Ray Tracing with Recursive Reflection and Shadow Calculation for 3D Rendering of Simple Geometric Shapes

Abstract

This project aims to implement a ray tracing technique with recursive reflection for 3D rendering of simple geometric shapes, specifically a sphere and a quad polygon. The program takes into account the shading and shadows caused by a single light source. The rendered image is then saved as a binary file.

Introduction

Ray tracing is a computer graphics technique used to generate photorealistic images by simulating the path of light rays as they interact with objects in a virtual 3D scene. This method can handle complex phenomena such as reflections, refractions, and shadows. In this project, we implement a basic ray tracing algorithm to render simple geometric shapes, namely a sphere and a quad polygon, with a single point light source. The algorithm accounts for reflection, shading, and shadows, producing an image with a more realistic appearance.

When a ray of light hits a surface in a 3D scene, it may bounce off the surface and continue on in a new direction. This is known as a reflection ray. The color of the pixel at the point where the reflection ray intersects with another object or surface is calculated by blending together the colors of the reflection ray and the original surface based on the weighting factors w1 and w2.

The weighting factor w1 represents the percentage of the shading value at the current intersection point that contributes to the final color of the pixel, while w2 represents the percentage of the shading value returned by the reflection ray that contributes to the final color.

By adjusting the weighting factors, it is possible to create different effects in the final rendered image, such as emphasizing the reflection or the original surface. These weighting factors are commonly used in ray tracing algorithms to create more realistic and visually appealing images.

Methodology

The program is implemented in C++ and structured into several functions that handle different aspects of the ray tracing process. The main components include assigning values to 3D points, calculating transformations, constructing rays, detecting intersections, and calculating shading.

First, there is definitions of point3d struct with three float fields representing x, y, and z coordinates and ray struct that contains two pointers to Point3d objects, representing the endpoints of a ray.

Then some helper functions performing vector math on Point3d objects, including subtracting, calculating the dot product and cross product of two vectors, and normalizing a vector:

1. Assigning Values: Using the predefined values in `mymodel.h`, we initialize the scene with a sphere, a polygon, cameras, and a light source.
2. Transformations: Calculation of translation inverse and rotation matrices to transform points between world and camera coordinates.
3. rotation_transpose: It computes the transpose of the rotation matrix that would align the vector from p2 to p3 with the positive z-axis. It does this by first determining the normal n of the vector from p2 to p3, then determining the vector u that is orthogonal to both n and the vector from p2 to p3, and lastly determining the vector v that is orthogonal to both n and u. The rotation matrix that results is then placed in a matrix.

Additionally, the main functions are:

4. Ray Construction: A ray is constructed for each pixel of the image, representing a potential path from the camera to the objects in the scene. With an image pixel position and camera settings, this function generates a ray in 3D space.

Pixel location (i,j), focal length, picture boundaries (xmin, xmax, ymin, ymax), a camera transformation matrix Mcw, and a viewpoint in world coordinates are all input parameters (vrp).

The result is a Ray object with the origin point (ray->a) and a normalized direction vector (ray->b).

The function first uses the camera transformation matrix and focal length to map the pixel position to a 3D point in camera coordinates, and then uses the viewpoint to convert that point to world coordinates. The difference between the 3D point and the perspective is used to construct the direction vector, which is then normalized. The Ray object stores the resultant origin and direction.

5. Ray_sphere_intersection: This function computes the point at which a ray intersects a sphere. It accepts a Ray and an SPHERE as input, as well as pointers to a Point3d to store the sphere's normal at the intersection, a Point3d to record the intersection location, and a float pointer to save the sphere's material attribute. It returns the distance from the ray's origin to the junction point, or 0 if no intersection exists. The function first computes the intersection equation's quadratic coefficients A, B, and C, and then determines if the discriminant D is positive. If D is positive, the function computes the intersection point, the normal of the sphere at the intersection point, and the material property, and then returns the distance from the ray's origin to the junction point. The function returns 0 if D is not positive.

For Ray_polygon_intersection, we need the following helper functions:

6. Find_project_plane: This function accepts an array of length 3 representing the surface normal, a 2D array of size 4x3 representing the four vertices of a quadrilateral in 3D space, and a 2D array of size 4x2 containing the vertices' 2D coordinates projected onto a plane perpendicular to the surface normal.

7. Point_in_polygon_test: The point-in-polygon test determines if a given point is within or outside of a polygon defined by a collection of vertices. It accepts as input the number of polygon vertices (number of vert), arrays of the vertices' x and y coordinates (vert x and vert y), and the x and y coordinates containing the test point (test x and test y). The function then runs around the polygon's edges, checking if the test point is to the left or right of the edge. If the test point is to the left of an odd number of edges, the polygon is within, and the method returns 1. If the test point is to the left of an even number of edges, it is outside the polygon and yields 0.

