // Save the rendered image to a file
116                 ImageRenderer::renderToFile(cam.getimage(), "output" + std::to_string(camIndex) + ".ppm");
117
118                 // Log thread end
119             }));
120         }
121         std::cout << "Thread N*= " << camIndex << " |ended!" << std::endl;
122     }
123 }
```

Figure 54: Demonstration of the function using async behavior for rendering.

6 Texture Rendering from a Mesh

Texture rendering from a mesh involves mapping a 2D image (texture) onto a 3D object. This process is fundamental for adding details, such as patterns or colors, to 3D models.

6.1 Original Texture

The base texture serves as a reference for all mapping and blending. This texture is the starting point for applying details to the 3D model.

Figure 55: Original Leopard Skin Texture. Click the image to view the source.

6.2 Focused Area and Cropped Sections

The texture is applied precisely to sections of the mesh, often requiring focus on specific areas to ensure accuracy.

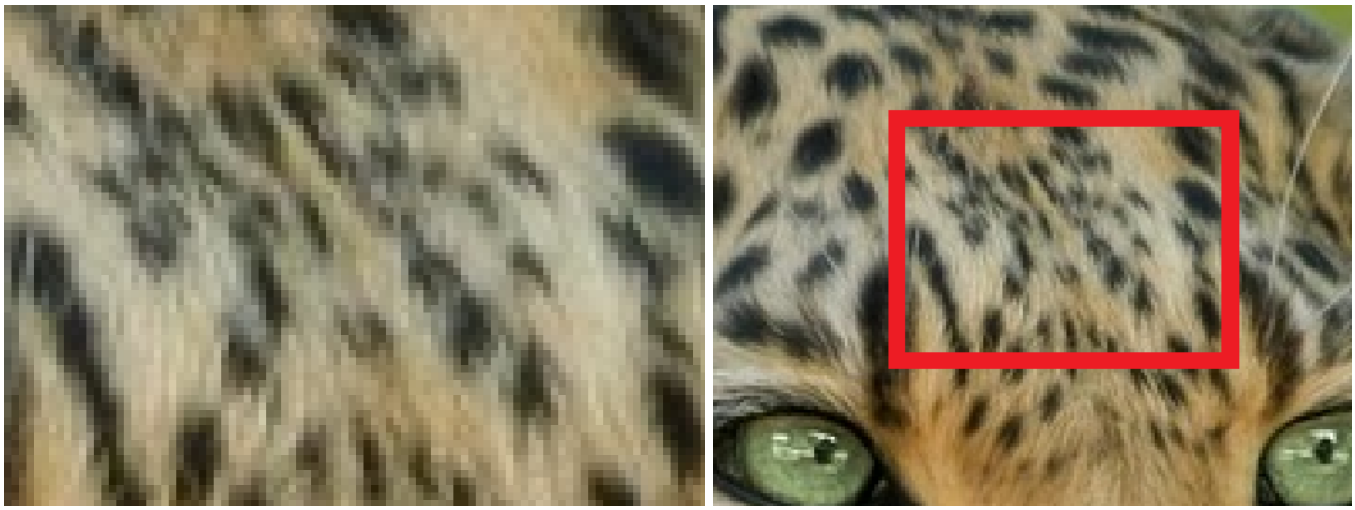

Figure 56: Focused Area and Cropped Section of the Leopard Texture

6.3 Plane Projection

Texture rendering begins with projecting a 2D texture onto a triangular surface in the 3D space. Each triangle in the mesh is associated with a specific portion of the texture.

