Assignment 3: Ray Tracing Basics

NAME: ENTROPY-FIGHTER
STUDENT NUMBER: 2020533
EMAIL: XXXX@SHANGHAITECH.EDU.CN

1 INTRODUCTION

I do the must, bonus1, bonus2 and bonus4.

- (must)generate rays from camera (10 pts)
- (must)ray geometry intersection (30 pts)
- (must)Phong lighting at intersection (15 pts)
- (must)light ray sampling for soft shadow (25 pts)
- (must)anti-aliasing by super-resolution (10 pts)
- (optional)texture mapping (10 pts)
- (optional)normal/displacement texture (15 pts)
- (optional)advanced anti-aliasing (10 pts)

2 IMPLEMENTATION DETAILS

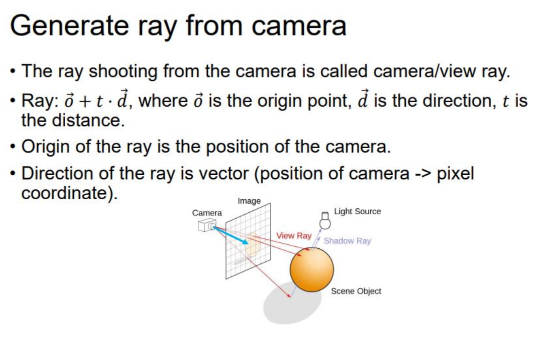
2.1 Construct a virtual Camera

This part is implemented in the "camera.cpp". In order to construct a virtual camera, we need to know the resolution, height, weight, the world position, the field of view, the focal lenght.

Also, it is important to get the lookat matrix. It is implemented in "lookat" function. The forward direction is calculated based on the "lookat coordinate" and the camera position. Then we calculate the right direction by the cross product of calculated forward direction and ref_up direction. At last, the up direction is calculated by the cross product of right direction and forward direction.

2.2 Generate Rays From Camera

This part is implemented in the "camera.cpp". This part is illustrated clearly in the tutorial6, first, let us see 2 pictures.



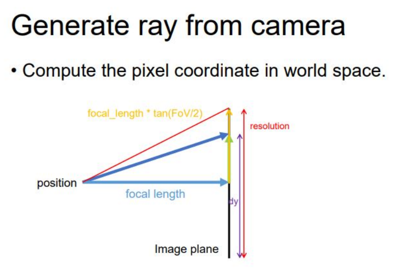
Generate ray from camera

- The ray shooting from the camera is called camera/view ray.
- Ray: $\vec{o} + t \cdot \vec{d}$, where $\vec{o}$ is the origin point, $\vec{d}$ is the direction, t is the distance.
- Origin of the ray is the position of the camera.
- Direction of the ray is vector (position of camera -> pixel coordinate).

Image
Camera
Light Source
View Ray
Shadow Ray
Scene Object

In order to get a ray from the camera, we have "dx" and "dy" as input parameters but we need the world coordinates.

Firstly we should normalize the dx and dy, where (0, 0) is the orgin. After that, we have some small tricks to transform these coordinates



Generate ray from camera

- Compute the pixel coordinate in world space.

to screen coordinates. The small tricks to transform to the screen coordinates can be easily knowned from the figure 2 above.

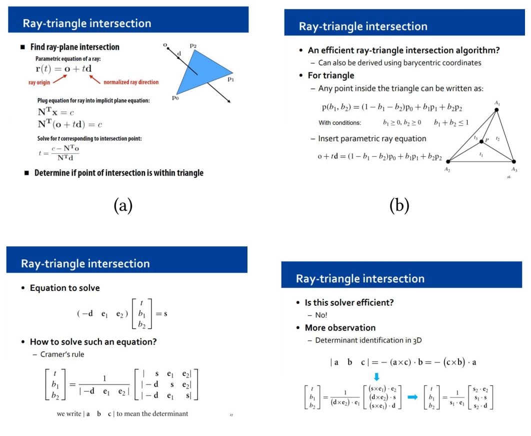
Then, we use the formula: "position = screenCenter + screen_x * right + screen_y * up" to get world position, use formula: "(position - this->position).normalized()" to get direction.

At last, The ray can be easily discribed as O + t * d.

2.3 Ray Geometry Intersection

This part is implemented in the "geometry.cpp". There are 3 types of geometries: triangle, rectangle and ellipsoid.