Using these two helper functions, the Ray_polygon_intersection can be defined as below:

8. Ray_polygon_intersection: This function computes the intersection of a ray with a four-vertex polygon. The function accepts a Ray and a POLY4 structure

as input (which contains the coordinates of the four vertices and the polygon's normal vector) and returns the distance between the ray origin and the intersection point, or 0 if no intersection exists. The function also provides the polygon's surface normal, intersection point, and diffusion coefficient. It initially computes D and the point of intersection of the ray with the polygon. The intersection point is then projected onto a plane specified by the polygon's normal vector, and the projected point is checked to see if it is within the polygon. If so, the function returns the distance between the origin of the ray and the intersection point. If not, it returns 0.

With introducing Ray_sphere_intersection and Ray_polygon_intersection, we can define Ray_object_intersection function as below:

9. Ray_object_intersection: The program tests for intersections between rays and each object (sphere and polygon) to determine the nearest intersection point. In a 3D environment, this function computes the intersection point of a given ray and either a sphere or a polygon object.
It yields the distance between the ray's origin and the intersection point, as well as the object's normal at the intersection point and the intersection pointolated color at that point.
It initially uses two distinct helper functions to check for an intersection with a sphere and a polygon object.
If neither intersection happens, the method returns 0.0, indicating that there is no intersection. If just one junction occurs, the method returns the distance to the intersection and sets the normal and intersection point to the values provided by the appropriate helper function.
If both intersections happen, the method returns the intersection with the shortest distance based on which intersection happened first.
The function additionally assigns the nearest intersection's normal and intersection point to the output parameters n and intersection point, as well as the intersection pointolated color to kd.

10. Shading: If the intersection point is illuminated, we calculate its shading based on the object's diffuse coefficient and the angle between the surface normal and the light vector. An additional reflection ray is computed to handle reflective surfaces. The light source position lrp, the normal vector n, the intersection point p, the diffuse coefficient kd, and the light intensity ip are all inputs into this function. The shading value is calculated by taking the dot product of the normalized light vector and the normal vector and multiplying it by the diffuse coefficient and the light intensity. If the cosine is negative, the shading value is set to zero, indicating that the surface is not lighted since it is

facing away from the light. After that, the shading value is returned as an unsigned char.

11. Shadows: To check for shadows at an intersection point, a new ray is cast towards the light source. If this ray intersects with any object before reaching the light source, the point lies in shadow.

12. Ray Tracing: We apply a recursive ray tracing function to compute the color of each pixel in the final image. The ray_tracing function accepts a Ray, two shapes, an SPHERE and a POLY4, as well as a light location lrp and intensity.
The function initially calculates the ray's intersection point P, normal vector n, and intersection point with the first object it intersects (either the sphere or the polygon). If the intersection is valid (i.e., P is greater than zero), the function computes using it and also checks for shadows and reflections with helper functions and returns the shading color C at the intersection point using the shading function, reflect function, w1, and w2. The method returns 0 if the intersection is not valid.

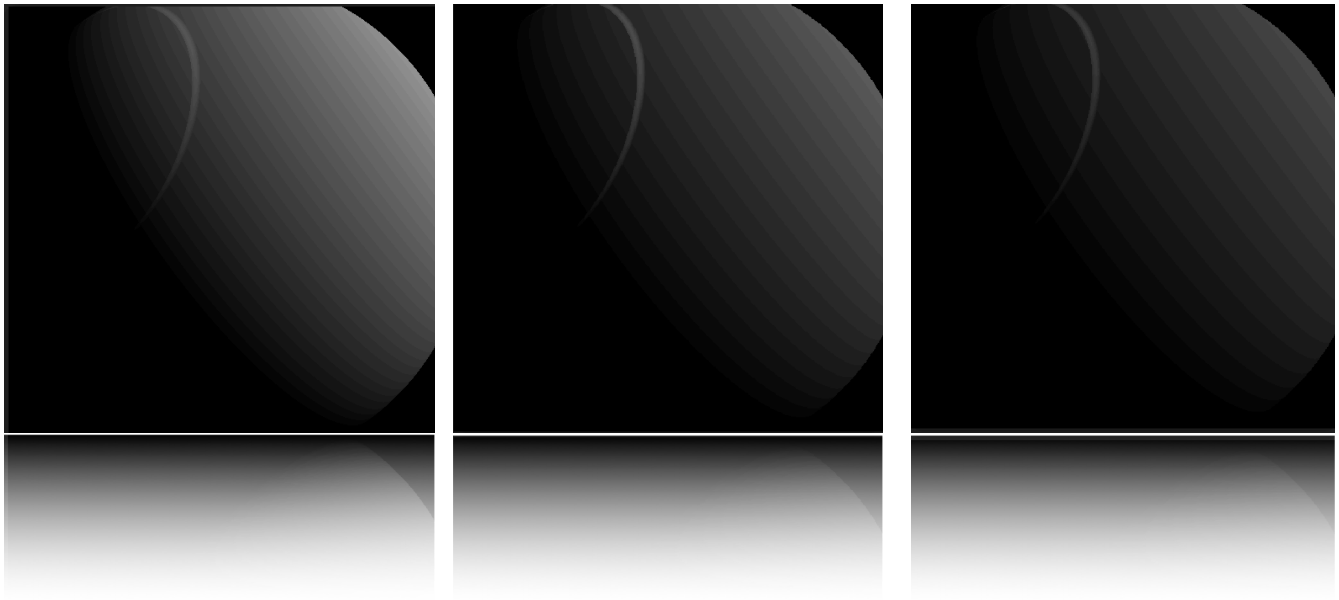
Finally, in the main function, we can save the rendered image as a binary file. The main function loads the scene from mymodel.h and does ray tracing on each pixel in the picture plane.
It creates rays for each pixel and initializes transformation matrices. The ray tracing function is then used to identify the color at the junction point of each ray with the scene. The findings are saved in an image buffer before being written to a binary file.

