Ray Tracing with Rotation and Object Composition for Scene Rendering

Marina Lorraine Rodriguez
University of Puerto Rico - Mayaguez

Abstract
This paper explores the implementation of a ray tracing program designed to render a 3D scene of a ceramic unicorn figurine. The program employs mathematical principles, including vector calculus, matrix transformations, and geometric intersection algorithms, to simulate light interactions with various objects such as spheres and cones. Techniques such as ambient, diffuse, and specular lighting are utilized to achieve realistic shading effects. The rendering process incorporates recursion for reflective and refractive materials, resulting in a vivid and dynamic scene. This paper thoroughly analyzes the methods used, presenting key mathematical formulas, computational strategies, and results of the rendering process.

Contents

1 Key Words

Ray tracing, scene rendering, 3D graphics, lighting models, vector calculus, computational geometry, recursive algorithms.

2 CCS Concepts

- **Computing methodologies** → Ray tracing
- **Mathematics of computing** → Vector algebra
- **Graphics systems and interfaces** → Rendering techniques

Final rendering of scene.

3 Introduction

Ray tracing is a foundational technique in computer graphics, enabling the simulation of realistic lighting by modeling light rays' interactions with virtual objects. The program detailed here renders a stylized ceramic unicorn figurine scene using spheres and cones to model the unicorn's body and environment.

This rendering employs core principles of computational geometry, vector algebra, and lighting physics. It uses ray-object intersection testing and recursive reflections/refractions to achieve lifelike effects. A significant feature is the rotation of objects within the scene, which applies matrix transformations to simulate motion and perspective.

The primary purpose of this code is to demonstrate the principles and practical implementation of ray tracing, a fundamental technique in computer graphics for simulating realistic lighting and shading. By leveraging mathematical models such as vector calculus, geometric transformations, and recursive algorithms, the program highlights how light interacts with 3D objects in a virtual scene. The code showcases the power of combining computational efficiency with visual creativity to produce a dynamic figurine. This serves as both an educational tool for understanding ray tracing fundamentals and a proof of concept for how advanced graphics techniques can generate visually compelling and mathematically robust images.

4 Set Up

To execute the ray tracing program and render the unicorn scene, follow these steps:

4.1 Required Tools

- **Hardware:** A computer with a modern processor capable of handling computationally intensive tasks is recommended. For smooth performance, at least 4 GB of RAM is advisable.
- **Operating System:** Any operating system with support for a C++ compiler, such as Windows, Linux, or macOS.
- **Compiler:** A C++17-compatible compiler is required to build and run the code. Common options include GCC (GNU Compiler Collection), Clang, or Microsoft Visual Studio.
- **Text Editor or IDE:** Any code editor, such as Visual Studio Code, Sublime Text, or an Integrated Development Environment (IDE) like CLion or Code::Blocks, can be used to edit and compile the code.

4.2 Instructions for Execution

1. Clone or download the source code file, ensuring all dependencies (if any) are included.
2. Open the source code file in your chosen text editor or IDE.
3. Compile the program using your preferred compiler. For example, with Visual Studio Code, the command would be:

   ```
   g++ RaytracingUnicorn.cpp
   ```

4. Run the compiled executable. For example:

   ```
   ./a.exe
   ```

   This will render the scene and save the output to a PPM file named `Unicorn.ppm`.

5. Open the PPM file using an image viewer that supports PPM format. This can then be saved to device as PNG or copied to clipboard.

4.3 Code Customization

Users can modify the scene by editing the `setup_scene()` function in the code. Adjusting parameters such as object positions, colors, lighting intensities, and the `frame` variable allows for dynamic customization of the rendered scene. The computational workload of the program depends on the recursion depth, the number of objects, and the resolution of the image (`Cw` and `Ch`). To improve performance on less powerful machines, consider reducing the resolution or the recursion depth in the `trace_ray()` function. By following these steps and recommendations, users can successfully run the program and render high-quality ray-traced images.[4]

5 Methods

5.1 Vector Operations and Transformations

The `Vector3D` class encapsulates operations essential for 3D geometry, which are fundamental for rendering and geometric transformations.[2] These operations ensure accurate movement, scaling, and interaction of 3D objects within the scene:

- **Length Calculation**: Computes the vector's magnitude:

$$\|V\| = \sqrt{x^2 + y^2 + z^2},$$

  This step is necessary to determine the scale of a vector, which will be used in various transformations, such as normalization and projection.

