Image Rendering with Phong Illumination and Gouraud Shading Model

Weiji Xie
521030910421
xieweiji249@sjtu.edu.cn

Abstract

This project implements a basic software renderer that simulates lighting and shading in 3D scenes using the Phong illumination model in combination with Gouraud shading. The renderer constructs a 3D scene composed of mesh-based geometric objects, performs per-vertex lighting computations based on material and lighting properties, and projects the illuminated geometry into 2D image space. Rasterization is carried out based on barycentric interpolation with z-buffering for visibility. The system is implemented entirely in Python, without the use of external graphics libraries such as OpenGL.

1 Introduction

Realistic image synthesis requires accurate modeling of how light interacts with surfaces in a scene. In both real-time and offline rendering systems, physically inspired illumination models are used to approximate this interaction. In this project, we implement a classical local illumination model — the **Phong illumination model** — paired with **Gouraud shading**, a vertex-level shading approach that interpolates lighting values across polygonal surfaces.

The entire rendering pipeline is built from scratch in Python, serving as a demonstration of fundamental rendering techniques. The renderer operates in three major stages:

- **Scene Construction:** Geometric primitives such as cones, spheres, and cubes are converted into triangular meshes. Each object is assigned material properties such as diffuse and specular reflectance, shininess, and base color. We also implement Surface Subdivide, which recursively splits each triangle into four, to improve surface resolution and smoothness in the resulting mesh.
- **Illumination:** Using the Phong model, lighting is computed per vertex based on surface normals, viewer direction, and the contributions from multiple point light sources. Each light has a position, intensity, and color.
- **Projection and Rasterization:** The 3D vertices are projected into 2D screen space using a perspective projection defined by the camera's field of view, aspect ratio, and near-plane. The triangles are then rasterized using barycentric coordinate interpolation to interpolate vertex colors across the triangle surface.

This renderer supports multiple lights and objects, smooth shading, and depth-aware rendering. By implementing each stage explicitly, we gain insight into the principles that underlie modern rendering pipelines.

2 Methods

2.1 Scene Construction

Our renderer constructs a scene using three core components: **Objects**, **Lights**, and a **Camera**.

Objects. Each object in the scene is represented by a triangular mesh and an associated material.

- **Mesh:** A geometric object is defined by its vertices and triangular faces. To support subsequent rendering, we compute face and vertex normals. Additionally, we implement *Surface Subdivide* to recursively split each triangle into four smaller ones, allowing for smoother geometry and better surface resolution.
- **Material:** The material defines the object's appearance using ambient (k_a), diffuse (k_d), and specular (k_s) reflection coefficients, shininess (n), distance attenuation parameters (a, b, c) and base RGB color.

Light Sources. Each light source is defined with the following attributes:

- **Position**
- **Light Intensity**
- **Color:** specified in RGB space within the range $[0, 1]$

These parameters influence the illumination model by modulating the contribution of each light to the shading computation, in accordance with the Phong reflectance model.

Camera. The camera serves as the viewpoint for image generation, projecting the 3D scene onto a 2D image plane. It is parameterized as follows:

- **Position:** The camera origin in world space
- **View Direction:** A normalized vector indicating the direction from the camera toward the scene target
- **Up Vector:** A vector defining the upward direction relative to the camera's orientation; typically set to $[0, 0, 1]$
- **Field of View (FOV):** The vertical angle of view, specified in degrees (default: $60°$)
- **Aspect Ratio:** The ratio of image width to height (default: 1.5)
- **Resolution:** The number of pixels per unit length on the image plane (e.g., 800 pixels per meter)

These parameters are used to construct the camera's perspective projection matrix, define the image rasterization grid, and compute the primary rays required for rendering.