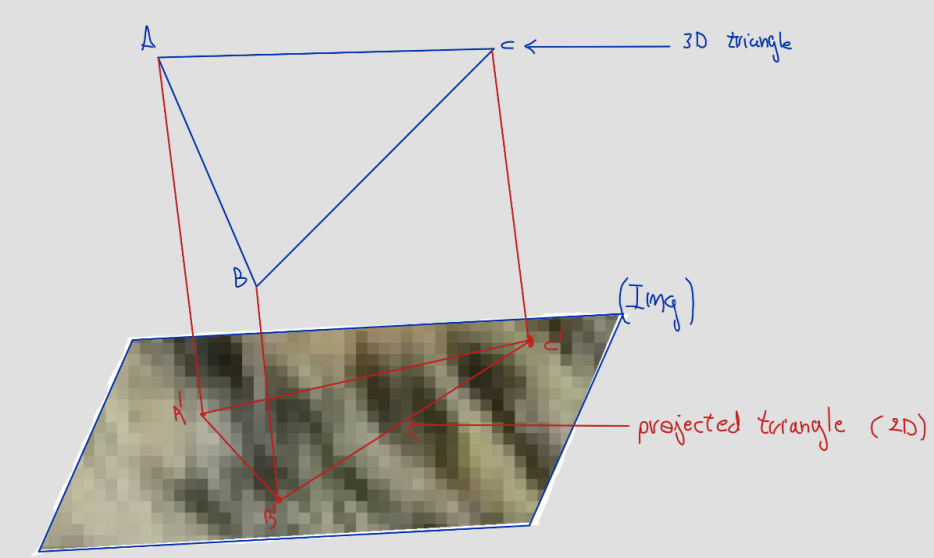

Figure 57: Plane Projection of a Triangle into a Leopard Skin Texture

6.4 Texture Blending with Vertices

Textures are blended with the vertices of the 3D object at various levels of detail, ensuring a smooth distribution across the mesh surface.

Figure 58: Final Textured Cube with Casted Texture and Vertex Colors

6.5 Final Textured Output

Once the texture is applied and blended, the final textured mesh is rendered. The texture adds realism and detail to the 3D model.

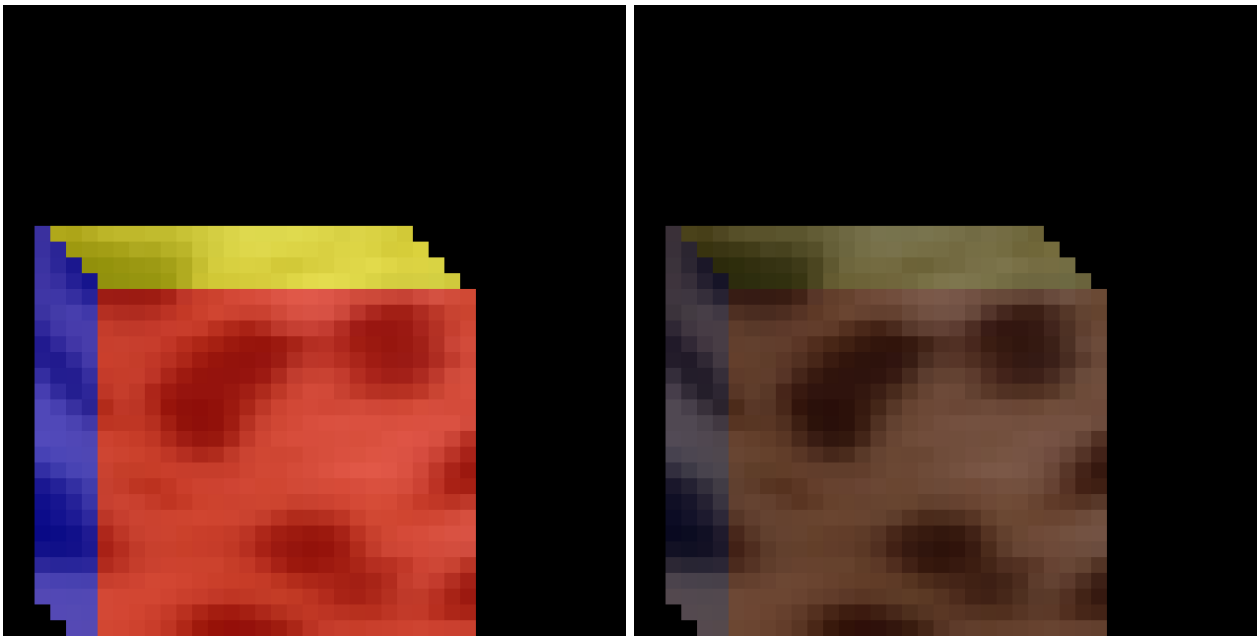

Figure 59: Blending Textures with Vertices at Different Levels

7 Built In Primitives

The project includes a set of built-in 3D primitives that are generated using hard-coded parameters. These primitives serve as fundamental building blocks for constructing complex 3D scenes. By defining fixed values for properties such as vertex positions, normals, and texture coordinates, these shapes can be quickly instantiated and rendered without requiring external models or additional geometry processing.