As for triangles, the class ppt gives us a quick solution. The concrete steps can be seen in pictures bellow. Although the matrix calculation is complex, we just use the conclusion in slides to code.



Ray-triangle intersection

- Find ray-plane intersection
  - Parametric equation of a ray:
    $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
  - Plug equation for ray into implicit plane equation:
    $\mathbf{N}^{\mathbf{T}}\mathbf{x} = c$
    $\mathbf{N}^{\mathbf{T}}(\mathbf{o} + t\mathbf{d}) = c$
  - Solve for *t* corresponding to intersection point:
    $t = \frac{c - \mathbf{N}^{\mathbf{T}}\mathbf{o}}{\mathbf{N}^{\mathbf{T}}\mathbf{d}}$
- Determine if point of intersection is within triangle

(a)

Ray-triangle intersection

- An efficient ray-triangle intersection algorithm?
  - Can also be derived using barycentric coordinates
- For triangle
  - Any point inside the triangle can be written as:
    $\mathrm{p}(b_1, b_2) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2$
    With conditions: $b_1 \geq 0, b_2 \geq 0 \quad b_1 + b_2 \leq 1$
  - Insert parametric ray equation
    $\mathrm{o} + t\mathrm{d} = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2$

(b)

Ray-triangle intersection

- Equation to solve

$$\begin{pmatrix} -\mathrm{d} & \mathrm{e}_1 & \mathrm{e}_2 \end{pmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \mathrm{s}$$

- How to solve such an equation?
  - Cramer's rule

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{|-\mathrm{d} \quad \mathrm{e}_1 \quad \mathrm{e}_2|} \begin{bmatrix} |\ \mathrm{s} \quad \mathrm{e}_1 \quad \mathrm{e}_2| \\ |-\mathrm{d} \quad \mathrm{s} \quad \mathrm{e}_2| \\ |-\mathrm{d} \quad \mathrm{e}_1 \quad \mathrm{s}| \end{bmatrix}$$

we write $|\mathrm{a} \quad \mathrm{b} \quad \mathrm{c}|$ to mean the determinant

27

(c)

Ray-triangle intersection

- Is this solver efficient?
  - No!
- More observation
  - Determinant identification in 3D

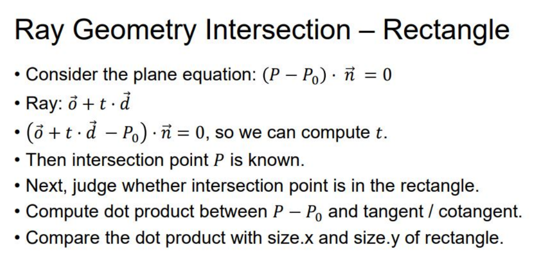
$$|\mathrm{a} \quad \mathrm{b} \quad \mathrm{c}| = -(\mathrm{a} \times \mathrm{c}) \cdot \mathrm{b} = -(\mathrm{c} \times \mathrm{b}) \cdot \mathrm{a}$$

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{(\mathrm{d} \times \mathrm{e}_2) \cdot \mathrm{e}_1} \begin{bmatrix} (\mathrm{s} \times \mathrm{e}_1) \cdot \mathrm{e}_2 \\ (\mathrm{d} \times \mathrm{e}_2) \cdot \mathrm{s} \\ (\mathrm{s} \times \mathrm{e}_1) \cdot \mathrm{d} \end{bmatrix} \Rightarrow \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\mathrm{s}_1 \cdot \mathrm{e}_1} \begin{bmatrix} \mathrm{s}_2 \cdot \mathrm{e}_2 \\ \mathrm{s}_1 \cdot \mathrm{s} \\ \mathrm{s}_2 \cdot \mathrm{d} \end{bmatrix}$$

(d)

Fig. 1. Triangle interseciton

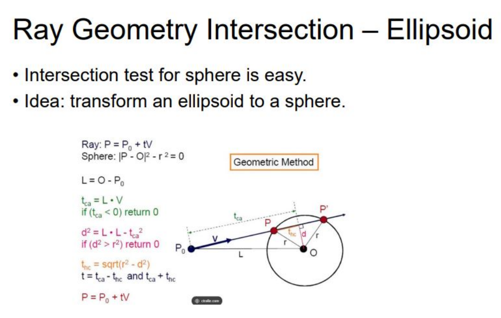
As for rectangles, the tutorial6 gives us a good method. First, we need to calculate the intersection point p, then we judge whether intersection point is in the rectangle. The basic idea is to compute dot product between p - p0 and tangent(cotangent), then compare them with size.x/2 and size.y/2.



