Assignment 3: Ray Tracing Basics

NAME: HIDDEN
STUDENT NUMBER: HIDDEN
EMAIL: HIDDEN@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment, the following 'must' tasks are completed:

- Implement a pin-hole camera model, which is able to shoot optical rays from the camera to the scene.
- Implement ray-geometry intersection test, which is able to determine whether a ray intersects with a geometry, including triangle, rectangle and ellipsoid.
- Implement light ray sampling with soft shadow, and apply phong lighting model at intersection points.
- Implement anti-aliasing using super sampling with rotated grid.

And the following 'bonus' tasks are done:

- Texture mapping.
- Normal texture mapping.
- Advanced anti-aliasing with the help of Halton sequence.

2 IMPLEMENTATION DETAILS

2.1 Camera Model

The camera model is defined by the following parameters:

- Camera position: pos.
- Camera field of view: fov.
- Focal length: fixed to 1 in this assignment.
- Forward, Up and Right direction vectors: forward, up and right.
- Image resolution and aspect ratio: image.

Each time the look_at position or camera position is updated, function Camera::lookAt will be called to update 'Forward', 'Up' and 'Right'.

$$\begin{cases} forward = (look\_at - pos).normalized() \\ right = (forward \times up).normalized() \\ up = (right \times forward).normalized() \end{cases}$$

Then we need to implement the 'generate ray' function. We need to generate a ray according to the screen coordinate (dx, dy).

To shoot a ray from the camera to the scene, we need to first determine the position of the pixel on the image plane. So we calculate the ratio of d_x, d_y relatively to the whole image. The leftmost pixel gets x ratio -1, rightmost 1, and the other linearly distributed. The code is:

```
float ratio_x = (dx / width) * 2 - 1;
float ratio_y = (dy / height) * 2 - 1;
```

Then consider fov and calculate how much the camera needs to biase to the upper and right. The pseudocode is:

```
float half_up = tan(fov / 2) * focal_len;
float half_right = half_up * aspect_ratio;
```

The ray direction is a normalized vector starting from screen center and pointing to the pixel position. The pseudocode is:

```
screen_center = cam_pos + forward * focal_len;
pixel = screen_center
        + right * ratio_x * half_right
        + up * ratio_y * half_up;
ray_dir = (screen_pos - cam_pos).normalized();
```

Then we return a ray with the origin at camera position and the direction calculated above.

2.2 Ray-Geometry Intersection

We need to determine whether a ray intersects with a geometry, including triangle, rectangle and ellipsoid. The ray-geometry intersection test is implemented in `geometry.cpp`.

- **Ray-triangle intersection**
  Suppose the triangle's three vertices are p_0, p_1, p_2. Any point inside the triangle can be written as

  $$p(b_1, b_2) = (1 - b_1 - b_2)p_0 + b_1 p_1 + b_2 p_2$$

  with conditions $b_1 \geq 0, b_2 \geq 0$ and $b_1 + b_2 \leq 1$.
  To test whether a ray intersects with the triangle, we simply need to solve the following equation:

  $$o + td = (1 - b_1 - b_2)p_0 + b_1 p_1 + b_2 p_2$$

  The solution of the above equation is

  $$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{s_1 \cdot e_1} \begin{bmatrix} s_2 \cdot e_2 \\ s_1 \cdot s \\ s_2 \cdot e \end{bmatrix}$$

  where

  $$\begin{cases} e_1 = v_1 - v_0 \\ e_2 = v_2 - v_0 \\ s = o - v_0 \\ s_1 = d \times e_2 \\ s_2 = s \times e_1 \end{cases}$$

  Check if the t, b_1 and b_2 meets the requirements, and we can determine whether the ray intersects with the triangle.
- **Ray-rectangle intersection**
  Suppose the rectangle plane is defined by $(P - P_0)\overrightarrow{n} = 0$.
  To determine intersection, we solve the following equation:

  $$(o + td - P_0)\overrightarrow{n} = 0$$

The solution of t is

$$t = \frac{\overrightarrow{n} \cdot (P_0 - o)}{\overrightarrow{n} \cdot d}$$

Next, we need to judge whether the point is inside the rectangle. We compute the dot product between $P - P_0$ and the rectangle's 'tangent' and 'cotangent' ('cotangent' is given by the cross product of 'normal' and 'tangent').

