Linear Algebraic Approach to Ray Tracing/3D Graphics

Anson Wang(ansonw2), Jason Ding(jasondin)

1 Introduction

Modern 3D graphics heavily rely on linear algebra to simulate realistic shading/texture manipulations on rendered objects. Ray tracing provides an intuitive way to simulate graphics, using linear algebra and geometry to mimic realistic behaviors of materials and objects. Although ray tracing is usually associated with more physics based implementations, the math behind the calculations consist of many linear transformations and coordinate changes.

The goal of this project is to implement a ray tracing engine to simulate viewing details of various spherical objects in a virtual space that maps points in this fashion: object space $\rightarrow$ world space $\rightarrow$ camera space $\rightarrow$ clip space $\rightarrow$ normalized device coordinates (NDC) $\rightarrow$ screen space, and to use this pipeline for ray intersections and perspective projection. To support arbitrary object movement and scaling, we represented each object using a 4×4 homogeneous transformation matrix M, which contains information on object translation, rotation, and scaling in a single linear operator. Rays are then transformed into the object's local space using its inverse M^{-1}, allowing us to treat all objects as the "default" orientation (e.g. unit spheres at the origin). This is a slightly different approach to the standard geometrical method in ray tracers, and while this method may hinder performance, it allows a much more simplified mathematical framework.

A camera view matrix was also created, which is a change-of-basis transformation that converts world coordinates into camera coordinates using an orthonormal matrix derived from the camera's position and direction. A projection matrix was also implemented that allows visualization of 3D points and wireframe objects even though the renderer is ray based instead of rasterized.

Different lighting models were also experimented with, such as Lambertian diffuse shading and Blinn-Phong specular highlights. Those are just clever ways of utilizing dot products and vector projections and we extended them to support reflective surfaces via recursively traced rays.

2 Configuration

The project is written in Python with NumPy for basic vector and matrix manipulation and PyGame for recording live inputs. The full repository, including past commits, are included here: `https://github.com/JaddotWuzHere/raytracer_21-241`

3 Implementation Behavior & Examples

3.1 Object Transformations

We used 4×4 homogeneous matrices to represent objects that can be translated, scaled, and rotated. Each object has a local coordinate system (the object space), a transformation matrix M mapping it into world space, and its inverse matrix M^{-1} used to transform rays into object space.

A point in world coordinates P_{world} is expressed in homogeneous form as

$$P_{world} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

A transformation is thus represented as $M = T \cdot S$ where $T(x, y, z)$ is a 4×4 translation matrix, and $S(s)$ is a scaling matrix.

Objects are defined as unit spheres at the origin in object space, so calculating the intersection between them and rays become simpler quadratic equations. A ray in world coordinates is expressed as

$$O_{world}, \quad D_{world}$$

which is transformed into object space using

$$O_{local} = M^{-1} O_{world}, \qquad D_{local} = M^{-1} D_{world}.$$

Matrices are created in `util.translation` and `util.scale`. Spheres are instantiated via

```
M = T @ S
M_inv = np.linalg.inv(M)
```

inside `makeSphere`. Ray transformation occurs in `calcHitObj`, where both the origin and direction are implemented into homogeneous vectors and transformed by `sphere.M_inv`.

3.2 Ray-Sphere Intersection

A ray is in the form $O + tD$, where O represents its origin and D is the unit vector representing its direction. Rays intersecting with spheres boils down to solving

$$\|O + tD\|^2 = 1,$$

which expands into the quadratic equation

$$at^2 + bt + c = 0,$$

with

$$a = D \cdot D, \qquad b = 2(O \cdot D), \qquad c = O \cdot O - 1.$$

The discriminant

$$\Delta = b^2 - 4ac$$

determines whether the ray intersects the sphere or not.

In `calcHitObj`, the coefficients are calculated using `dot`, the discriminant using `discriminant(a, b, c)`, and the roots with `findRoots` such that the smallest positive `t` is returned. A hit is then wrapped in a `HitRecord`, which stores the parameter t, the position, the surface normal at that position, and the object it collided with.

3.3 Surface Normals

The normal vector at the intersected point on a unit sphere is simply just

$$N_{local} = \frac{P_{local}}{\|P_{local}\|}.$$

In `normalVec`, the point in world space is converted into object space via `PLocMatr = obj.M_inv @ PWrldMatr` and the normal is computed by normalizing `PLoc`. The normal in world space is then computed back by via `NWrldMatr = obj.M @ NLocMatr` and then normalizing it again.

