Course Project for Computer Graphics

Developer: Zhang Zichuan,
School of Software Engineering,
Tongji University

Duration: 2021.03-2021.04

Contents

Introduction

This is an introduction to the course project for Computer Graphics. The codes are available in my github repository: https://github.com/a389071432/CourseProjectCG. The purpose of doing this project was to help me acquire a better understanding of the knowledge and pipeline involved in computer rendering. Due to time constraints, only the main rendering workflow was implemented without caring about many details. The code directly related to rendering was written by myself, while several modules (including OBJ file parsing, TGA file reading and the model of a man's head used for demo) were from this repository: https://github.com/ssloy/tinyrenderer.

A purely software-based raster rendering demo and a ray-tracing rendering demo were implemented in C++. In the following sections, the relevant principles and technical details will be given.

1.Soft rasterizer

1.1 Rendering pipeline

For rendering the scene in a frame, the objects are shaded twice. The first shading process fills the ShadowBuffer, which serves as the basis for the shadow mapping of the second shading process; the second performs the usual shading, i.e., computing the RGB color.

In the first shading process, make the position and orientation of the camera coincide with the position and direction of illumination of the light source, respectively, and detect whether a fragment in a triangular mesh is sheltered by something. The results are recorded in shadow buffer:

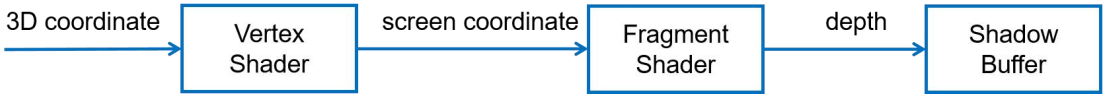

Figure 1.1: The first shading process.

In the first shading process, set the camera to where it should be, and handle all fragments of triangular meshes. Shadow mapping is included in fragment shader:

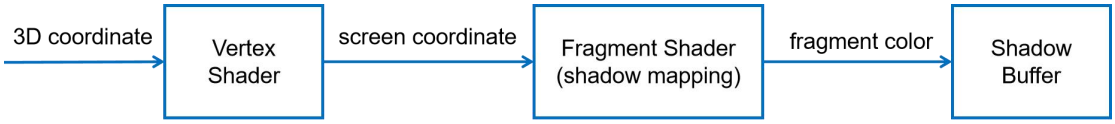

Figure 1.2: The second shading process.

1.2 Implementation for core modules

1.2.1 Camera

(Tips: The codes corresponding to the definition of *class Camera* can be found in file 'Camera.h' in my repository mentioned previously)

Pose

The pose of a camera is defined by three parameters: ***pos*** specifies where the camera is put, ***lookAt*** specifies where the camera is looking at, and a vector ***up*** specifies how the camera is set. They are used to define a local coordinate system (***D***, ***R***, ***U***) for the camera, where ***D, R, U*** are three orthogonal vectors determined by:

$$D = normalize\ (pos - lookAt)$$
$$R = normalize\ (up \times D)$$
$$U = normalize\ (D \times R)$$

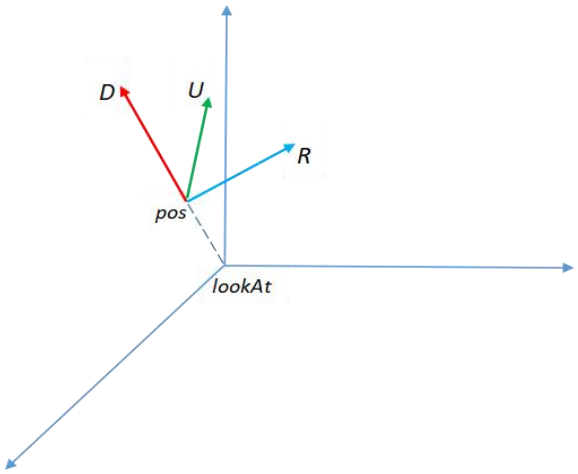

Figure 1.3: An illustration of how to define a camera's pose.

Projection

The projection parameters of the camera are defined by a viewing frustum, including the near plane ***n***, the far plane ***f***, the field of view ***FOV*** and the width to height ratio ***aspect***.

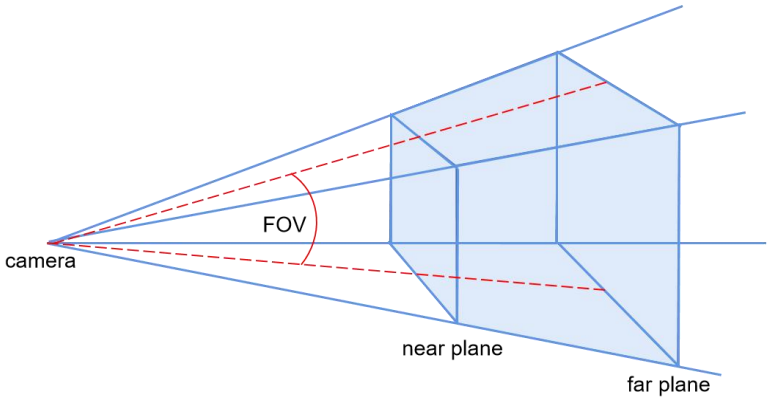

Figure 1.4: An illustration of the viewing frustum.