- **Normalization (General)**: Produces a unit vector:

$$V_{\text{norm}} = \frac{1}{\|V\|}(x, y, z),$$

  Normalization converts any vector to a unit vector with a magnitude of 1, which is essential for calculations such as direction vectors and normalizing light sources.

- **Sphere Normalization**: For spheres, the normal vector at a given point is obtained by subtracting the sphere's center (C) from the point (P) and normalizing the result:

$$N_{\text{sphere}} = \frac{1}{\|P - C\|}(P - C).$$

  This normalization is specific to the surface of the sphere and ensures that the normal is always perpendicular to the sphere at the given point.

- **Cone Normalization**: For cones, the normal vector involves a more complex calculation that incorporates the cone's axis vector (V), apex (A), and the point of interest (P):

$$N_{\text{cone}} = \frac{1}{\|N\|}N,$$

  where:

$$N = P - A - (V \cdot (P - A))\, V.$$

  This formula accounts for the orientation and slope of the cone, ensuring the normal vector is perpendicular to the cone's surface at any point.[3]

- **Dot Product**: Projects one vector onto another:

$$A \cdot B = x_1x_2 + y_1y_2 + z_1z_2,$$

  The dot product is crucial for computing the angle between vectors and is heavily used in lighting calculations like diffuse shading, where the angle between the surface normal and the light direction determines brightness.

- **Cross Product**: Generates a vector orthogonal to two given vectors:

  $$A \times B = (y_1 z_2 - z_1 y_2, z_1 x_2 - x_1 z_2, x_1 y_2 - y_1 x_2),$$

  The cross product provides a vector perpendicular to the plane defined by two vectors, which is critical for calculating surface normals and implementing transformations such as rotations.

Rotation Transformation: The rotation mechanism in the code uses a rotation matrix to transform the coordinates of objects around the Y-axis in 3D space. This enables objects, such as the unicorn's components, to dynamically change position, simulating motion or perspective shifts. The rotation is determined by the `frame` variable, which represents an angular fraction of a full 360-degree rotation. The angle of rotation is calculated in radians using the formula:

$$\text{angle (radians)} = \frac{\text{frame}}{360} \cdot 2\pi.$$

When `frame` is set to 0, the rotation angle is 0 radians, meaning no rotation is applied. If `frame` is set to 180, the rotation angle becomes π radians (180 degrees), and at `frame` $= 360$, the rotation angle completes a full circle with 2π radians (360 degrees), returning objects to their original positions. This calculation enables a smooth, continuous rotation effect around the Y-axis.

Objects rotate around the Y-axis using the following rotation matrix:

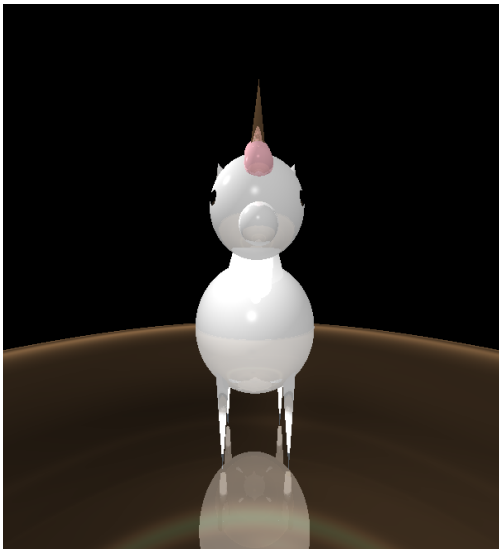
$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

The transformed coordinates are calculated as:

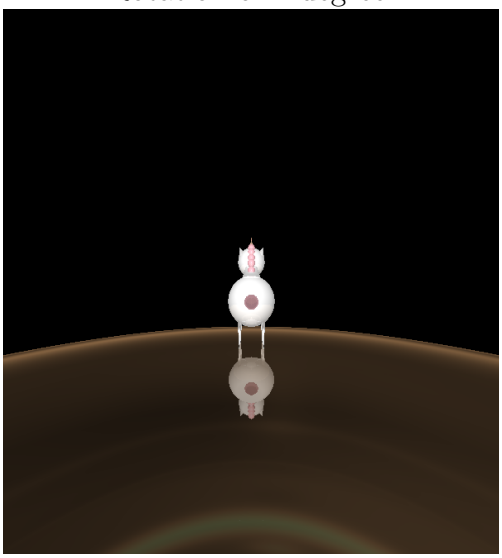
$$x' = \cos(\theta)x - \sin(\theta)z,$$
$$z' = \sin(\theta)x + \cos(\theta)z.$$

This transformation is applied to every object to simulate rotational motion, dynamically changing the scene's visual composition and object positioning.