3.4 Camera Frame and View Matrix

The camera has its own orthonormal basis defined using `forward`, `right`, and `up`, all of which are vectors.

$$f = \frac{(lookAt - pos)}{\|lookAt - pos\|}, \quad r = f \times WORLD\_UP, \quad u = r \times f$$

These vectors construct the rows of the change-of-basis matrix that maps world coordinates into camera coordinates.

The view matrix is

$$V = \begin{bmatrix} r_x & r_y & r_z & -r \cdot pos \\ u_x & u_y & u_z & -u \cdot pos \\ -f_x & -f_y & -f_z & f \cdot pos \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the basis is computed in `Camera.__init__` and the matrix is constructed in `createViewMatr`.

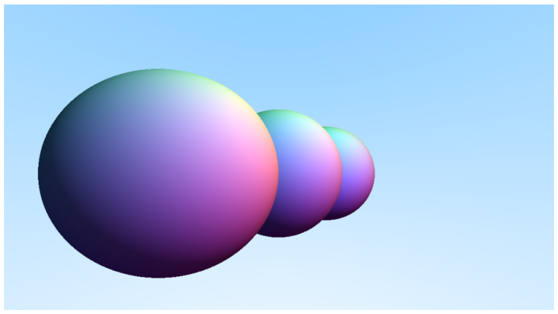

Figure 1: Three side by side spheres positioned at increasing depth

3.5 Perspective Projection

A 4×4 projection matrix is used to project 3D camera coordinates onto 2D screen coordinates.

$$P = \begin{bmatrix} f/A & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \end{bmatrix}$$

The point then simply travels from world space to camera space via $P_{cam} = V \cdot P_{wrld}$, and then into clip space via $P_{clip} = P \cdot P_{cam}$. The NDC is then obtained through homogeneous divide

$$(x, y, z)_{ndc} = \frac{1}{w}(x, y, z)$$

and the screen coordinates are then mapped to $[0, w] \times [0, h]$.

The projection matrix is created in `createProjMatr` and the entire coordinate pipeline is implemented in `scene.projPt`.

3.6 Lambertian Diffuse Shading & Blinn-Phong Speculars

Lighting is computed using both Lambertian diffuse shading and Blinn-Phong specular highlights. The diffuse term is

$$L = \frac{lightPos - P}{\|lightPos - P\|}, \qquad \text{diff} = \max(0, N \cdot L).$$

To calculate the specular highlights, the half vector H is used such that

$$H = \frac{L + V}{\|L + V\|}$$

and the specular intensity is

$$\text{spec} = \max(0, N \cdot H)^n.$$

In the method `lambertianShade`, N, L, V, and H are computed as well as the ambient, diffuse, and specular components. The constants `DIFFUSE_COEF`, `SPECULAR_COEF`, and `SHINE` can be adjusted to achieve varying effects.

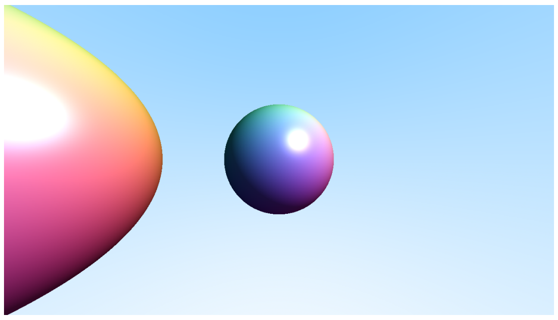

Figure 2: Two glossy spheres reflecting a single defined light source

3.7 Recursive Reflections

Reflective surfaces generate recursive ray paths based on

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

where the reflected ray origin is offset by ϵ to avoid self intersection which may cause performance and visual issues. The reflected color is calculated by blending the local and reflected colors of the rays.

Reflections are implemented in `traceRefl`. The base case is that if the depth is greater than 2, the sky gradient is returned. The local shading is computed via `lambertianShade`, and the reflection direction is computed using dot products. `traceRefl` is called recursively if the ray isn't "lost", and the local and reflected color are based on the reflectivity of the object.