1.2.2 Light

(Tips: The codes corresponding to the definition of *class Light* can be found in file 'Light.h' in my repository mentioned previously.)

The basic properties of the light include intensity and RGB color. Two types of light sources are implemented, that is, the point light and the directional light. The point source has a property of position, and the directional light has a property of direction but no position.

1.2.3 Vertex shader

(Tips: The codes corresponding to the vertex shader can be found in the function *'Shader.cpp/shade_a_vertex'* , which performs a set of transformation on the vertexes' coodinations)

The vertex shader is responsible for transforming the vertices of triangular meshes from an object's local coordinate space to the screen space and recording the intermediate results at each stage of transformation, which will be used in the fragment shader. Figure 1.5 shows the workflow of the vertex shader, a detailed description of each transformation will be given.

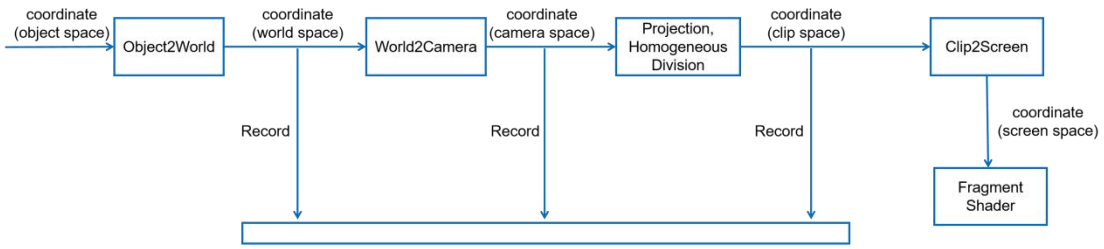

Figure 1.5: Workflow of the vertex shader.

Transformation: World2Camera

The transformation of a point from the world space to camera space is performed by a matrix:

$$World\ \ 2Camera\ = \begin{bmatrix} Rx & Ry & Rz & 0 \\ Ux & Uy & Uz & 0 \\ Dx & Dy & Dz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -Cx \\ 0 & 1 & 0 & -Cy \\ 0 & 0 & 1 & -Cz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where ***(R,U,D)*** are vectors defining the pose of the camera(as described in section 1.2.1), and ***(Cx,Cy,Cz)*** is the position of the camera(in world space).

Transformation: Camera2Clip

The transformation of a point from camera space to clip space consist of two steps. The first step is performed by a projection matrix:

$$Projection = \begin{bmatrix} \frac{cot(FOV/2)}{aspect} & 0 & 0 & 0 \\ 0 & cot(FOV/2) & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where n,f, aspect and FOV are parameters used to define a viewing frustum(as described in section 1.2.1).

The second step is to perform the homogeneous division on the result obtained from step 1:

$$X' = \frac{1}{w} X = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$

The resulting coordinate describes both a vertex's relative position in the near plane and the relative position in the viewing frustum. If any item of the resulting coordinate (i.e., *x,y,z*) is beyond the range of (-1,1), the vertex will be rejected as this indicates that it lies outside the viewing frustum.

Transformation: Clip2Screen

The transformation of a point from the clip space to the screen space is performed by a matrix:

$$Clip\,2Screen = \begin{bmatrix} W/2 & 0 & 0 & W/2 \\ 0 & H/2 & 0 & H/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where W and H are the width and height of the screen in pixels, respectively. After this, a vertex in the 3D world space is projected to a pixel on the screen.

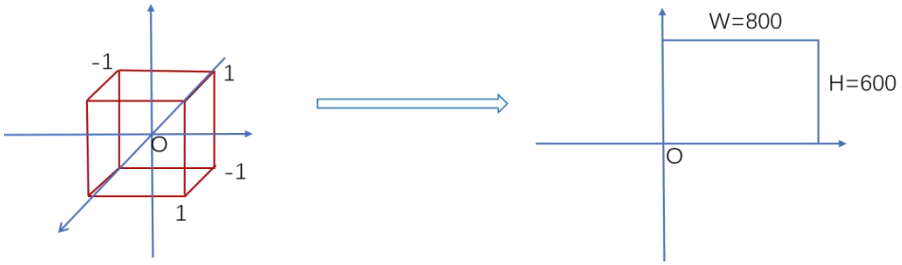

Figure 1.6: An illustration of the Transformation from the cip space to screen space.

1.2.3 Fragment shader

(Tips: The codes corresponding to the fragment shader can be found in the function 'Shader.cpp/shade_a_triangle' and 'Shader.cpp/shade_a_fragment')

The fragment shader uses the output of the vertex shader, various types of mapping information and the illumination model to determine the color of each fragment in triangular meshes. Depth testing and shadow mapping are also included. The workflow of the fragment shader is shown in Figure 1.7. Detailed description of the main steps will be given.