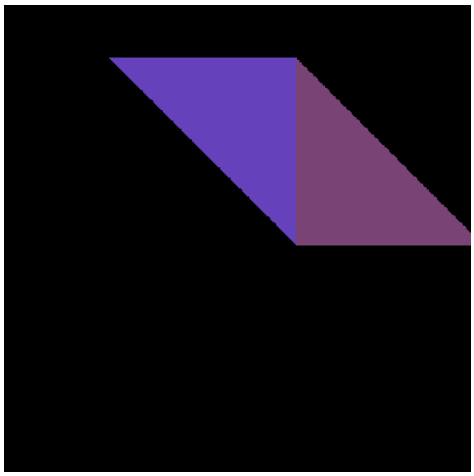

Figure 60: Plane (1)

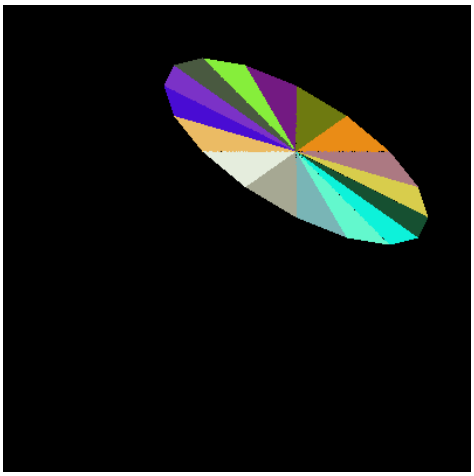

Figure 61: Circle (2)

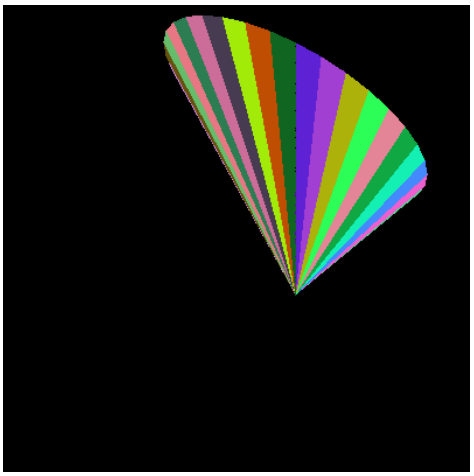

Figure 62: Cone (3)

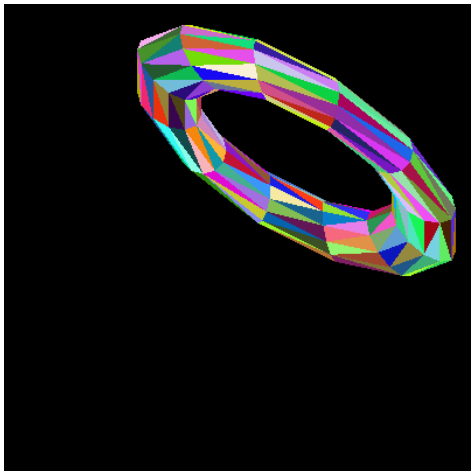

Figure 63: Torus (4)

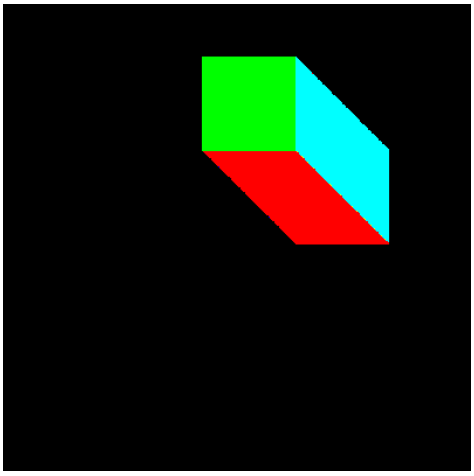

Figure 64: Cube (5)

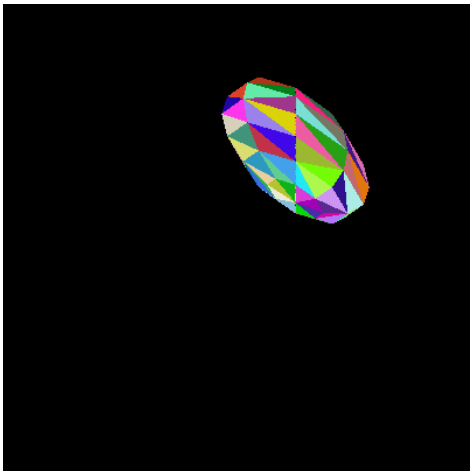

Figure 65: Sphere (6)

8 Video Generation with FFmpeg

- Camera moving in a space with a cube
- Camera moving in a space with a cone

The project includes built-in functionality to generate videos from a folder of PPM image frames using FFmpeg. This allows for a seamless transformation of sequentially numbered images into a high-quality video file.