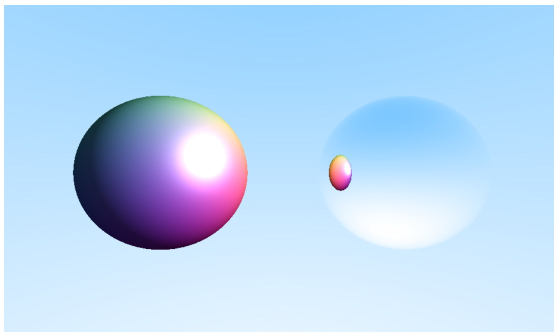

Figure 3: Two side by side spheres, one reflective of the sky and the other sphere

3.8 Camera Movement and Rotation

Rotations around arbitrary axes is implemented in `rotateAroundAxis` by breaking down a vector into its projection onto the axis and its perpendicular component. The perpendicular part rotates using

$$v' = v_\perp \cos\theta + (a \times v_\perp \sin\theta)$$

which is Rodrigues' rotation formula. Camera yaw and pitch update the basis vectors `forward`, `right`, and `up` using this function.

Live camera movement is also implemented via PyGame and can be controlled with WASD controls as well as mouse movement, however it only works in theory as performance fixes have not been fully implemented and runs at a very low 0.5 frames per second. For a future endeavor, we propose moving the code structure to work on top of Numba or perhaps a different, lower level language to increase performance enough for a stable 20 frames per second to allow a smoother traversal around the digital environment.

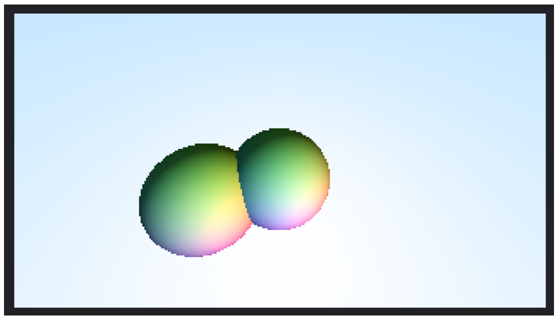

Figure 4: A scene viewed from a different angle in PyGame

To launch the ray tracer in PyGame, simply run `live.py`.

3.9 Wireframe Projection for Camera

The wireframe renderer projects a cube defined in world space directly onto the 2D screen. Unlike the 2D ray traced image, it uses explicity matrix multiplication and works in real time.

A unit cube is defined by 8 vertices and 12 edges. For each vertex P_{world}, the full transformation pipeline is applied. The resulting normalized coordinates are mapped to pixel coordinates on the screen. This functionality is only seen when the program is run from `live.py`.

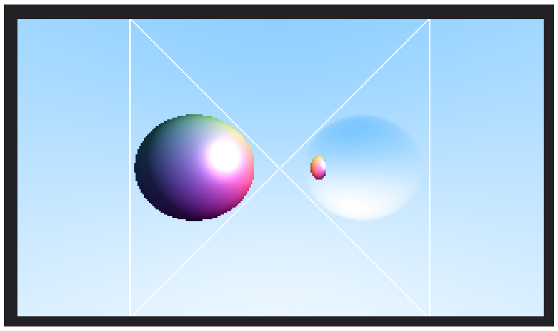

Figure 5: Wireframe render in PyGame, with resolution decreased for performance

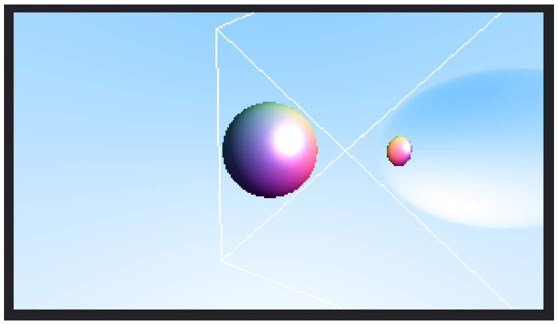

Figure 6: Same scene viewed from a different angle

3.10 Sky Gradient Background

The background color is simply a vertical blend between two RGB values based on the y component of the ray direction

$$t = \frac{d_y + 1}{2}.$$

3.11 Results

This implementation of a ray tracer demonstrates how a complete 3D rendering pipeline can be done almost entirely in linear algebra.

Objects are translated and scaled using matrices that behave as expected, where spheres retain their shape under uniform scaling and their positions in world space also match the transformations applied to them. The correct appearances of intersections confirm that rays are being intersected properly in object space.

As for camera behavior, movement through the scene produces the same parallax and perspective distortion as one would expect in reality. The consistency between the rendered wireframe overlay and the rest of the scene also support this fact.