5.2 Rotation rendering



Rotation of 1 degree.



Rotation of 180 degrees.

5.3 Scene Composition

The program supports two primary geometric objects essential for building the unicorn model and rendering:

- **Spheres**: Defined by a center (C), radius (r), and material properties (color, reflectivity, refractivity). Spheres are fundamental components for creating the unicorn's body, head,

and cheeks, as they represent 3D objects that interact with light in realistic ways.

- **Cones**: Characterized by apex (A), base center (B), radius, and height, with axis vector:

$$V = \frac{B - A}{\|B - A\|}.$$

Cones are used to model the unicorn's legs, tail, and horn, offering a geometric shape that can be easily manipulated within 3D space.

Three types of light illuminate the scene to create realistic shading and visibility effects:

- **Ambient Light**: Uniform across all objects, ensuring that no object remains completely dark. This simulates global light sources like the sun or room lighting.
- **Point Light**: Originates from a specific position in space, mimicking light sources such as light bulbs or lamps. This allows for more localized illumination effects.
- **Directional Light**: Mimics sunlight with a fixed direction, allowing parallel rays of light to illuminate the scene. This light type is used to simulate sunlight or moonlight, which affects the scene uniformly.

5.4 Ray-Object Intersection

Sphere Intersection: The intersection between a ray and a sphere is solved using the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

where:

$$a = D \cdot D, \quad b = 2 \cdot CO \cdot D, \quad c = CO \cdot CO - r^2.$$

This is necessary for determining the points where rays intersect with spherical objects, which is a core component of the ray tracing algorithm for rendering.

Cone Intersection: The intersection with cones is more complex, involving the following coefficients:

$$a = (D \cdot V)^2 - \cos^2(\theta)(D \cdot D),$$

$$b = 2((D \cdot V)(CO \cdot V) - \cos^2(\theta)(D \cdot CO)),$$

$$c = (CO \cdot V)^2 - \cos^2(\theta)(CO \cdot CO).$$

These coefficients are derived from the geometry of the cone and are necessary to determine where the ray intersects with the conical surface, which affects the rendering of shapes like the unicorn's horn and legs.

5.5 Lighting Model

The lighting model used combines various components to simulate how light interacts with surfaces, affecting the final image color:

- **Ambient**: Constant intensity, ensuring that all objects are illuminated uniformly to some degree.
- **Diffuse**: Based on Lambertian reflection, which calculates how light diffuses across a surface:

$$I_{\text{diffuse}} = I_L \cdot \max(0, N \cdot L).$$

The diffuse term simulates the light scattering across rough surfaces, providing a realistic matte finish.

- **Specular**: Based on the Phong model:

$$I_{\text{specular}} = I_L \cdot (\max(0, R \cdot V))^s.$$

Specular reflection simulates the shiny highlights on smooth surfaces, like the unicorn's horn or eyes.

5.6 Recursive Ray Tracing

Recursive ray tracing handles reflections and refractions, crucial for rendering realistic materials like glass or water. The recursion helps trace secondary rays to simulate reflections and refractions[1]:

$$R = L - 2(N \cdot L)N,$$

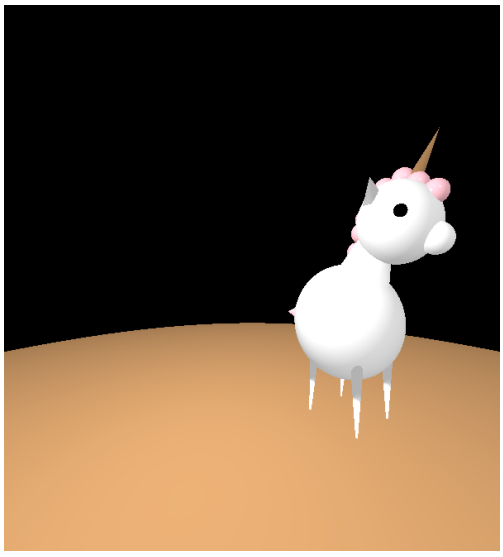
$$\eta = \frac{n_1}{n_2}, \quad \cos(\theta_t) = \sqrt{1 - \eta^2(1 - \cos^2(\theta_i))}.$$

Refracted rays bend according to Snell's law:

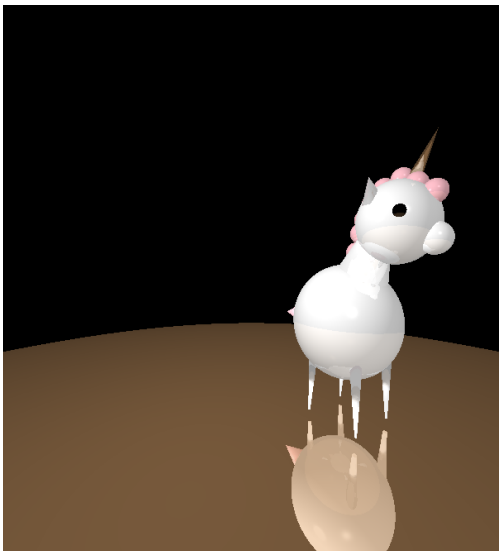
$$T = \eta D + (\eta \cos(\theta_i) - \cos(\theta_t))N.$$

Recursive tracing simulates realistic interactions between rays and surfaces, creating effects like reflection in the unicorn's horn and the surrounding environment. The recursive function computes color contributions from local illumination and secondary rays, blending them based on material properties to generate realistic shading and reflections.

5.7 Adding reflections and refraction to render



Rendering without reflections and refraction.



Rendering without reflections and refraction.

Rendering with reflections and refraction.

6 Results

Rendered Image: The program successfully generates a 3D ceramic unicorn figurine scene composed of spheres and cones with realistic lighting and shading effects.

6.1 Lighting and Material Effects

The rendered image effectively showcases the impact of different lighting types:

- **Ambient Lighting**: Adds a uniform base brightness to the scene, ensuring visibility of all objects.

- **Diffuse Lighting**: Highlights the directional interaction of light sources with objects' surfaces, providing a sense of depth and texture.
- **Specular Lighting**: Creates sharp reflections on the unicorn's features, such as the metallic cones representing the horn, which demonstrates high specular intensity.

Material properties further enhance realism:

- **Reflectivity**: Reflective surfaces on the cones simulate metallic effects.
- **Refractivity**: Partial refraction through some materials shows subtle color blending and light bending effects.

6.2 Dynamic Scene Elements

Object rotation around the Y-axis creates a dynamic perspective, simulating camera movement or object animation. The program's rotation transformations ensure consistent object alignment, maintaining the scene's coherence.

6.3 Performance Considerations

The recursive ray tracing mechanism, while computationally intensive, successfully balances depth and realism. The performance is directly proportional to:

- **Recursion Depth**: Controls the number of reflective and refractive bounces, increasing computational load with each additional level.
- **Scene Complexity**: The addition of objects and light sources requires more intersection tests and lighting calculations.

The final output, a PPM image file, captures the scene in full detail, demonstrating the effectiveness of the implemented algorithms.

7 Summary

This ray tracing program demonstrates the effective use of mathematical and computational methods to render a realistic 3D scene. Key achievements include:

- **Mathematical Rigor:** Employing vector algebra and quadratic equations ensures precise calculations for object intersections.
- **Lighting Realism:** The combination of ambient, diffuse, and specular components produces nuanced lighting effects.
- **Material Properties:** The recursive approach to reflections and refractions adds depth and realism to the scene.
- **Dynamic Scene Management:** Rotational transformations enhance the versatility of the rendering process, allowing for dynamic perspectives.

The rendered image, a stylized ceramic unicorn composed of spheres and cones, illustrates the program's ability to blend mathematical precision with artistic creativity. Future enhancements could include acceleration structures, like bounding volume hierarchies, to optimize intersection testing, or support for additional primitives like cylinders or planes. Moreover, incorporating global illumination techniques could elevate the realism of the scene further.

References

[1] GetIntoGameDev. Realtime raytracing: Refractions. https://www.youtube.com/watch?v=ND2AQFqzvgI, 2023. [Accessed 11-11-2024].

[2] Steve Hollasch Peter Shirley, Trevor David Black. Ray tracing in one weekend. Technical report, Limnu, 2018.

[3] QuantitativeBytes. Ray tracing [c++ sdl2] - episode 8 - cones cylinders. https://www.youtube.com/watch?app=desktop&v=UTz7ytMJ2yk&t=1557s, 2021. [Accessed 3-11-2024].

[4] Marina Rodriguez. https://github.com/marin802/comp4046/blob/main/, 2024.