Ray Geometry Intersection – Rectangle

- Consider the plane equation: $(P - P_0) \cdot \vec{n} = 0$
- Ray: $\vec{o} + t \cdot \vec{d}$
- $(\vec{o} + t \cdot \vec{d} - P_0) \cdot \vec{n} = 0$, so we can compute t.
- Then intersection point P is known.
- Next, judge whether intersection point is in the rectangle.
- Compute dot product between $P - P_0$ and tangent / cotangent.
- Compare the dot product with size.x and size.y of rectangle.

As for ellipsoid, it seems hard. However, the process for sphere is easy. Therefore, our basic idea is to do matrix transformation to simplify the problem to ray-sphere interseciton problem. The concrete steps are below.



Ray Geometry Intersection – Ellipsoid

- Intersection test for sphere is easy.
- Idea: transform an ellipsoid to a sphere.

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Geometric Method

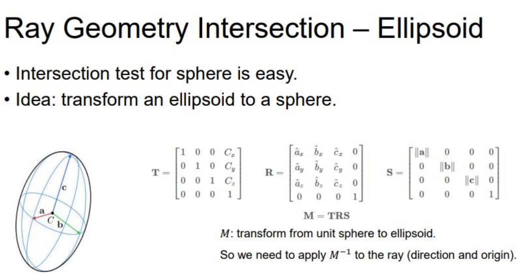
$L = O - P_0$

$t_{ca} = L \cdot V$
if ($t_{ca} < 0$) return 0

$d^2 = L \cdot L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \text{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

$P = P_0 + tV$



Ray Geometry Intersection – Ellipsoid

- Intersection test for sphere is easy.
- Idea: transform an ellipsoid to a sphere.

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & C_x \\ 0 & 1 & 0 & C_y \\ 0 & 0 & 1 & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} \hat{a}_x & \hat{b}_x & \hat{c}_x & 0 \\ \hat{a}_y & \hat{b}_y & \hat{c}_y & 0 \\ \hat{a}_z & \hat{b}_z & \hat{c}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} \|\mathbf{a}\| & 0 & 0 & 0 \\ 0 & \|\mathbf{b}\| & 0 & 0 \\ 0 & 0 & \|\mathbf{c}\| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M} = \mathbf{TRS}$$

M: transform from unit sphere to ellipsoid.

So we need to apply M^{-1} to the ray (direction and origin).

2.4 Phong Lighting at Intersection(Evaluate Radiance)

This part is implemented in the "integrator.cpp". We need to evaluate radiance using Phong lighting model. In concrete, we should calculate the ambient, diffuse and specular at interseciton.

Ambient light comes from the environment. It is easy to calculate.

As for diffuse, we have to calculate dot product of light direction vector and normal vector, and multiply it with ligth_color to get diffuse light.

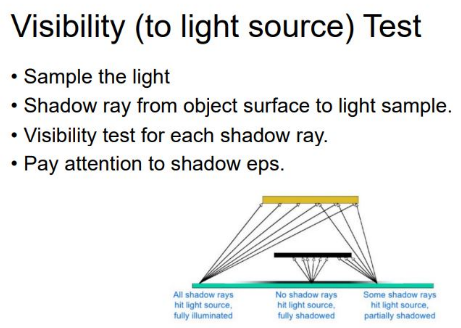
As for specular, we first use math knowledge to calculate the reflected light direction vector. Also, we need to calculate the view direction vector. The next step is to calculate the dot product between the view direction and the reflect direction (it's non-negative) and then raise it to the shininess value of the highlight. Next step is to multiply this with light color and a specular intensity value to get specular light.

At last, we sum them up.

2.5 Direct Ray Tracing and Light Ray Sampling

This part is implemented in the function render() in "integrator.cpp". If the ray first intersects with the LIGHT, then the radiance is the light color. If the ray first intersects with the GEOMETRY, we need to generate a new ray which comes from the intersection postion to the light position. If the generated ray intersects with the GEOMETRY, then it is shadowed. We just need to calculat the ambient light. Otherwise, the radiance is set to the return value of radiance() function which is based on phong lighting model.