Lambertian shading and Blinn-Phong specular highlights behave as expected under changes in light source position and intensity, as well as surface orientation. Surfaces facing the light show brighter speculars, while angles facing away are rendered darker. Reflective objects also exhibit correct recursive behavior, as nearby objects can be clearly seen rendered off of their surfaces.

Overall, the results confirm that linear algebra can be used effectively to create a ray tracer than can consistently and accurately render objects with similar real life properties in a digital space. Geometries can be expressed using matrices and vector normalization, and the rest of the functionality simply stems from pipelines that apply change-of-basis matrices. The behavior of this ray tracer matches what is expected.

3.12 Discussion

To generate an image, simply run `genImage.py` and execute the command `xdg-open output.ppm` in `.venv` to open the image. To enter a live simulation, run `live.py`.

Live camera movement is implemented via PyGame and can be controlled with WASD controls as well as mouse movement, however it only works in theory as performance fixes have not been fully implemented and runs at a very low 0.5 frames per second. For a future endeavor, we propose moving the code structure to work on top of Numba or perhaps a different, lower level language to increase performance enough for a stable 20 frames per second to allow a smoother traversal around the digital environment.

Ambient lighting is included during the calculations of Lambertian diffuse shading and Blinn-Phong speculars to simulate more realistic lighting, as shadows would become pitch black otherwise.

4 Bibliography

All images not cited were generated by the authors. OpenGL images come from the second source.

Wikimedia Foundation. (2025, November 13). Lambertian reflectance. Wikipedia. https://en.wikipedia.org/wiki/Lambertian_reflectance

Tutorial 3: Matrices. (n.d.). https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Wikimedia Foundation. (2025, November 25). Rodrigues' rotation formula. Wikipedia. https://en.wikipedia.org/wiki/Rodrigues'_rotation_formula

5 Appendix

5.1 sphere.py

```
# |P-C|^2 = R^2
from geometry import *


class Sphere(Renderable):
    def __init__(self, M, M_inv, reflectivity):
        self.M = M
        self.M_inv = M_inv
        self.reflectivity = reflectivity

    def intersect(self, ray):
        t = calcHitObj(ray, self)

        if t is None or t <= 0:
            return None

        P = intersectPt(ray, t)
        N = normalVec(self, P)
        return HitRecord(t, P, N, self)
```

5.2 camera.py

```
from ray import Ray
from util import *

WORLD_UP = create_vector(0, 1, 0)

def rotateAroundAxis(v, axis, angle):
    ax = normalize(axis)
    vParallel = ax * dot(ax, v)
    vPerp = v - vParallel

    vRot = vPerp * math.cos(angle) + cross(ax, vPerp) * math.sin(angle)
    return vParallel + vRot

class Camera:
    def __init__(self, pos, lookAt, imgWidth, imgHeight, focalLength):
        self.pos = pos # vector3d
        self.imgWidth = imgWidth # int
        self.imgHeight = imgHeight # int
        self.focalLength = focalLength # float
        self.aspectRatio = imgWidth / imgHeight #float

        self.forward = normalize(lookAt - pos)
        self.right = normalize(cross(self.forward, WORLD_UP))
        self.up = normalize(cross(self.right, self.forward))

    def createViewMatr(self):
        pos = self.pos
        f = self.forward
        r = self.right
        u = self.up

        V = np.array([
            [r[0], r[1], r[2], dot(-r, pos)],
            [u[0], u[1], u[2], dot(-u, pos)],
            [-f[0], -f[1], -f[2], dot(f, pos)],
            [0, 0, 0, 1]
        ])
        return V

    def createProjMatr(self):
        f = self.focalLength
        ar = self.aspectRatio
        far = 1000
        near = 0.1

        V = np.array([
            [f / ar, 0, 0, 0],
            [0, f, 0, 0],
            [0, 0, (far + near) / (near - far), 2 * far * near / (near - far)],
            [0, 0, -1, 0]
        ])
        return V

    def genRay(self, i, j): # ret: vector3d
        # clamping [0, 1]%
        u = (i + 0.5) / self.imgWidth
        v = (j + 0.5) / self.imgHeight

        # mapping [-1, 1]
        x = (2 * u - 1) * self.aspectRatio
        y = 1 - 2 * v

        cameraDir = normalize(create_vector(x, y, -self.focalLength))
        OCamMatr = np.array([
            0.0, 0.0, 0.0, 1.0
        ])
        DCamMatr = np.array([cameraDir[0], cameraDir[1], cameraDir[2], 0.0])

        V_inv = np.linalg.inv(self.createViewMatr())

        OWrlMatr = V_inv @ OCamMatr
        DWrldMatr = V_inv @ DCamMatr

        OWrld = create_vector(OWrlMatr[0], OWrlMatr[1], OWrlMatr[2])
        DWrld = normalize(create_vector(DWrldMatr[0], DWrldMatr[1], DWrldMatr[2]))
        return Ray(OWrld, DWrld)

    def moveDepth(self, amt):
        self.pos += self.forward * amt

    def moveHorizontal(self, amt):
        self.pos += self.right * amt

    def moveVertical(self, amt):
        self.pos += self.up * amt

    def yaw(self, angle):
        axis = WORLD_UP
        newForward = rotateAroundAxis(self.forward, axis, angle)
        newRight = rotateAroundAxis(self.right, axis, angle)
        self.forward = normalize(newForward)
        self.right = normalize(newRight)
        self.up = normalize(cross(newRight, newForward))

    def pitch(self, angle):
        axis = self.right
        newForward = rotateAroundAxis(self.forward, axis, angle)
        newUp = rotateAroundAxis(self.up, axis, angle)
        self.forward = normalize(newForward)
        self.right = normalize(cross(newForward, newUp))
        self.up = normalize(cross(self.right, newForward))
```