If the absolute value of the dot product is less than half of the rectangle's width or height, the point is inside the rectangle.

- **Ray-ellipsoid intersection**

Since it would be too complicated to solve the intersection of a ray and an ellipsoid, we first transform the ellipsoid into a unit sphere, solving the intersection with the sphere, and then transform the interaction back.

To transform from sphere to ellipsoid, we need the matrix M defined by:

$$T = \begin{bmatrix} 1 & 0 & 0 & C_x \\ 0 & 1 & 0 & C_y \\ 0 & 0 & 1 & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, R = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, S = \begin{bmatrix} \|a\| & 0 & 0 & 0 \\ 0 & \|b\| & 0 & 0 \\ 0 & 0 & \|c\| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$M = TRS$$

where C_x, C_y, C_z is the center of the sphere, and a, b, c are normalized value.

Then we compute the inverse of M, M^{-1}, and transform the ray from the ellipsoid to the sphere:

$$\begin{cases} o' = M^{-1}o \\ d' = M^{-1}d \end{cases}$$

Then we solve the intersection of the ray and the sphere. The discriminant is given by

$$\Delta = d' \cdot d' - 4(o' \cdot d') + 4(o' \cdot o' - 1)$$

If $\Delta < 0$, there is no intersection. Otherwise, there're one or two intersections, given by

$$\begin{cases} t_1 = \frac{-d' + \sqrt{\Delta}}{2} \\ t_2 = \frac{-d' - \sqrt{\Delta}}{2} \end{cases}$$

We take the smaller one as the intersection point, $\min(t_1, t_2)$, check if it is within $[t_min, t_max]$ range, and transform the intersection point back to the ellipsoid by multipling the matrix M:

$$p = M(o + td)$$

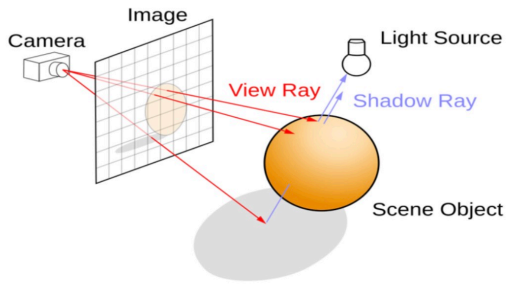
2.3 Light Ray Sampling

In this scene, the light is a square-area light, using rectangle geometry.

To check if a ray intersects with the light, we simply need to check if the ray intersects with the rectangle, and modify the interaction type to `Light` if it does.

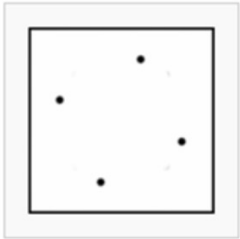
Then we need to sample light rays from the light source to the intersection point. We generate rays using a uniform grid inside the rectangle. The light sample contains the light's color (which is white in this scene), and the sample position inside the area.

Multiple samples are generated for each light source to allow for soft shadows. We will use `sample_num = 16` to generate 256 light samples for this scene.



2.4 Anti-Aliasing by super sampling

To reduce aliasing, we need to generate multiple rays in one pixel and take the average color of them. One of the best ways is the rotated grid super sampling.



Like in the above picture, we generate 4 rays in one pixel rotated an angle of $\arctan(0.5) \approx 26.6$.

2.5 Scene intersection and shadowing

To check if a ray intersects with a scene, we traverse all geometries and check if the ray intersects. We pick the intersection with shortest t value.

To check whether a ray is shadowed is easier. If a ray has any geometry intersection, it is shadowed.