We have square area light, so we need to sample points from the rectangle. Therefore, different from point light, we can render the soft shadow effect by calculating the mean value. This soft shadow effect is illustrated in the picture below.



Visibility (to light source) Test

- Sample the light
- Shadow ray from object surface to light sample.
- Visibility test for each shadow ray.
- Pay attention to shadow eps.

All shadow rays hit light source, fully illuminated
No shadow rays hit light source, fully shadowed
Some shadow rays hit light source, partially shadowed

2.6 Anti-aliasing by Super-resolution

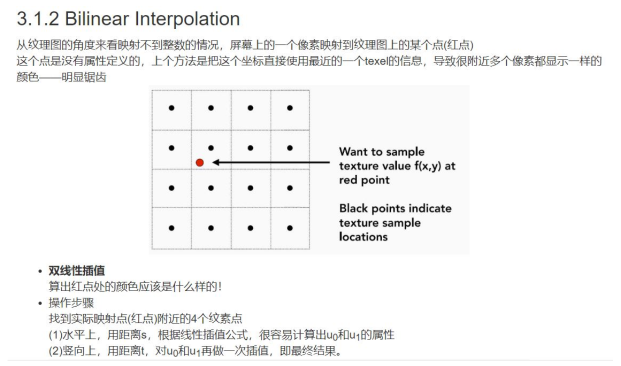
This part is implemented in sample_pixel_uniform() function in "camera.cpp". I do the super resolution, which means I sample many points in each single pixel. The easiest way to do this is to take sample uniformly. Furthermore, I apply some small tricks on it to get better effect, using the magic angle 26.6f. I do some rotations on the uniform samples. See more information in section "Advanced anti_aliasing".

2.7 Texturing

This part is implemented most in the "texture.cpp", some in the "material.cpp" and "geometry.cpp".

The basic idea for texturing is mapping. In geometry.cpp, I need to add information in interaction.uv when doing the intersection. I just want to texture some rectangles, so I only modify the ray-rectangle intersection function. It is easy to get uv when we know the 2 direction dot product. The one thing to notice is that the uv would better be restricted to [0, 1]*[0, 1].

In "texture.cpp", the most important thing is to do the interpolation to get pixel value. I do the bilinear interpolation. I learn the knowledge about bilinear interpolation from the https://copyfuture.com/bl details/202112121313325372. The concrete algorithm is showed clearly in the picture below.

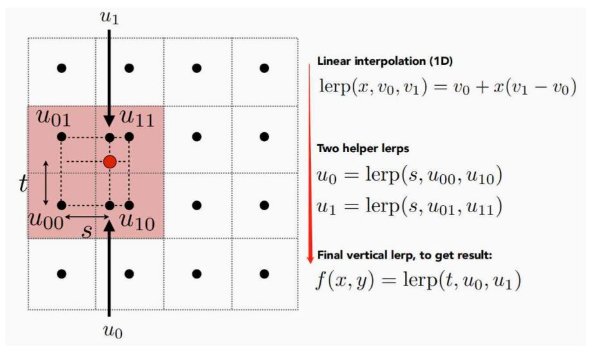


3.1.2 Bilinear Interpolation

从纹理图的角度来看映射不到整数的情况，屏幕上的一个像素映射到纹理图上的某个点(红点)
这个点是没有属性定义的，上个方法是把这个坐标直接使用最近的一个texel的信息，导致很附近多个像素都显示一样的颜色——明显锯齿

Want to sample texture value f(x,y) at red point
Black points indicate texture sample locations

- **双线性插值**
  算出红点处的颜色应该是什么样的！
- 操作步骤
  找到实际映射点(红点)附近的4个纹素点
  (1)水平上，用距离s，根据线性插值公式，很容易计算出u_0和u_1的属性
  (2)竖向上，用距离t，对u_0和u_1再做一次插值，即最终结果。


u_1
u_{01}
u_{11}
t
u_{00}
s
u_{10}
u_0
Linear interpolation (1D)
$\text{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$
Two helper lerps
$u_0 = \text{lerp}(s, u_{00}, u_{10})$
$u_1 = \text{lerp}(s, u_{01}, u_{11})$
Final vertical lerp, to get result:
$f(x, y) = \text{lerp}(t, u_0, u_1)$

In "material.cpp", we need to load textures and save them, specifying the type of them.(This step is to distinguish the normal texture and disp texture from basic texture.)