5.3 genImage.py

```
from camera import Camera
from render import *
from scene import Scene, makeSphere
from util import *


def buildScene():
    s1 = makeSphere(create_vector(-2, 0, -3), 1, 0.0)
    s2 = makeSphere(create_vector(2, 0, -3), 1, 1.0)
    return Scene([s1, s2])

def buildCamera():
    return Camera(ORIGIN, create_vector(0, 0, -1), DEFWIN_WIDTH, DEFWIN_HEIGHT, 1)

def main():
    camera = buildCamera()
    scene = buildScene()
    pixels = renderFrame(camera, scene, DEFWIN_WIDTH, DEFWIN_HEIGHT)

    writePPM("output", DEFWIN_WIDTH, DEFWIN_HEIGHT, pixels)

if __name__ == "__main__":
    main()
```

5.4 geometry.py

```
# calculations regarding 3d shapes
from dataclasses import dataclass

import numpy as np
```

```
from util import *

@dataclass
class HitRecord:
    t: float
    point: any
    normal: any
    obj: "Renderable"

class Renderable:
    def intersect(self, ray) -> HitRecord | None:
        raise NotImplementedError


# DEPRECATED
# def calcHit(ray, objList): # ret: (float, sphere) or (None, None)
#     t_hit = None
#     obj = None
#     for i in objList:
#         t = calcHitObj(ray, i)
#         if t is not None and (t_hit is None or t < t_hit):
#             t_hit = t
#             obj = i
#
#     return (t_hit, obj)

def calcHitObj(ray, sphere): # ret: float or None
    O = ray.origin
    D = ray.direction

    OMatr = np.array([O[0], O[1], O[2], 1.0])
    DMatr = np.array([D[0], D[1], D[2], 0.0])

    OLocMatr = sphere.M_inv @ OMatr
    DLocMatr = sphere.M_inv @ DMatr

    OLoc = OLocMatr[:3]
    DLoc = DLocMatr[:3]

    C = ORIGIN
    R = 1
    oc = OLoc - C

    a = dot(DLoc, DLoc)
    b = 2 * (dot(oc, DLoc))
    c = dot(oc, oc) - R ** 2
    discr = discriminant(a, b, c)
    if discr < 0:
        return None # no hit
    else:
        t1 = findRoots(a, b, c, 1)
        t2 = findRoots(a, b, c, -1)

        t_hit = None
        if t1 > 0: t_hit = t1
        if t2 > 0 and (t_hit is None or t2 < t_hit):
            t_hit = t2

        return t_hit

def intersectPt(ray, t_hit): # ret: vector3d or None
    if t_hit is None:
        return None
    O = ray.origin
    D = ray.direction
    P = O + t_hit * D

    return P

def normalVec(obj, PWrld):
    PWrldMatr = np.array([PWrld[0], PWrld[1], PWrld[2], 1.0])
    PLocMatr = obj.M_inv @ PWrldMatr
    PLoc = PLocMatr[:3]

    NLoc = normalize(PLoc)
    NLocMatr = np.array([NLoc[0], NLoc[1], NLoc[2], 0.0])
    NWrldMatr = obj.M @ NLocMatr
    NWrld = normalize(NWrldMatr[:3])

    return NWrld
```