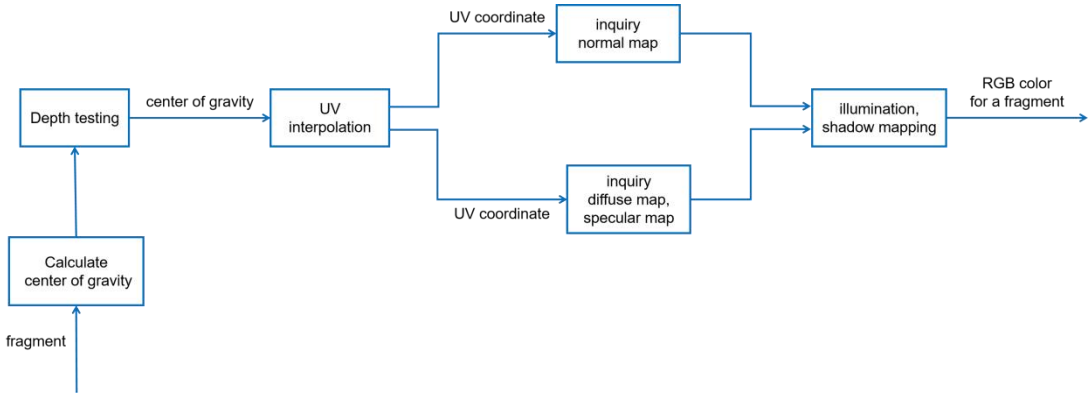

Figure 1.7: Workflow of the fragment shader.

Calculating the center of gravity

The center of gravity defines a fragment's position relative to the three vertices of a triangle, and is used to get the UV coordinate of that fragment by interpolation. Then, information about that fragment can be located from various types of maps (e.g., normal map, diffuse map, etc.) using its

UV coordinate.

There is a non-linear process (i.e.,the homogenous division) in the transformation of the fragment from 3D world space to screen space, so the 2D center of gravity coordinates cannot be used for interpolation. The 2D center of gravity of the fragment $(\alpha', \beta', \gamma')$ needs to be corrected to obtain the 3D center of gravity (α, β, γ). Here gives the derivation from 2D to 3D center of gravity coordinates.

Let the three vertices of a given triangle have 2D coordinates (P_A', P_B', P_C') in screen space and 3D coordinates (P_A, P_B, P_C) in world space respectively. Let the coordinates of a fragment in the triangle in two spaces be P' and P respectively. Then P' can be represented by:

$$P' = \alpha' P'_A + \beta' P'_B + \gamma' P'_C \quad \text{(Eq 1.1)}$$

As described in section 1.2.3, the transformation of a point from the world space to the screen space can be performed by a set of matrixs (whose product is denoted by $\boldsymbol{T}$) and homogenous division, then we get:

$$P' = \frac{TP}{Z},\ P_A' = \frac{TP_A}{Z_A},\ P_B' = \frac{TP_B}{Z_B},\ P_C' = \frac{TP_C}{Z_C} \quad \text{(Eq 1.2)}$$

Substituting equation 1.2 into equation 1.1, we get:

$$\frac{TP}{Z} = \frac{\alpha' TP_A}{Z_A} + \frac{\beta' TP_B}{Z_B} + \frac{\gamma' TP_C}{Z_C} \quad \text{(Eq 1.3)}$$

Then, we get:

$$P = \frac{\alpha' ZP_A}{Z_A} + \frac{\beta' ZP_B}{Z_B} + \frac{\gamma' ZP_C}{Z_C} \quad \text{(Eq 1.4)}$$

Where, Z_A, Z_B and Z_C are the Z coordinates of vertex A,B,C in camera space respectively. P can be written as:

$$P = \alpha P_A + \beta P_B + \gamma P_C \quad \text{(Eq 1.5)}$$

Comparing equation 1.4 and 1.5, we get the final result:

$$\alpha = \frac{Z\alpha'}{Z_A}, \beta = \frac{Z\beta'}{Z_B}, \gamma = \frac{Z\gamma'}{Z_C}$$

Depth testing

To obtain the Z coordinate of a given fragment in clip space, interpolate the Z coordinates of the three vertices of a triangle in clip space using its 2D center of gravity. The Z coordinate of the fragment is then used for clipping testing and depth testing.The painter's algorithm is used for depth testing, whose result is independent of the processing order of the triangles in the scene.

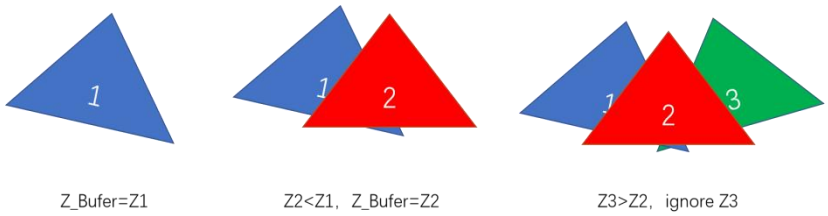

Figure 1.8: An illustration of the painter's algorithm when processing three triangles.

Get the normal for a fragment

Normal maps record the normal vector of each point on the surface of an object, which provides details for illumination. For a fragment in a triangular mesh, its normal vector is recorded at somewhere in the normal map located by its UV coordinates. For a fragment in a triangular mesh, its normal vector is recorded somewhere in the normal map located by its UV coordinates.