2.2 Phong Illumination Model

Illumination at each vertex is computed using the Phong reflection model, expressed as:

$$I_\lambda = I_{a\lambda} k_a + \sum_{1 \le i \le m} S_i \left[k_d I_{d\lambda}^{(i)} (\mathbf{l}_i \cdot \mathbf{n}) + k_s I_{s\lambda}^{(i)} (\mathbf{r}_i \cdot \mathbf{v})^n \right],$$

where the distance attenuation factor S_i for the i-th light source is defined as:

$$S_i = \frac{1}{a + b d_i + c d_i^2}.$$

with the following notations:

- I_λ: Final light intensity for color channel λ (e.g., R, G, or B)

- $I_{a\lambda}$: Ambient light intensity for channel λ, defined by the material color
- $I_{d\lambda}^{(i)}, I_{s\lambda}^{(i)}$: Diffuse and specular intensity of light source i for channel λ, defined by material color times the light color
- k_a, k_d, k_s: Material coefficients for ambient, diffuse, and specular reflection
- n: Shininess exponent controlling the size of the specular highlight
- a, b, c: Attenuation coefficients controlling how light intensity diminishes with distance
- $\mathbf{n}$: Surface normal vector at the vertex
- $\mathbf{l}_i$: Unit vector from the surface point to light source i
- $\mathbf{v}$: Unit view vector from the surface point to the camera
- $\mathbf{r}_i$: Unit reflection vector of $\mathbf{l}_i$ with respect to $\mathbf{n}$
- d_i: Distance from the vertex to light source i

Gouraud Shading. To achieve smooth shading across surfaces, the illumination values I are computed per vertex as described above. These per-vertex intensities are then linearly interpolated across each triangle's interior during rasterization, following the Gouraud shading technique. This approach efficiently approximates varying illumination while maintaining computational tractability.

2.3 Rasterization

Rasterization is the process of converting geometric primitives, such as triangles, into pixel data on the image plane. It is a fundamental step in the rendering pipeline where 3D scene information is transformed into a 2D image representation. The rasterization process involves several key stages: vertex projection, face traversal with interpolation, and depth buffering.

Vertex Projection The first step in rasterization is projecting 3D vertices onto the 2D image plane. Given a camera model, each vertex in world coordinates is transformed into camera space by applying an inverse camera transformation matrix. Subsequently, vertices are projected onto the image plane via perspective projection. The projected 2D coordinates are then adjusted to pixel coordinates in the image frame, accounting for image width and height. The depth value z in camera space is retained for later depth testing.

Face Traversal and Interpolation Each triangular face, is rasterized by determining the pixels it covers on the image plane. To optimize this, a bounding box surrounding the projected triangle is computed, clamping the region to the image boundaries to limit rasterization to relevant pixels.

Within this bounding box, each pixel center is tested for inclusion inside the triangle using barycentric coordinates. The barycentric coordinates (u, v, w) represent the relative weights of the triangle's vertices that define any point inside the triangle. They are computed using vector algebra and dot products, ensuring efficient and numerically stable calculations.

Pixels inside the triangle are interpolated in depth and vertex color by weighting the vertex values according to the barycentric coordinates. This interpolation produces smooth color gradients and enables depth testing.

Z-Buffering To correctly handle occlusions, rasterization maintains a depth buffer (z-buffer) that records the closest depth value per pixel encountered so far. For each pixel inside the triangle, the interpolated depth is compared against the current value in the z-buffer. If the new depth is closer to the camera, the pixel's color and depth values in the frame buffer and z-buffer, respectively, are updated.