2.8 Normal Texturing

This part is most implemented in the "geometry.cpp".

For the basic texturing of a rectangle, the normals of texture are all the normal of the rectangle, which looks weird. If we use normal mapping, the normals of texture are given by the norm.jpg. However, these normals are not actually what we want, we need to do some transformation on them(aka normal mapping). The core code of normal mapping is given below.

```
if(material->norm_t){
  Vec2f pos;
  Vec3f local_normal;
  pos = Vec2f(interaction.uv[0], interaction.uv[1]).
       cwiseProduct(Vec2f((float)material->norm_t->
       weight, (float)material->norm_t->height));
  local_normal = material->norm_t->get_tex_data((int
       )pos[1] + (int)pos[0] * material->norm_t->
       height);
  Vec3f final_normal = local_normal[2] * normal +
       local_normal[0] * tangent.normalized() +
       local_normal[1] * tangent.normalized().cross(
       normal.normalized());
  interaction.normal = final_normal.normalized();
  }
```

2.9 Advanced Anti-aliasing

This part is implemented in sample_pixel() function in "camera.cpp". In section "Anti-aliasing", I take samples uniformly. Furthermore, to get better effects, I apply some small tricks.

Firstly, I pick the samples randomly. However, there would be some hot/noise points! This isn't what I want. Therefore, I try to rotate the uniform samples using the magic angle 26.6f. I use the rotation matrix to multiply the samples and get points which are proper(not out of range). In this way, the effect of anti-aliasing seems better than the uniform sampling. The comparison can be seen in the result pictures.(I create a new scene where there is a ground made up of chess board, where the difference can be seen clearly)

3 RESULTS

Figure 2 is the Must part. Figure 3 is the Texturing. Figure 4 is Normal Texturing. Figure 5 is Anti-aliasing with Uniform Sampling. Figure 6 is Advanced Anti-aliasing by Rotating the uniform samples.