The normal map is defined in the tangent space, which is a local space defined by the three vertexes of the triangle. Figure 1.9 is an illustration for this.

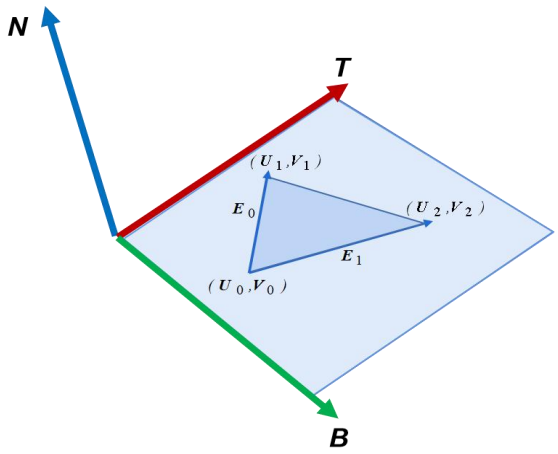

Figure 1.9: Tangent space. The light blue plane represents the normal map for an object, and the part in deep blue represents a triangle mesh of the object.

As shown in Figure 1.9, *T, B, N* are the basis vectors of the tangent space determined by the three vertexes of the triangle, and (U_0,V_0) , (U_1,V_1) and (U_2,V_2) are the coordinates of the three vertexes respectively.

In order to transform the normal vector from tangent space to world space, a transformation matrix (denoted as *TBN*)is needed:

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} U_1 - U_0 & V_1 - V_0 \\ U_2 - U_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} E_{0x} & E_{0y} & E_{0z} \\ E_{1x} & E_{1y} & E_{1z} \end{bmatrix}$$

$$N = B \times T$$

Once the transformation matrix TBN is obtained, the normal vectors from the normal map can

be transformed into the world space through the method shown in Figure 1.10.

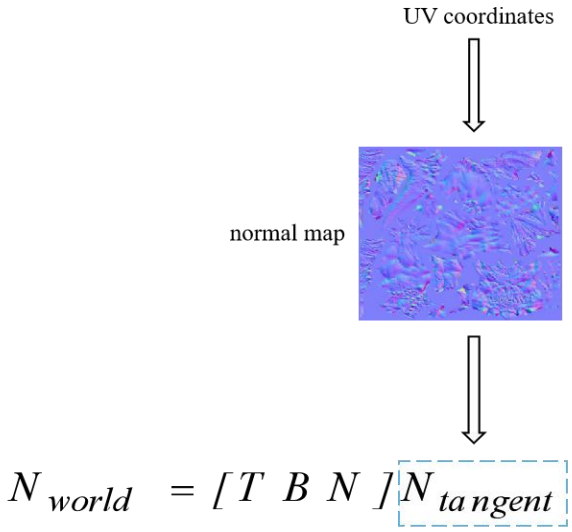

Figure 1.10: Transformation of the normal vector from a normal map into world space.

Illustration

Use the Phong model to determine the color of each pixel on the screen. In the Phone model, the light received by the camera/screen consists of three distinct parts, that is, diffuse, specular and ambient.Figure 1.11 is an illustration of diffuse and specular.

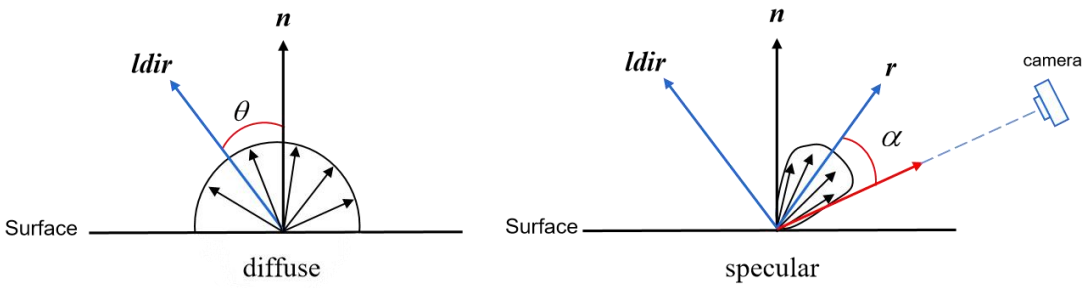

Figure 1.11: diffure reflection and specular reflection in Phong model.

In the Figure 1.11, ***ldir*** is the direction of light, ***n*** is the normal vector of the surface of an object(or a fragment), ***r*** is the reflection vector of the light, and ***v*** denotes the view vector that points at the camera(or screen). Denote the incident light intensity as I_0 , and denote the parameter for specular as ***k*** (obtained from specular mapping), then the intensity of three parts of illumination are calculated as:

$$I\_diffuse = I_0 max(0, \cos\theta)$$

$$I\_specular = I_0 max(0, \cos^k \alpha)$$

$$Ambient = const$$

For both diffusion and specular, intensity of light decays as the angle between the target vector

and the normal vector increases, while the intensity of specular reflection decays much faster as it decays exponentially. Ambient is set to be a constant.

Then, the color of a fragment showing in the corresponding pixel of the screen is computed as:

$$color = (I\_diffuse + I\_specular) \cdot diffuse + Ambient$$

Where *diffuse* is the self-luminous color of an object(or a fragment) obtained from texture mapping. The effect of each of the three parts and their combined effect is shown in Figure 1.12.