2.6 Texture Mapping

We will apply texture mapping to rectangle geometry to make the scene more realistic. A texture is a `jpg` image.

First, we read in the color of each pixel in the texture. We store the pixel color data and the image resolution in `class Texture`.

Then, in ray-geometry intersection, we need to store `uv` value additionally. In rectangle intersection, after computing the dot product values between $P - P_0$ and the rectangle's 'tangent' and 'cotangent', we compute the `uv` value by dividing the dot product by the rectangle's width and height. In other words, we compute the ratio of the distance between P and P_0 to the rectangle's width and height. The pseudocode is given below:

```
float dot1 = (p - position).dot(tangent);
float dot2 = (p - position).dot(cotangent);
// Process intersection here ...
float u = clamp01((dot1 / (width / 2) + 1) / 2);
```

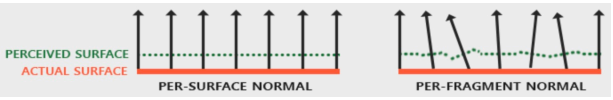
```
float v = clamp01((dot2 / (height / 2) + 1) / 2);
```

After that, instead of `Const Color Material` that returns a constant phong-lighting-model color, we use `Texture Material` that first get the color of the texture at uv, and then returns the phong-lighting-model by that color. The pixel id is computed by `u * width` and `v * height`. The shininess value keeps the same, and the remaining process keeps the same.

In this way, we can see a textured rectangle.

2.7 Normal Texture Mapping

In the above section, we applied texture mapping to the rectangle geometry. However, the normal vector of the surface kept the same as the constant value of the rectangle, making it looks completely flat and too smooth. In this section, we will modify the normal value after applying texture mapping to make the surface more realistic.



The normal value of the texture is stored in another picture. However, this is actually not a picture. Its RGB values are representing normal vectors, not colors. The picture have a blue-ish tint because most of the the normals are all closely pointing outwards towards the positive z-axis (0,0,1).

To apply normal texture mapping, we need to store two textures in the texture material class. One is the color texture, and the other is the normal texture. We also need to add another member function to the texture class, `getNormal(uv)`, to get the normal value of the texture. The value needs to be rounded to $[-1, 1]$, so before it returns the RGB value (in range $[0, 1]$), it multiplies the value by 2 and then subtract 1. The pseudocode is

```
Vec3f getNormal(float u, float v) {
  float R, G, B = getColor(u, v);
  return {R * 2 - 1, G * 2 - 1, B * 2 - 1};
}
```

However, this is not the normal we would expect. The normal value is correct only when the rectangle is lying on the ground, facing upwards. Therefore we need to apply a transformation by a 3x3 matrix TBN. T means 'tangent', B means 'bitangent (cotangent)' and N means 'normal'. In pseudocode:

```
// In ray-rectangle intersection ...
interaction.TBN = Mat3(tangent,cotangent,normal);

// In texture evaluation ...
Vec3f normal = texture.getNormal(u, v);
interaction.normal = interaction.TBN * normal;
```

In this ray, we can have a normal texture at almost no additional cost, since other than the normal vector, nothing else (including intersection calculation, phong lighting model) are changed.

2.8 Advanced Anti-Aliasing by Halton Sequence

In this section, we will apply anti-aliasing by halton sequence. The halton sequence is a sequence of points with low discrepancy, that is, looks as if it is randomly distributed in the unit square. The halton sequence is generated by the following formula:

$$h_i = \sum_{j=1}^{i} \frac{1}{p^j} \bmod 1 \tag{1}$$

where p is a prime number, and i is the index of the point. The halton sequence is generated by the following pseudocode:

```
float halton(int index, int base) {
  float result = 0;
  float f = 1.0f / base;
  int i = index;
  while (i > 0) {
    result = result + f * (i % base);
    i = floor(i / base);
    f = f / base;
  }
  return result;
}
```