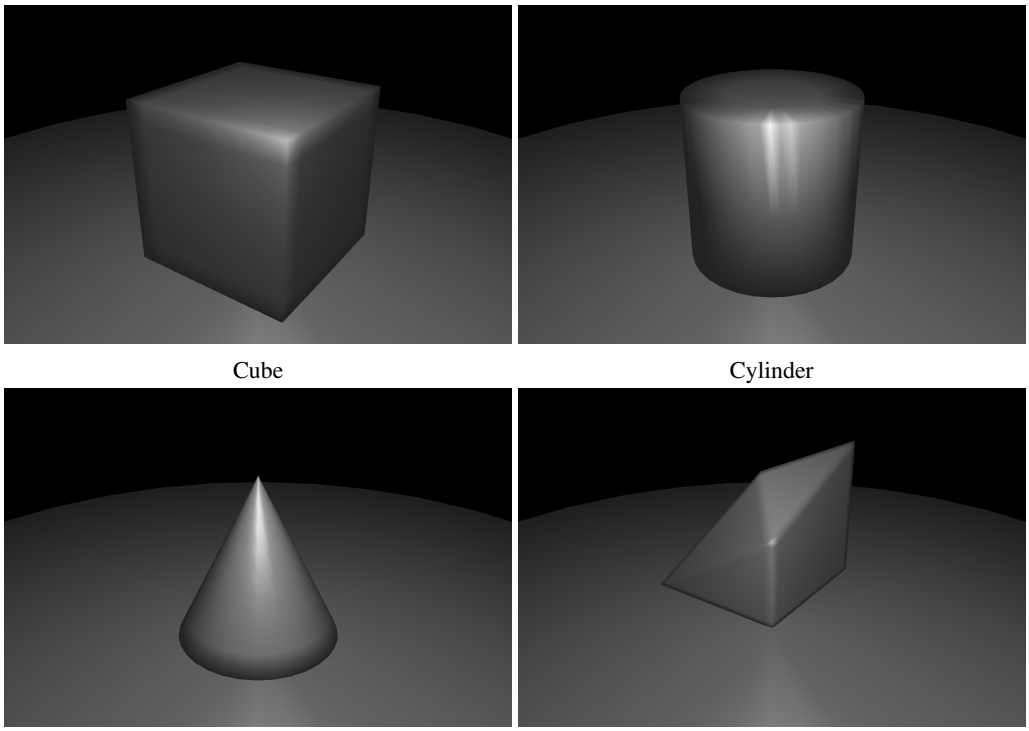
Color Discretization After processing all faces, the frame buffer, which stores interpolated colors in floating-point format normalized between 0 and 1, is discretized into an 8-bit per channel format suitable for display. Values are clipped to the valid range and scaled to $[0, 255]$ to produce standard 8-bit per channel RGB images suitable for display or saving as image files.

3 Results

We present a series of visual results produced by our custom rasterization pipeline to demonstrate its correctness, flexibility, and rendering quality. The experiments encompass diverse object geometries, lighting conditions, material properties, and diverse camera viewpoints.

3.1 Basic Object Rendering

To evaluate geometric correctness and fundamental shading accuracy, we render standard geometric primitives including a cube, cylinder, cone, and polyhedron. Each object is uniformly grey and illuminated by a single white point light source.



Cube Cylinder

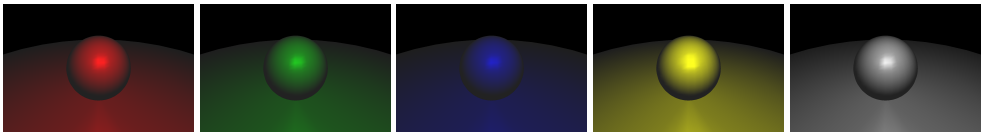
Cone Polyhedron

Figure 1: Rendering of basic geometric objects under a single light source.

3.2 Color Variation

To assess the pipeline's color fidelity and interpolation capabilities, we conduct two sets of experiments. The first set renders a white sphere illuminated by single-colored point lights to verify lighting color application. The second set applies two light sources, one fixed to white and another one with varying colors, testing the interpolation of vertex-level color attributes across fragments.

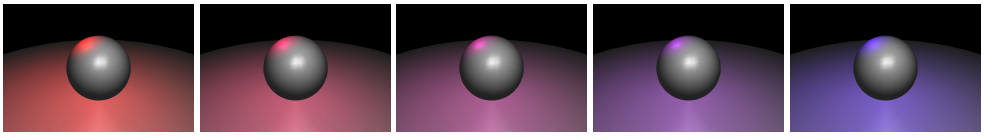
3.2.1 Single Light



Red Green Blue Yellow White

Figure 2: White sphere with different light colors.