Figure 1.12: An illustration of the Phong model.

Shadow mapping

After the basic color values for each fragment have been calculated, shadow mapping is performed to darken the occluded fragments.Shadow mapping is performed by transforming a fragment's coordinate in the world space into the clip space(corresponding to the pseudo camera used only for shadow mapping), checking whether there is an occlusion by comparing its Z-value with the Z-value of the corresponding pixel recorded in the ShadowBuffer. If there is an occlusion, lower the fragment's color by a proportion(here set to a constant of 0.3).

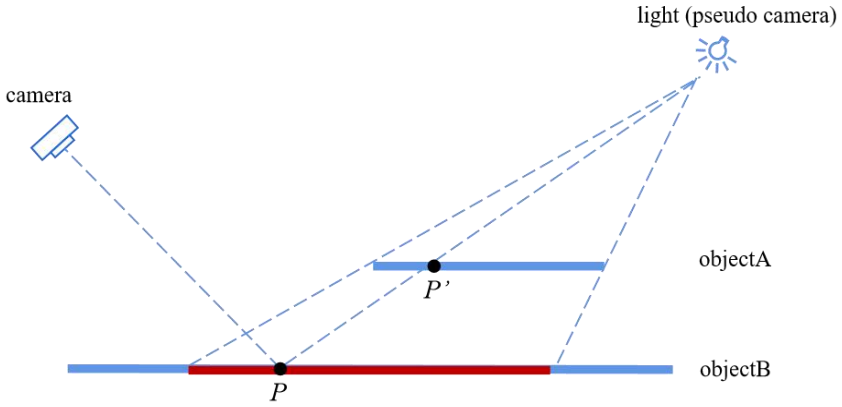

Figure 1.13: Shadow mapping. The red part of object B is occluded by object A, and *P* is a fragment within it. *P* and fragment *P'* are projected to the same pixel in the screen of the pseudo camera, and the Z-value of *P* is greater than that of *P'*. Therefore, fragment *P* is darkened.

Figure 1.14: An illustration of shadow mapping. The left image shows the Z-values (depth) recorded in shadow buffer viewing from the pseudo camera.

1.3 Rendering results

Figure 1.15 shows the rendering results with the light being set in different directions.

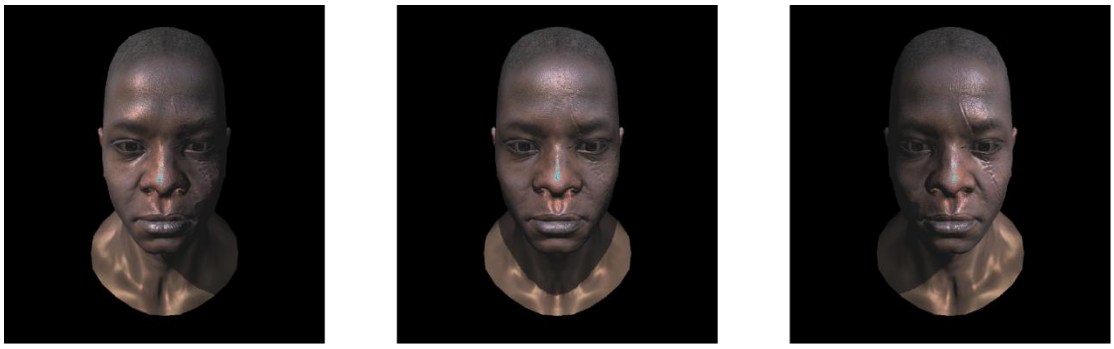

Figure 1.15: Rendering results.

2 Soft raytracer

2.1 Workflow

(Tips: The codes corresponding to the main workflow of raytracer can be found in the function 'main.cpp/RayTraceForOnePixel')

At the beginning, cast a ray to each pixel in the screen starting at the center of the camera, and thereafter the following steps are performed recursively:

1. **Detection of intersection**. Detect the first intersection point of the ray with all objects in the scene. Denote the current ray as ***L***, the reflection direction of ***L*** as ***RL***, the intersection as ***A***, and the corresponding object as ***obj***.

2. **Local illumination**. If the ***A*** is not inside the object and there is no occlusion between point ***A*** and the light, then calculate the local illumination at ***A*** using all lights in the scene.

3. **Trace on the reflection path**. If ***obj*** is attached with a material of diffuse reflection, then continue tracing in a direction randomly selected on the hemisphere centered on ***RL***. If ***obj*** is attached with a material of specular reflection, simply continue tracing in the direction of ***RL***.

4. **Trace on the refraction path**. If no total reflection is occurring, continue tracing in the refraction path.

5. **Return tracing result**. Denote the RGB value obtained from local illumination as ***il_local***, the result of tracing on the reflection path as ***il_reflect***, the result of tracing on the refraction path as ***il_refract***, and the self-luminous color at the ***A*** as diffuse, then the final result returned to the last level of tracing is ***il_local+diffuse (il_reflact+il_refract)***.

Figure 2.1 is an illustration of the workflow.