In this way, we can generate a sequence of points distributed in a unit square. We can use these points to generate rays and sample the pixels in the image. Since a pixel may have multiple ray samples, the pixel color is the average of them. The pseudocode is given below:

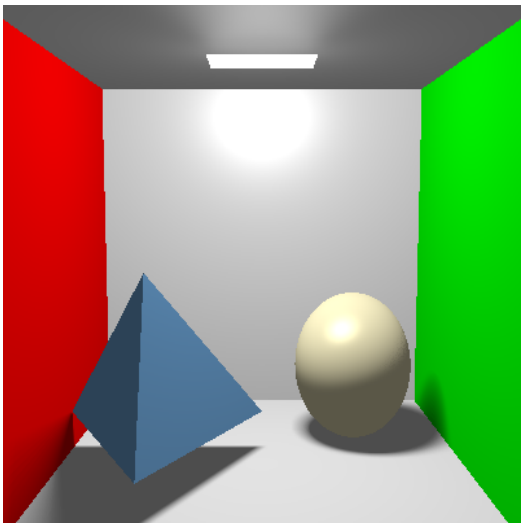
```
for (int i = 0; i < samples; i++) {
  float u = (x + halton(i, 2)) / width;
  float v = (y + halton(i, 3)) / height;
}

// Generate ray and compute color
generateRay(u, v, ray);
color += radiance(ray);
```

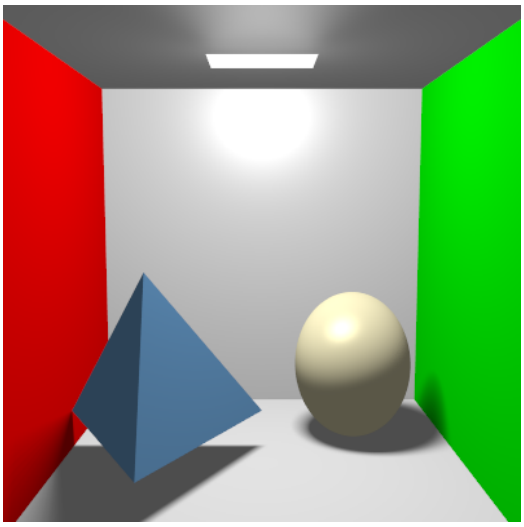
3 RESULTS

3.1 Plain result and Anti-aliasing using super sampling

The ordinary result, along with the result of anti-aliasing using super sampling, are shown below. In anti-aliasing, we use rotated grid.



(a) No antialiasing

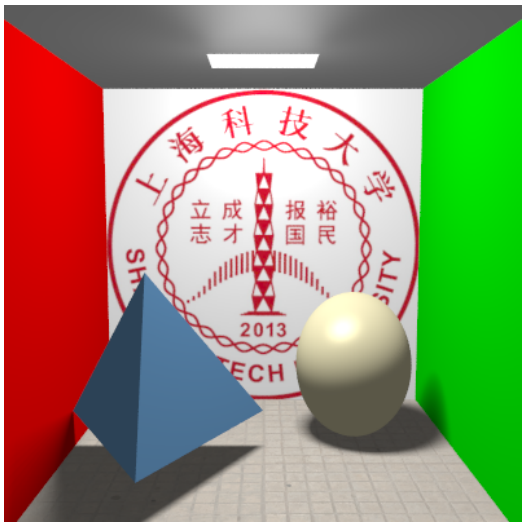


(b) With antialiasing

Fig. 1. Super resolution anti-aliasing

3.2 Texture mapping

We apply texture mapping to the 'back wall' and 'floor' rectangle geometry. The rendered image is below.