Results

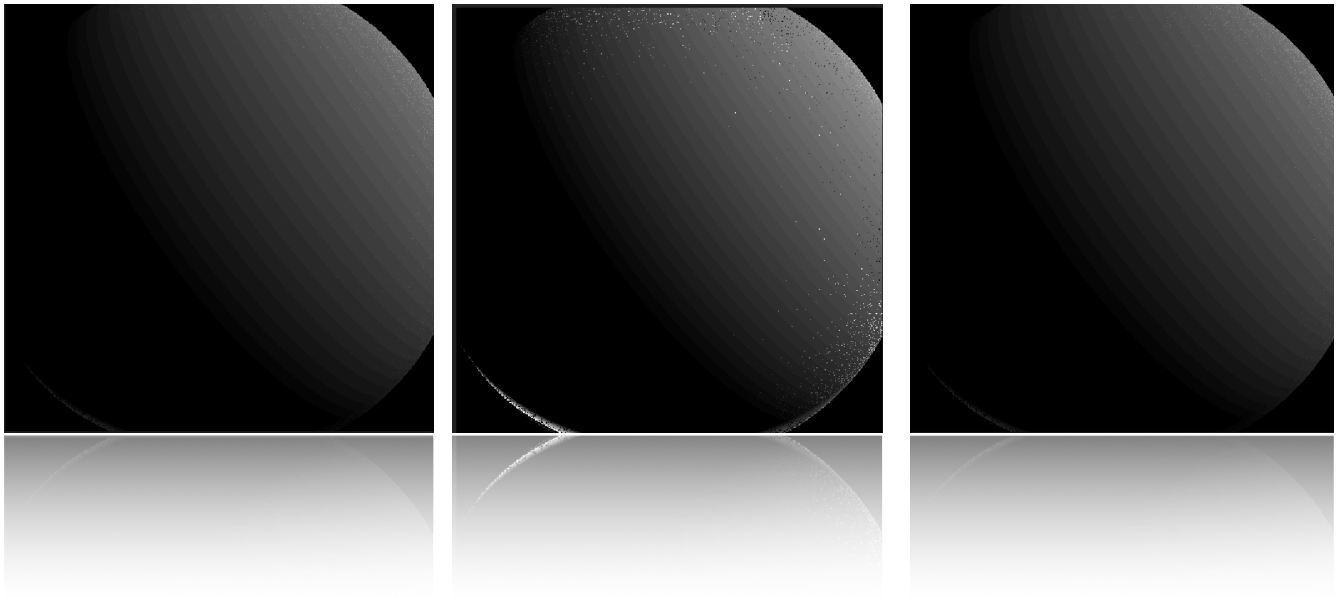
The resulting program renders a simplistic 3D scene with a sphere and a quad polygon, taking into account varying levels of shading and shadows produced by a single point light source. The output image is rendered in a 512x512 resolution and stored in a binary file.

The code is tested using the default parameters in the model.h and then some parameters were changed to create different scenes. If we want to create a reflective image, we should set w1 and w2 to appropriate values that balance the contribution of the surface and the reflection color. The exact values of w1 and w2 depend on the specific material properties and lighting conditions in the scene. However, it's important to note that setting the values of w1 and w2 is not the only factor that affects the appearance of a reflective surface. Other

factors such as the angle of incidence of the incoming light, the roughness of the surface, and the environment map used to generate reflections can also have a significant impact on the final appearance of the surface. So, I have experimented different parameters and scenes and provided the generated results. The following 3 figures show the rendered image with w1 and w2 of sphere as 0.5 and reducing the sphere kd from left to right.



The following 3 figures show that with changing the light and camera position, we might have some dots for shadow:



Conclusion

This project demonstrates the implementation of a basic ray tracing algorithm for rendering simple 3D objects with shading and shadows. It serves as a starting point for further development, including the addition of more complex geometries, multiple light sources, and advanced rendering techniques such as transparency and refraction. Extensions of the project could also include optimizations to improve rendering performance and adding support for other image formats.

Acknowledgments

The basis of this project is the C++ program implemented for the Assignment 2, the pseudo-algorithms provided in the slides, and the values in the `mymodel.h` header file, which has allowed for a deeper understanding of ray tracing techniques and their applications in computer graphics.