The Demo

- Creating and editing 3D object
  - The operation to create and edit 3D objects is the same as that of the template used in previous programming assignment.
  - Select an object, click  on the toolbar to change the material.
  - Click  on the toolbar to add a light source.
  - Select a light source and click  to change its color.
- Ray tracing
  - Click the menu item "Tool → Render Full Window" or the icon  on the toolbar to do ray tracing.
  - You can first click the menu item "Tool → Render Region" and drag your mouse to choose a region to do ray tracing.
- Texture Mapping
  - You can first select an object and click the icon  on the toolbar to choose a texture for it.

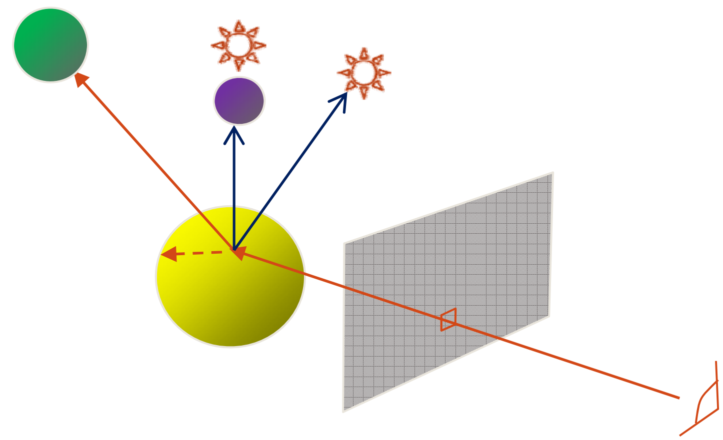
Provided Functions

- Reading and parsing a scene file.
- Creating and editing 3D objects.
- Navigation.
- Rendering the scene with Gouraud Shading.
- Texture mapping of the sphere.

Your Task

- Fill in five functions in code.cpp to implement ray tracing.
  - RayTracing(colorMap);
    - Calculate the color of each pixel and save it in colorMap.
  - Trace(ray);
    - Given a ray, calculate the color at the nearest intersection point.
  - Shade(point);
    - Given a point, calculate the color of this point.
  - IntersectQuadratic(ray, quadratic);
    - Do intersection of a ray and a quadratic surface.
  - IntersectTriangle(ray, triangle);
    - Do intersection of a ray and a triangle.

Relationship between functions



RayTracing(colorMap)
- Calculate the color of each pixel and save it in colorMap

call

Trace(ray, depth, color)
- Given a ray, calculate the color at the nearest intersection point.

- Color=ambient+$\sum S_i$(diffuse + specular)+reflection+refraction

Second call

Shade(ray, depth, color)
- Given a point, calculate the color of this point.

If depth<maxDepth
call depth+1

First call

call

Intersect(ray)
- Provided by the template.
- Visit each object in the scene. Find the nearest intersection point.

call

call

IntersectQuadratic()
Do intersection of a ray and a quadratic surface.

IntersectTriangle()
Do intersection of a ray and a triangle.

Provided Information

```
// in code.cpp

int winWidth = 640;             // window width
int winHeight = 480;            // window height

V3 ViewPoint;                   // view point
vector<CLightSource *> vLightSource;  // array of light sources
// coordinates at corners of image
V3 ImageLL;                     // lower left
V3 ImageLR;                     // lower right
V3 ImageUL;                     // upper left
V3 ImageUR;                     // upper right
```

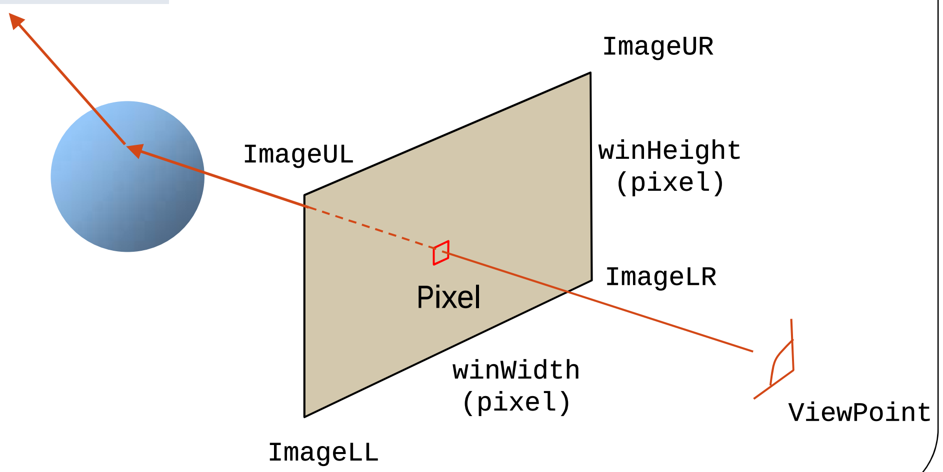
```
// in code.cpp

// depth of recursive ray-
// tracing

int MaxTraceDepth = 5;
```