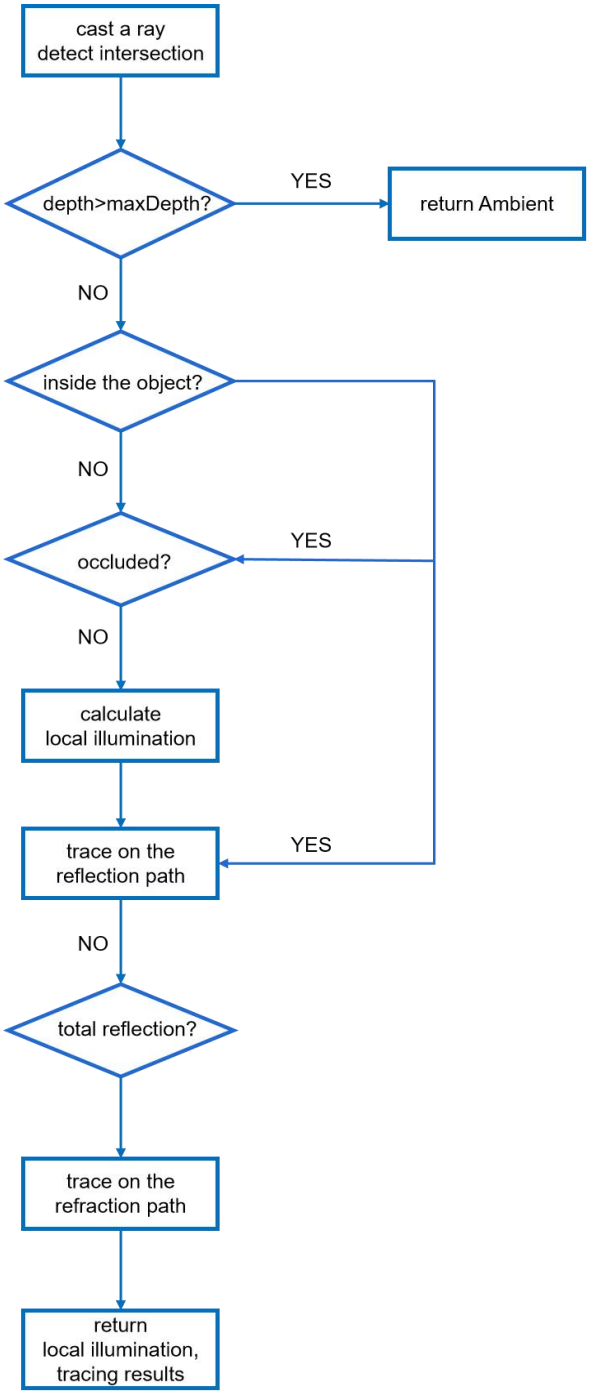

Figure 2.1: Workflow of the raytracer.

2.2 Initial casting

An initial ray is casted to each pixel (x,y) on the screen for tracing. To represent the direction of the light in camera space, we need to map the pixel (x,y) to a point (X,Y) on a virtual plane:

$$X = \frac{2n\tan(Fov/2) * aspect * (x - W/2)}{W}$$

$$Y = \frac{2n\tan(Fov/2)*(y - H/2)}{H}$$

In order to improve the accuracy of tracing results, each pixel on the screen is subdivided into several virtual grids, and the final color of the pixel is the average of the tracing results of these grids.

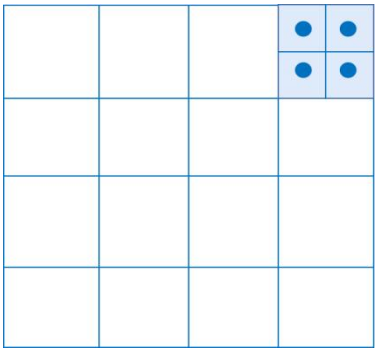

Figure 2.2: Subdivide a pixel into grids for higher tracing accuary.

2.3 Interaction of rays and objects

(Tips: The codes corresponding to the detection of the intersection point can be found in the virtual function 'FirstIntersectPoint' in each type of object)

Sphere

Let the center of a sphere be $\boldsymbol{C}$, its radius be $\boldsymbol{R}$, and a point on its surface be $\boldsymbol{X}$. Then the parametric equation corresponding to the spherical surface is:

$$\|X - C\|^2 = R^2 \qquad \text{(Eq 2.1)}$$

Let the point of incidence of the ray be P, and the direction be d. To get the intersection of the ray with the sphere, substitute the parametric equation of the ray into equation 2.1, we get:

$$\|\boldsymbol{P} + t\boldsymbol{d} - \boldsymbol{C}\|^2 = R^2$$

$$\|\boldsymbol{d}\|^2 t^2 + 2\boldsymbol{t}[\boldsymbol{d}\cdot(\boldsymbol{P}-\boldsymbol{C})] + (\boldsymbol{P}-\boldsymbol{C})\cdot(\boldsymbol{P}-\boldsymbol{C}) - \boldsymbol{R}^2 = 0 \qquad \text{(Eq 2.2)}$$