5.5 live.py

```
# live camera!!
import numpy as np
import pygame

from camera import Camera
from render import *
from scene import Scene, makeSphere, projPt
from util import *
```

```
WIDTH = 320
HEIGHT = 180

# for wireframe stuff
CUBE_VERTICES = [
    (-1, -1, -1),
    (1, -1, -1),
    (1, 1, -1),
    (-1, 1, -1),
    (-1, -1, 1),
    (1, -1, 1),
    (1, 1, 1),
    (-1, 1, 1)
]

CUBE_EDGES = [
    (0, 1), (1, 2), (2, 3), (3, 0),
    (4, 5), (5, 6), (6, 7), (7, 4),
    (0, 4), (1, 5), (2, 6), (3, 7),
]

# camera speed, def = 3.0
SPEED = 3.0
# mouse sensitivity, def = 0.003
SENS = 0.003
# mouse center
CENTER_X, CENTER_Y = WIDTH // 2, HEIGHT // 2

def camInput(camera, seconds):
    keys = pygame.key.get_pressed()

    if keys[pygame.K_w]:
        camera.moveDepth(+SPEED * seconds)
    if keys[pygame.K_s]:
        camera.moveDepth(-SPEED * seconds)

    if keys[pygame.K_d]:
        camera.moveHorizontal(+SPEED * seconds)
    if keys[pygame.K_a]:
        camera.moveHorizontal(-SPEED * seconds)

    if keys[pygame.K_SPACE]:
        camera.moveVertical(+SPEED * seconds)
    if keys[pygame.K_LSHIFT]:
        camera.moveVertical(-SPEED * seconds)
    # disable "vectoring"
    # if length(move) > 0.0:
    #     move = normalize(move)
    #     camera.pos += move * SPEED * seconds

def mouseLook(camera):
    dx, dy = pygame.mouse.get_rel()

    camera.yaw(-dx * SENS)
    camera.pitch(-dy * SENS)

    pygame.mouse.set_pos(CENTER_X, CENTER_Y)

def buildScene():
    s1 = makeSphere(create_vector(-1.5, 0, -3), 1, 0.0)
    s2 = makeSphere(create_vector(1.5, 0, -3), 1, 1.0)
    return Scene([s1, s2])

def buildCamera():
    return Camera(ORIGIN, create_vector(0, 0, -1), WIDTH, HEIGHT, 1)

def wireframe(screen, camera):
    center = ORIGIN
    scale = 1.0

    projected = []
    for (x, y, z) in CUBE_VERTICES:
        wx = center[0] + x * scale
        wy = center[1] + y * scale
        wz = center[2] + z * scale
        PWrld = create_vector(wx, wy, wz)

        pos2D, _ = projPt(PWrld, camera, WIDTH, HEIGHT)
        projected.append(pos2D)
    for i, j in CUBE_EDGES:
        ax, ay = projected[i]
        bx, by = projected[j]
        pygame.draw.line(screen, (255, 255, 255), (ax, ay), (bx, by))

def main():
    pygame.init()
    pygame.mouse.set_visible(False)
    pygame.event.set_grab(True)
    pygame.mouse.set_pos(CENTER_X, CENTER_Y)
    pygame.mouse.get_rel()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    clock = pygame.time.Clock()

    scene = buildScene()
    camera = buildCamera()

    running = True
    while running:
        seconds = clock.tick(30) / 1000
        if seconds > 0:
            print("FPS:", 1 / seconds)

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        camInput(camera, seconds)
        mouseLook(camera)

        pixels = renderFrame(camera, scene, WIDTH, HEIGHT)
        byteBuf = writeBytes(pixels, WIDTH, HEIGHT)

        surf = pygame.image.frombuffer(byteBuf, (WIDTH, HEIGHT), "RGB")
        screen.blit(surf, (0, 0))

        wireframe(screen, camera)

        pygame.display.flip()

    pygame.quit()

if __name__ == "__main__":
    main()
```