The coordinates at pixel (i, j) are given by:

```
ImageLL+i*(ImageLR-ImageLL)/(winWidth-1)
+j*(ImageUL-ImageLL)/(winHeight-1)
```



Function: RayTracing()

- Interface
  - **void RayTracing(V3 * colorMap);**
- Task
  - Visit each pixel on the screen.
  - For each pixel p(i,j), i $\in$ [0, winWidth-1] ,j $\in$ [0, winHeight-1]
    - Get the world coordinates of p.
    - Fire a ray from the viewpoint through p.
      - V3 rayStart= viewpoint ;
      - V3 rayDir=world coordinates of p - rayStart
    - Call Trace() function to trace the ray and get the color of p.
    - Save color in colorMap.
      - colorMap[j*winWidth+i]=color;

Function: Trace ()

- Interface
  - **void Trace(V3& rayStart, V3& rayDir, int depth, V3& color);**
- Task
  - Trace one particular ray and calculate the color at the closest intersection point.
- Parameters
  - rayStart: Start point of the ray
  - rayDir: Direction of the ray (unit vector). rayDir.normalize()
  - Depth: The current depth in ray tracing
  - color: The color at the closest intersection point .Range from 0 to 1.(return value)

Function: Trace ()

- Implementation

  IF Intersect() Then

  - Call shade() function to calculate the color of the intersection point.
  - Return this color.

  ELSE

  - Return the background color(0,0,0).

Function: Intersect ()

- Interface
  - **bool Intersect(V3 rayStart, V3 rayDir, CPrimitive *&objHit, V3& intersection, V3& normal);**
- Task
  - Do ray-object intersection.
  - If there is no intersection, return false. Otherwise, return the information of the closest intersection point.
- Parameters
  - objHit: The closest object which intersects with the ray.
  - intersection: The coordinates of the intersection point
  - normal: The normal vector at the intersection point

Function: Shade ()

- Interface
  - **void Shade(CPrimitive *obj,V3& rayStart,V3& rayDir, V3& intersection,V3& normal, int depth,V3& color);**
- Task
  - Calculate the color at a given intersection point.
- Parameters
  - obj: The object which intersect with the ray.
  - intersection: The coordinates of the intersection point
  - normal: The normal vector at the intersection point
  - depth: The current depth in ray tracing
  - color: The color at the intersection point.(output)

Shading formula

- Intensity (color) at a point P on an object is given by

$$I = \underbrace{O_a}_{\text{Ambient term}} + \underbrace{\sum S_i I_{pi} [(1 - k_t) O_d (N \cdot L_i) + k_s O_s (R \cdot V)^n]}_{\text{Diffuse and specular terms due to each light source}} + \underbrace{k_s I_r}_{\text{Reflected ray contribution}} + \underbrace{k_t I_t}_{\text{Refracted ray contribution}}$$

Refracted ray contribution
This assignment does not require implementation of refraction. So we simply discard this term.

where

O_a = object ambient color,

obj->GetAmbient(intersection, ambientColor);

S_i = 0 means light i is blocked at P, 1 means light is not blocked at P.

If N.dot(L_i) >0 , call Intersect function to see whether it's blocked.

I_{pi} = intensity(color) of light i , vLightSource[i]->color:V3

k_t = object transparency ,1 – obj->m_Opacity

O_d = object diffuse color,

obj->GetDiffuse(intersection, diffuseColor);

N = normal at P, normal:V3

n = object shininess, obj->m_Shininess

O_s = object specular color ,

obj->GetSpecular(intersection,specularColor);

L_i = direction to light i at P,

vLightSource[i]->position - intersection

k_s = object reflectance, obj->m_Reflectance

I_r = intensity of reflected ray

I_t = intensity of refracted ray

R = direction of reflection at P, 2 * N.dot(L_i) * N - L_i

V = direction to viewpoint at P, rayStart – intersection

Get Object Color - O_a, O_d, O_s

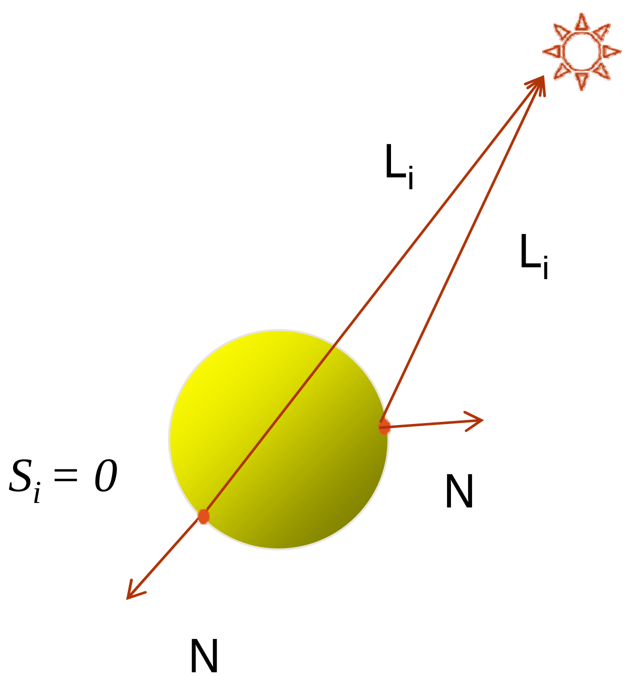
$$I = \boxed{O_a} + \sum S_i I_{pi} [(1 - k_t) \boxed{O_d} (N \cdot L_i) + k_s \boxed{O_s} (R \cdot V)^n] + k_s I_r + k_t I_t$$