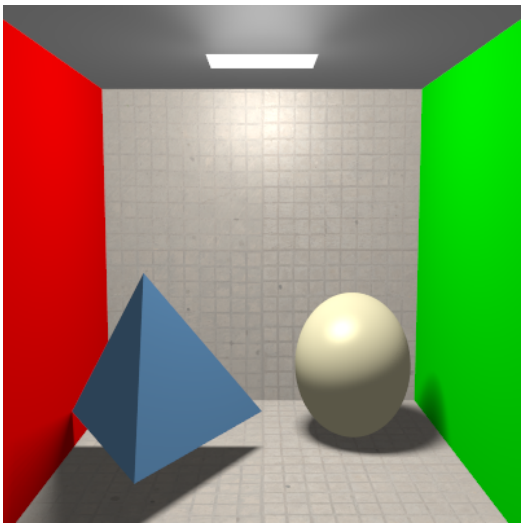


(a) Texture mapping

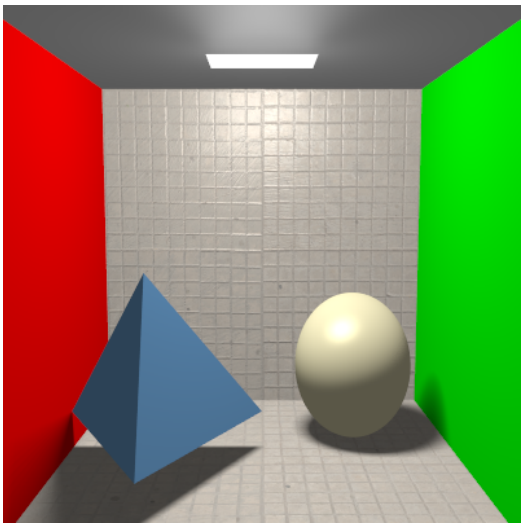
3.3 Normal texture mapping

We apply normal texture mapping to the 'back wall' and 'floor' rectangle geometry.

By comparing the image without and with normal texture mapping, we can clearly see that the latter one looks more rough and realistic.



(b) Without normal texture

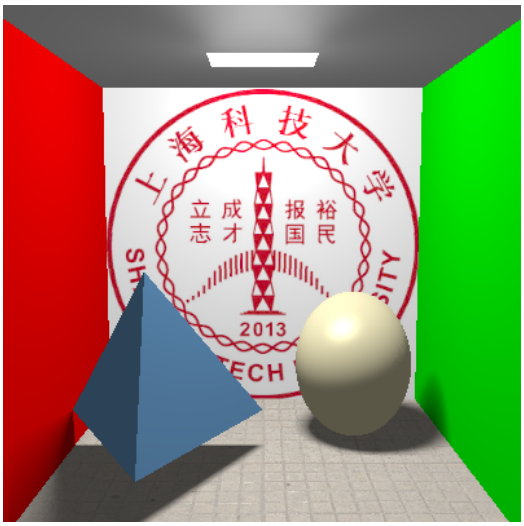


(c) With normal texture

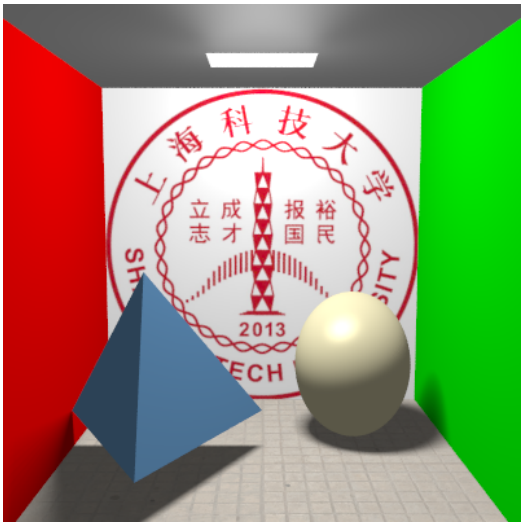
Fig. 2. Normal texture mapping

3.4 Advanced anti-aliasing by Halton sequence

We apply advanced anti-aliasing by Halton sequence. The rendered image is below.



(a) No antialiasing



(b) Halman sequence anti aliasing

Fig. 3. Advanced antialiasing