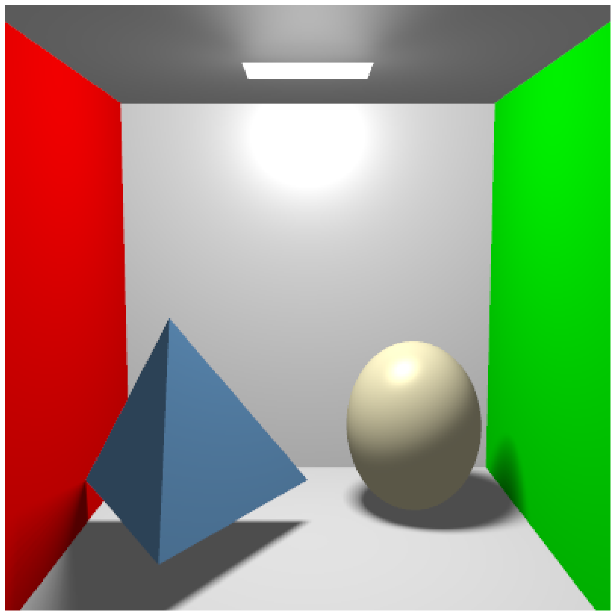


Fig. 2. Basic ray tracing

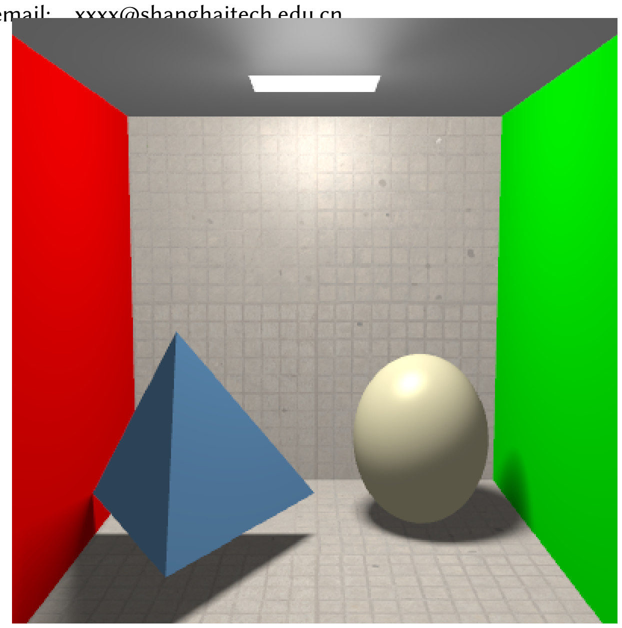


Fig. 3. Texturing

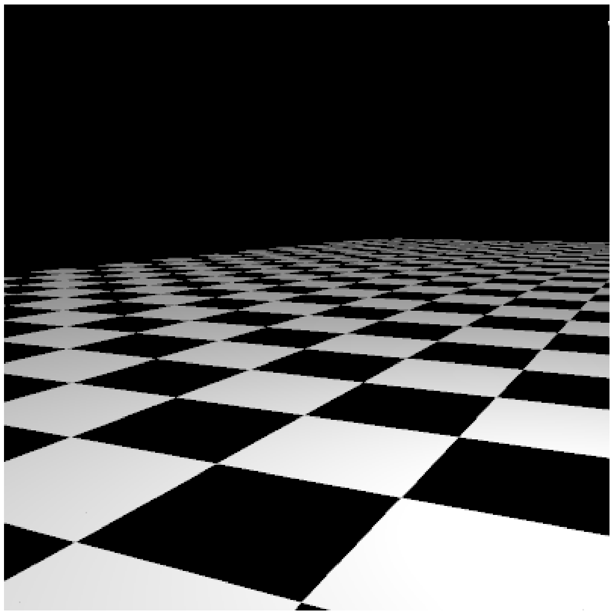


Fig. 5. Anti-aliasing with uniform sampling

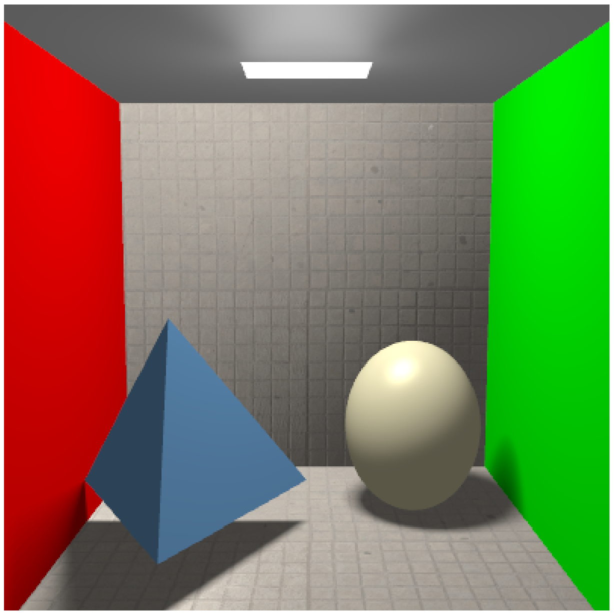


Fig. 4. Normal texturing

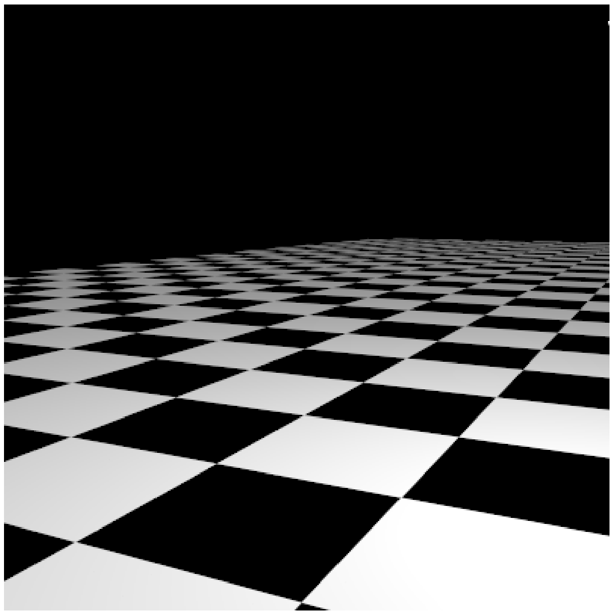


Fig. 6. Advanced anti-aliasing by rotating the uniform samples