3.2.2 Two Light



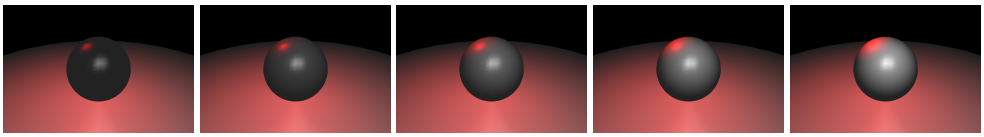
Red Pink Magenta Purple Blue

Figure 3: White sphere with one white light and one light with varying colors

3.3 Material Property Analysis

To better understand how material parameters affect rendered appearance, we conduct controlled experiments by varying key reflectance coefficients in the Phong illumination model. Specifically, we explore how changing the ambient reflectance coefficient (k_a) and the specular reflectance coefficient (k_s) influences the visual output of a rendered sphere. All other lighting and material parameters are held constant throughout each experiment to isolate the effect of the varying coefficient.

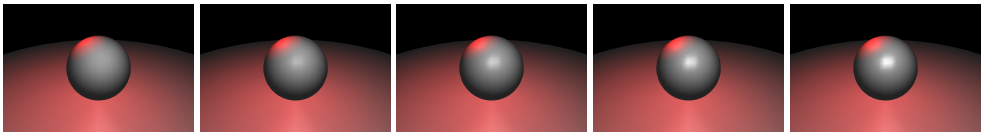
3.3.1 Ambient Reflectance k_a Variation



$k_a = 0.00$ $k_a = 0.25$ $k_a = 0.50$ $k_a = 0.75$ $k_a = 1.00$

Figure 4: Sphere with varying ambient coefficient k_a

3.3.2 Specular Reflectance k_s Variation



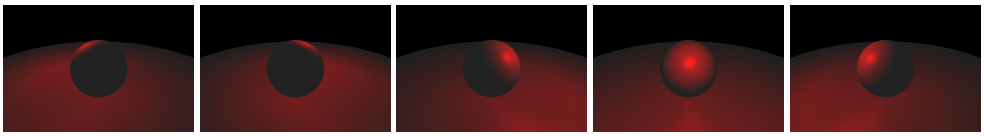
$k_s = 0.00$ $k_s = 0.25$ $k_s = 0.50$ $k_s = 0.75$ $k_s = 1.00$

Figure 5: Sphere with varying specular coefficient k_s

3.4 Multiple Viewpoint Rendering

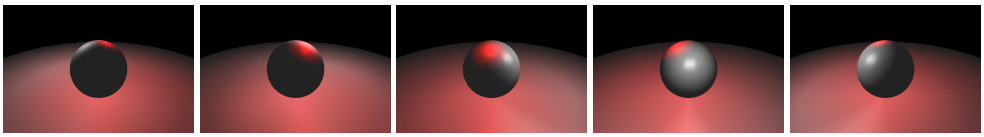
To validate the renderer's perspective consistency, lighting stability, and geometric robustness under varying viewpoints, we render image sequences with a rotating camera setup. These sequences simulate how a viewer would perceive the object from different angles, revealing the effectiveness of our shading model and rasterization accuracy in dynamic scenarios.

In this experiment, the virtual camera is rotated around the target object while maintaining a fixed distance and looking direction. The light sources remain static in world space, allowing us to observe how surface reflections and shading evolve across different viewpoints.



0° 72° 144° 216° 288°

Figure 6: Red light with rotating camera



0° 72° 144° 216° 288°

Figure 7: White and red lights with rotating camera

4 Conclusion

This project presents a complete implementation of a 3D renderer that combines the Phong illumination model with Gouraud shading to simulate realistic lighting effects. By constructing the rendering pipeline from scratch in Python, we developed a deeper understanding of the key components of computer graphics, including mesh representation, camera modeling, local illumination, and rasterization.

Through per-vertex lighting calculations and barycentric interpolation, the system efficiently produces smooth shading and visually convincing images of 3D scenes. The incorporation of surface subdivision further enhances the visual fidelity of curved surfaces, while the z-buffer algorithm ensures correct visibility handling.

Despite its simplicity, our renderer successfully demonstrates the core principles behind modern graphics engines. This foundational framework also opens up possibilities for further extensions, such as implementing texture mapping, shadow casting, or physically based rendering techniques. Overall, the project serves as both an educational tool and a stepping stone toward more advanced graphics programming.