8.1 Animation Parameters and Object Transformations

The system supports parameter-driven animations that can be used to control:

- Rotation: Objects can rotate dynamically based on predefined or computed values.
- Translation: movement through space can be defined for a single object or multiple entities.
- Scaling: Objects can change in size over time to simulate dynamic transformations.
- Custom Metrics: Any other transformation, procedural motion, or simulation-based effects, can be encoded into the frame sequence.

Each of these parameters can be applied to asingle object or multiple objects ssimultaneously, allowingex animations that simulate customcustom 3D environments.ctionNon-Interference with Multi-Threading and Image Stitching

This animation system operates independently of multithreading and image stitching logic , ensuring efficient rendering:

- Parallel Processing Compatibility: Multi-threaded rendering can continue without interruption, as each frame is generated independently.
- Frame Stitching Consistency: Sequential PPM images are composed correctly into a video, regardless of transformations applied per frame.
- Scalability for custom simulations: Different motion and animation parameters can be defined per object without disrupting the overall rendering process.

8.2 Generating Video from Image Sequences

A standard FFmpeg command for this process is:

```
ffmpeg -framerate 30 -i frame_%04d.ppm -c:v libx264 -pix_fmt yuv420p output.mp4
```

This command takes a series of PPM images (e.g., `frame_0001.ppm`, `frame_0002.ppm`, ...) and encodes them into an MP4 file with a framerate of 30 FPS, using the H.264 codec for efficient compression. The resulting video is compatible with most media players and web platforms.

8.3 Custom Video Simulations

Since any transformation can be controlled per frame, the system allows for the creation of custom video simulations of dynamically evolving 3D spaces. This enables:

- Simulating Virtual Environments where multiple objects move independently.
- Complex Camera Movements that create immersive video outputs.

By integrating FFmpeg for video processing, multi-threaded rendering , and parameter-driven transformations , this system enables highly flexible and efficient video generation workflows.

9 Perspective Rendering in Multi-Camera Systems

Perspective rendering plays a crucial role in generating realistic 3D projections. In this system, perspective is achieved using a dual-camera setup, where the distance between each ray is controlled, centered, and then shifted to align properly in the scene. The rays are projected from the first camera to the second, ensuring a more accurate depth perception.

10 Perspective Rendering in Multi-Camera Systems

Perspective rendering plays a crucial role in generating realistic 3D projections. In this system, perspective is achieved using a grid of rays instead of a second camera. The distance between each ray is controlled, centered, and then shifted to align properly in the scene. The rays are recalculated based on this grid, ensuring a more accurate depth perception.

10.1 Perspective Scale Variations

The following images illustrate different perspective scales applied in the rendering process:

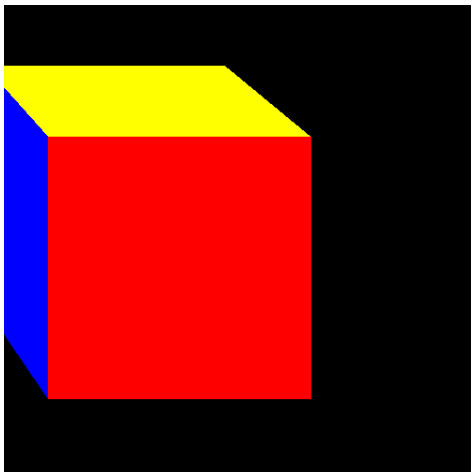

Figure 66: Perspective Scale 0 which is an orthographic projection.