Eq. 2.2 is a quadratic equation which is solved to obtain the parameter t:

$$\boldsymbol{t} = \frac{-\boldsymbol{d}\cdot(\boldsymbol{P}-\boldsymbol{C}) \pm \sqrt{[\boldsymbol{d}\cdot(\boldsymbol{P}-\boldsymbol{C})] + (\boldsymbol{P}-\boldsymbol{C})\cdot(\boldsymbol{P}-\boldsymbol{C}) - \boldsymbol{R}^2}}{\|\boldsymbol{d}\|^2} \qquad \text{(Eq 2.3)}$$

Infinite plane

Let X_0 be a known point in the plane , and N be the normal vector of the plane, then the parametric equation is:

$$N \times (X - X_0) = 0 \qquad \text{(Eq 2.4)}$$

Substitute $X = P + tD$ into Eq 2.4:

$$t = \frac{N \cdot (X_0 - P)}{N \cdot D} \quad \text{(Eq 2.5)}$$

Triangle

Triangle is used as the basic unit for representing complicated meshes in computer graphics, so it is necessary to specify how to get the intersection of a ray and a triangle. Based on the infinite plane case, the detection of ray and triangle intersections can be easily implemented.

Firstly, detect the interaction of the ray and the infinite plane in which the triangle lies using the method mentioned previously, and the intersection is denoted as *P*.

Secondly, check whether point P is inside the triangle using the following condition:

$$\begin{cases} (\overrightarrow{PA} \times \overrightarrow{PB}) \cdot (\overrightarrow{PB} \times \overrightarrow{PC}) > 0 \\ (\overrightarrow{PA} \times \overrightarrow{PB}) \cdot (\overrightarrow{PC} \times \overrightarrow{PA}) > 0 \\ (\overrightarrow{PB} \times \overrightarrow{PC}) \cdot (\overrightarrow{PC} \times \overrightarrow{PA}) > 0 \end{cases}$$

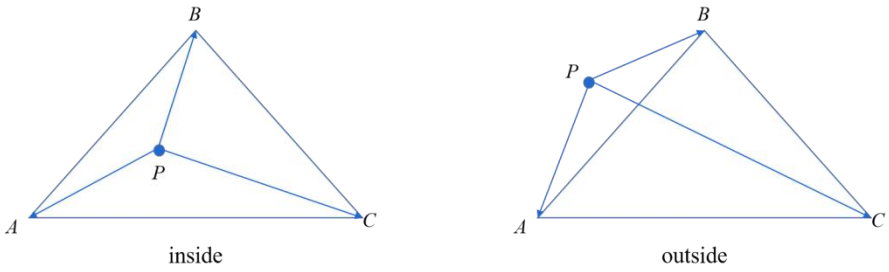

Figure 2.3: Check whether the intersection point is inside the triangle using its relative position to the three vertexes.

2.4 Path tracing

Reflection

Let ***i*** be the inverse direction of the ray, and *P* be the intersection point of the ray with an object, then the reflection ***r*** can be computed by equation 2.6:

$$r = 2(i \cdot N)N - i \quad \text{(Eq 2.6)}$$

Refraction

Let n_1, n_2 be the refractive indexes of medium A and medium B respectively, then the refraction ***r*** can be computed by equation 2.7:

$$r = \frac{n_1}{n_2} i - \left(\frac{n_1}{n_2}(i \cdot N) + \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - (i \cdot N)^2)}\right) N \quad \text{(Eq 2.7)}$$

When the ray enters a light rarer medium from a light denser medium, there is a critical angle θ_c . When the angle of incidence is greater than θ_c , only reflection and not refraction occurs (i.e. total reflection). The critial angle is computed as:

$$\theta_c = arcsin\ (\frac{n_1}{n_2})$$

Where n_1 is the refractive index of the light rarer medium, n_2 is the refractive index of the light denser medium.

2.5 Rendering results

A scene with two point sources, five infinite planes, and two spheres is constructed to test the rendering effect. All planes are in specular reflective materials. The left sphere is in a specular reflection material and the right one is in a refraction material. Set the sampling rate to 16 (i.e., each pixel is subdivided into 16 grids) and the maximum tracing depth to 6. The rendering results are shown in Figure 2.4.

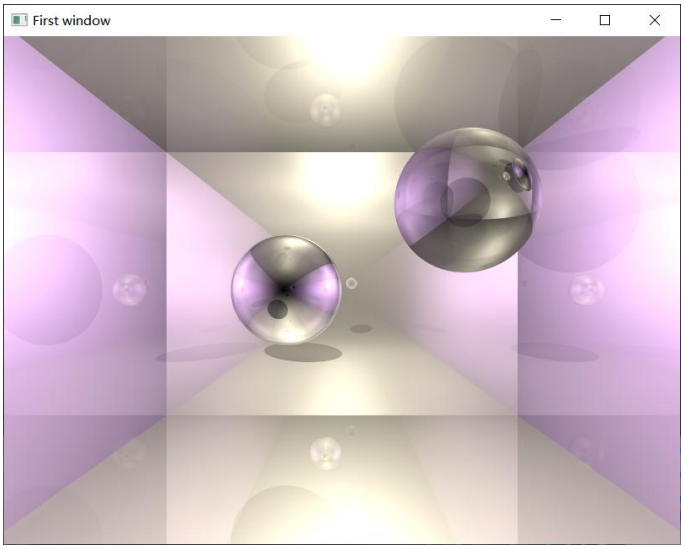

Figure 2.4: The rendering result with a sample rate of 36 and the maximum tracing depth of 6.