5.6 ray.py

```
# r(t) = O + tD

class Ray:
    def __init__(self, origin, direction):
        self.origin = origin # vector3d
        self.direction = direction # unit vector3d

def at(self, t): # ret: vector3d
    return self.origin + t * self.direction # vector3d
```

5.7 render.py

```
# draw scene
from geometry import *
from ray import Ray
from util import *

SKYBOX = (125, 200, 255)
GROUND = (255, 255, 255)
LIGHT_POS = create_vector(5, 2, 0)

AMBIENT = 0.25
DIFFUSE = 1.0
SHINE = 20
DIFFUSE_COEF = 1.0
SPECULAR_COEF = 0.3
LIGHT_COLOR = (1.0, 1.0, 1.0)

DEPTH_MAX = 2
EPSILON = 1e-8

# for rendering the window, default sizes
DEFWIN_WIDTH = 1280
DEFWIN_HEIGHT = 720

def renderFrame(camera, scene, width, height):
    pixelBuf = [[0 for _ in range(width)] for _ in range(height)]
    for i in range(height):
        for j in range(width):
            ray = camera.genRay(j, i)
            pixelBuf[i][j] = traceRefl(ray, 0, scene, camera)
    return pixelBuf

def traceRefl(ray, depth, scene, camera):
    if depth > DEPTH_MAX:
        return skyGradient(ray)

    hit = scene.trace(ray)
    if hit is None:
        return skyGradient(ray)

    obj = hit.obj
    P = hit.point
    N = normalVec(obj, P)

    CLoc = lambertianShade(hit, camera)
    r = obj.reflectivity

    if r <= 0.0:
        return CLoc

    D = normalize(ray.direction)

    rDir = normalize(D - 2 * dot(D, N) * N)
    rOrig = P + EPSILON * rDir

    reflectRay = Ray(rOrig, rDir)

    CRefl = traceRefl(reflectRay, depth + 1, scene, camera)

    C_r = (1 - r) * CLoc[0] + r * CRefl[0]
    C_g = (1 - r) * CLoc[1] + r * CRefl[1]
    C_b = (1 - r) * CLoc[2] + r * CRefl[2]

    C_r = max(0, min(255, int(C_r)))
    C_g = max(0, min(255, int(C_g)))
    C_b = max(0, min(255, int(C_b)))

    C = (int(C_r), int(C_g), int(C_b))
    return C

# render skybox
def skyGradient(ray): # ret: (r, g, b)
    dir = normalize(ray.direction)
    y = dir[1]
    t = (y + 1) / 2  # vertical blending

    s_r = int((1 - t) * GROUND[0] + t * SKYBOX[0])
    s_g = int((1 - t) * GROUND[1] + t * SKYBOX[1])
    s_b = int((1 - t) * GROUND[2] + t * SKYBOX[2])

    return (s_r, s_g, s_b)

def colorHelper(n): # ret: float
    return (n + 1) / 2

# color normals
def colorSurfNorm(N): # ret: (r, g, b)
    c_r = int(colorHelper(N[0]) * 255)
    c_g = int(colorHelper(N[1]) * 255)
    c_b = int(colorHelper(N[2]) * 255)
    return (c_r, c_g, c_b)

# shadows and specular
def lambertianShade(hit, camera):
    P = hit.point
    N = hit.normal

    L = normalize(LIGHT_POS - P)
    diff = max(0, dot(N, L))

    C = camera.pos
    V = normalize(C - P)
    H = normalize(L + V)
    specAngle = max(0.0, dot(N, H))
    spec = specAngle ** SHINE

    base_r_255, base_g_255, base_b_255 = colorSurfNorm(N)

    base_r = base_r_255 / 255.0
    base_g = base_g_255 / 255.0
    base_b = base_b_255 / 255.0

    ambient_r = AMBIENT * base_r
    ambient_g = AMBIENT * base_g
    ambient_b = AMBIENT * base_b

    diffuse_r = DIFFUSE_COEF * diff * base_r
    diffuse_g = DIFFUSE_COEF * diff * base_g
    diffuse_b = DIFFUSE_COEF * diff * base_b

    specular_r = SPECULAR_COEF * spec * LIGHT_COLOR[0]
    specular_g = SPECULAR_COEF * spec * LIGHT_COLOR[1]
    specular_b = SPECULAR_COEF * spec * LIGHT_COLOR[2]

    color_r = ambient_r + diffuse_r + specular_r
    color_g = ambient_g + diffuse_g + specular_g
    color_b = ambient_b + diffuse_b + specular_b

    color_r = max(0.0, min(1.0, color_r))
    color_g = max(0.0, min(1.0, color_g))
    color_b = max(0.0, min(1.0, color_b))

    C_r = int(color_r * 255.0)
    C_g = int(color_g * 255.0)
    C_b = int(color_b * 255.0)

    # DEPRECATED
    # c_r = colorSurfNorm(N)[0]
    # c_g = colorSurfNorm(N)[1]
    # c_b = colorSurfNorm(N)[2]
    # C_r = int(c_r * brightness)
    # C_g = int(c_g * brightness)
    # C_b = int(c_b * brightness)
    #
    # # clamping
    # if C_r > 255: C_r = 255
    # elif C_r < 0: C_r = 0
    # if C_g > 255: C_g = 255
    # elif C_g < 0: C_g = 0
    # if C_b > 255: C_b = 255
    # elif C_b < 0: C_b = 0

    return (C_r, C_g, C_b)



def writeBytes(pixels, width, height):
    byteBuf = bytearray(width * height * 3)
    k = 0
    for i in range(height):
        for j in range(width):
            r, g, b = pixels[i][j]
            byteBuf[k] = r
            byteBuf[k + 1] = g
            byteBuf[k + 2] = b
            k += 3

    return byteBuf

# generate image file
def writePPM(filename, width, height, pixels):
    with open(filename + ".ppm", "w") as f:
        f.write(f"P3\n"
                f"{width} {height}\n"
                f"255\n")
        for i in range(height):
            for j in range(width):
                r, g, b = pixels[i][j]
                f.write(f"{r} {g} {b} \n")
```