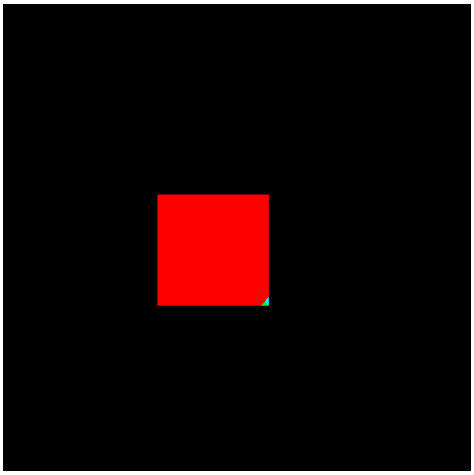

Figure 67: Perspective Scale 16

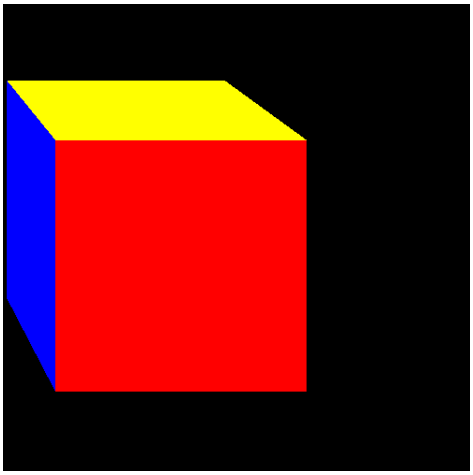

Figure 68: Perspective Scale 1.2

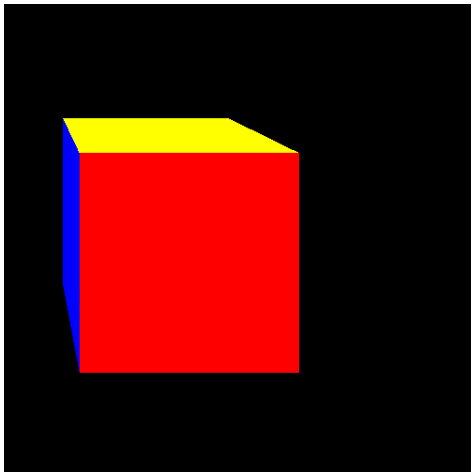

Figure 69: Perspective Scale 2.9

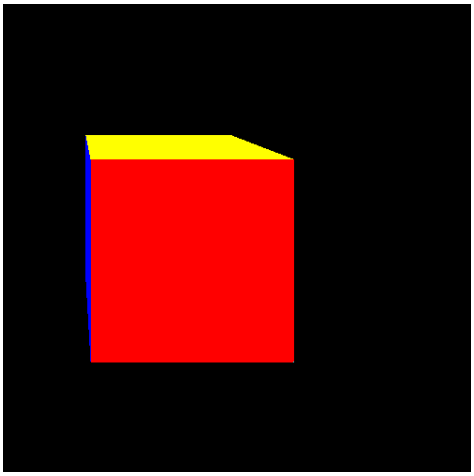

Figure 70: Perspective Scale 3.99

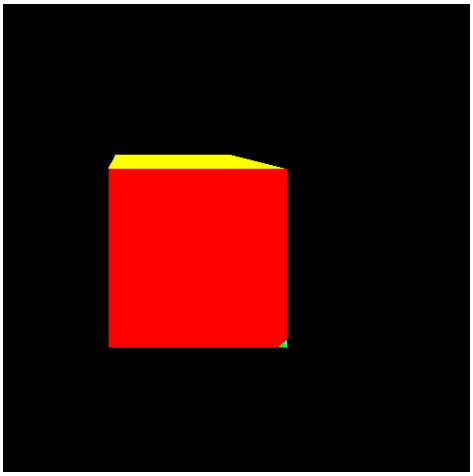

Figure 71: Perspective Scale 6

10.2 Grid-Based Perspective Calculation

The system uses a grid of rays to achieve perspective rendering:

- Ray Distance Control: The distance between each projected ray is adjusted to align properly.
- Centered Projection: Initial projection is calculated from a centered position to ensure correct depth perception.
- Scene Alignment: The rays are then shifted in the scene to match the required perspective.
- Projection Process: The rays are recalculated based on a grid structure, simulating depth realism in 3D space.

By dynamically adjusting the perspective scale and using a structured grid approach, the system ensures a highly customizable and realistic rendering method.

10.3 Applications of Perspective Rendering

The ability to control and modify perspective dynamically enables various applications:

- Simulated Camera Views: Generate realistic camera perspectives for animations and simulations.
- Depth-Based Effects: Implement perspective scaling for realistic 3D transformations.
- Enhanced Motion Rendering: Utilize different perspective scales to create immersive motion effects.

By leveraging grid-based ray projection and customizable scaling, the system enables high-fidelity 3D rendering, ensuring accurate and visually appealing results.