Replace all planes with the ones in a diffusion material, and render the scene with a sample rate of 1, 4 and 16 respectively. The rendering results are shown in Figure 2.5. It is clear that a higher sample rate leads to better quality.

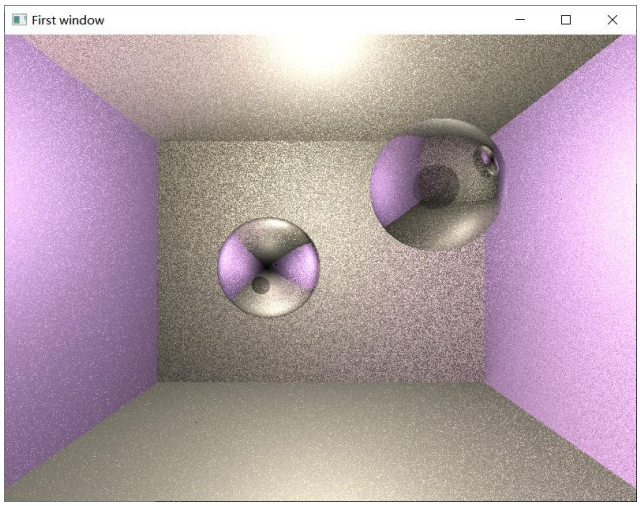

Figure 2.5(a): sample=1

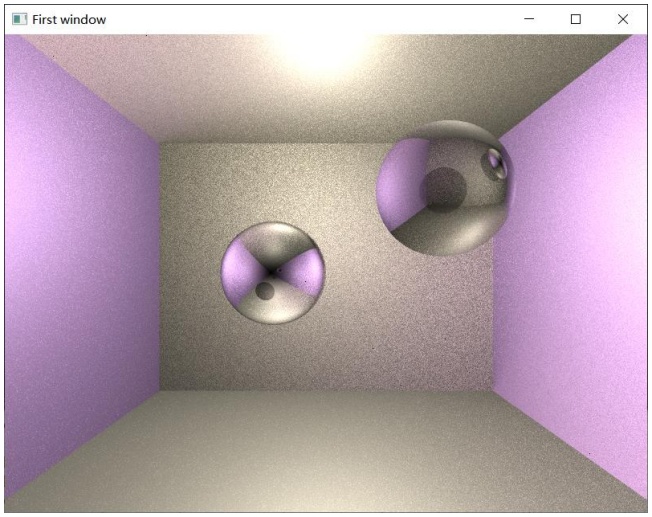

Figure 2.5(b): sample=2

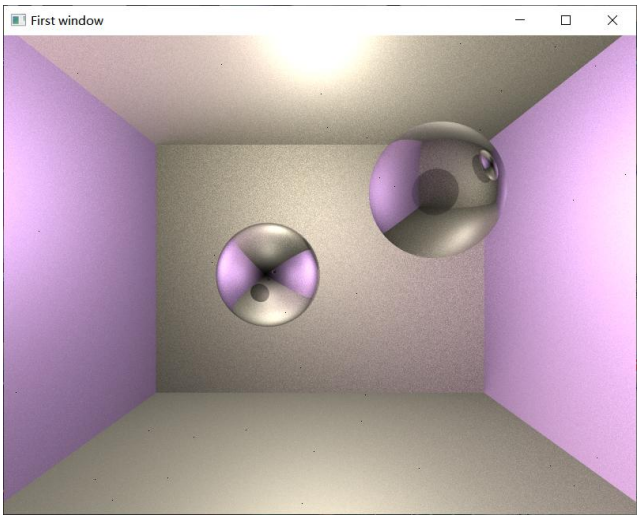

Figure 2.5(c): sample=4

Figure 2.5: Rendering results with different sample rates.

Reduce the maximum tracing depth from 6 to 3, the comparison between the rendering result and the previous one is shown in Figure 2.6. We can see that several details get lost due to insufficient tracing depth.

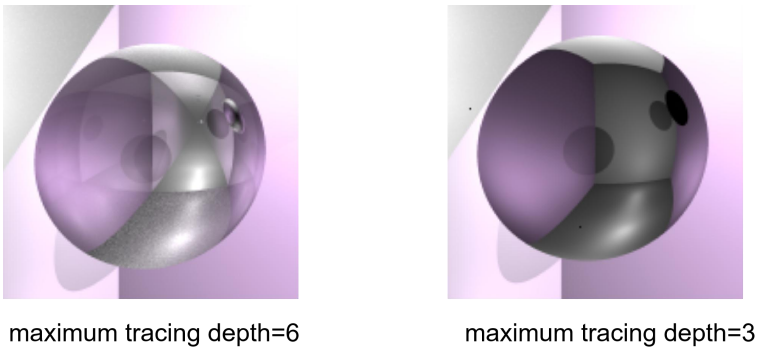

Figure 2.6: Rendering results with different maximum tracing depth.