5.8 scene.py

```
import camera
from objects.sphere import Sphere
from util import *


class Scene:
    def __init__(self, objList):
        self.objList = objList

    def trace(self, ray):
        closest = None
        for obj in self.objList:
            hit = obj.intersect(ray)
            if hit is not None and (closest is None or hit.t < closest.t):
                closest = hit

        return closest

def makeSphere(center, radius, reflectivity):
    S = scale(radius)
    T = translation(center[0], center[1], center[2])
    M = T @ S
    M_inv = np.linalg.inv(M)
    return Sphere(M, M_inv, reflectivity)

def projPt(PWrld, camera, width, height):
    V = camera.createViewMatr()
    P = camera.createProjMatr()

    PWrldMatr = np.array([PWrld[0], PWrld[1], PWrld[2], 1.0])

    PCam = V @ PWrldMatr
    PClip = P @ PCam

    #homogenous divide
    xndc = PClip[0] / PClip[3]
    yndc = PClip[1] / PClip[3]
    zndc = PClip[2] / PClip[3]

    screenX = (xndc + 1) * 0.5 * width
    screenY = (1 - yndc) * 0.5 * height

    return (screenX, screenY), zndc
```

5.9 util.py

```
# math stuff
# vectors are 3d
import math

import numpy as np


# vector stuff
ORIGIN = np.array([0.0, 0.0, 0.0])

def create_vector(x, y, z): # ret: vector3d
    return np.array([x, y, z], dtype=float)

def copy_vector(v): # ret: vector3d
    return create_vector(v[0], v[1], v[2])

def dot(v1, v2): # ret: vector3d
    return np.dot(v1, v2)

def cross(a, b): # ret: matrix
    return np.cross(a, b)

def length(v): # ret: float
    return np.linalg.norm(v)

def normalize(v): # ret: unit vector3d
    norm = length(v)
    v_u = v / norm
    return v_u

# algebra stuff
def discriminant(a, b, c): # ret: float
    return b ** 2 - (4 * a * c)

def findRoots(a, b, c, pm): # ret: float
    discr = discriminant(a, b, c)
    if pm >= 0:
        return (-b + math.sqrt(discr)) / (2 * a)
    else:
        return (-b - math.sqrt(discr)) / (2 * a)

# matrix stuff
```

```
def translation(tx, ty, tz):
    return np.array([
        [1, 0, 0, tx],
        [0, 1, 0, ty],
        [0, 0, 1, tz],
        [0, 0, 0, 1]
    ], dtype = float)

def scale(s):
    return np.array([
        [s, 0, 0, 0],
        [0, s, 0, 0],
        [0, 0, s, 0],
        [0, 0, 0, 1]
    ], dtype = float)
```