- Interface
  - void CPrimitive:: GetDiffuse(V3 point,V3& diffuse);
    - Get diffuse component and save it in diffuse.
  - void CPrimitive:: GetAmbient(V3 point,V3& ambient);
    - Get ambient component and save it in ambient.
  - void CPrimitive:: GetSpecular(V3 point,V3& specular);
    - Get specular component and save it in specular.
- Parameter
  - point: World coordinates of the intersection point.(input)
  - The second parameter is used to store the color you want. (output)

Get the value of S_i

$$I = O_a + \sum \boxed{S_i} I_{pi} \left[(1 - k_t)\, O_d\, (N \cdot L_i) + k_s\, O_s\, (R \cdot V)^n \right] + k_s\, I_r + k_t\, I_t$$

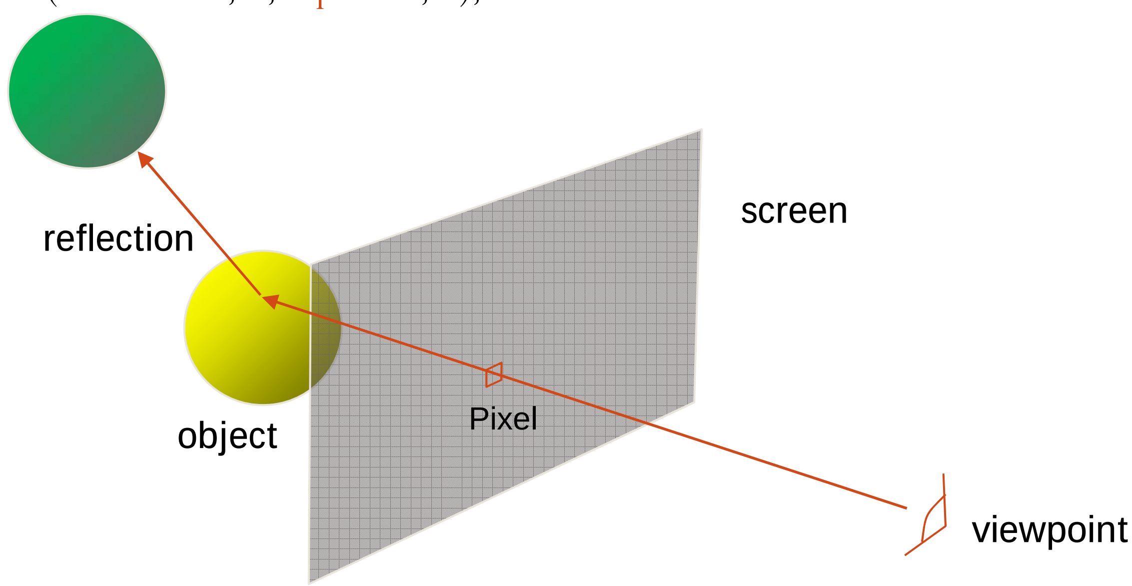
- If N.dot(L_i) <0 , no need to check.
- Otherwise, call Intersect() function to see whether it's blocked.



Reflection

$$I = O_a + \sum S_i I_{pi} [(1 - k_t) O_d (N \cdot L_i) + k_s O_s (R \cdot V)^n] + \boxed{k_s I_r} + k_t I_t$$

- If depth < MaxTraceDepth,
  - Get the reflected ray whose start point is the intersection point and direction is the reflection direction.
  - Call Trace() function to recursively trace this ray and get the color of the closest intersection point I_r.
    - Trace(intersection, R, depth + 1, Ir);



Function: Shade ()

Ambient term

Diffuse and specular terms due to each light source

Reflected ray contribution

```
Void  Shade(obj, rayStart, rayDir, intersection, normal, depth, color)
{
    color := ambient color;
    for each light source do {
        sRay := ray to light from intersection point;
        if N.dot(L_i) >0  then{
            call Intersect() to see whether sRay is blocked;
            if(sRay is not blocked by other objects)
            compute the second part of the shading formula and add it to color;
        }
    }
    if depth < MaxTraceDepth then
            if object is reflective then {            // i.e., reflectance > 0
                        rRay := ray in reflection direction from intersection;
                        Trace( intersection, rRay, depth + 1,rColor );
                        scale rColor by reflectance and add to color;
                        }
    if color > 1.0 then     clamp color to 1.0
}
```

Function: IntersectQuadratic()

General form of quadratic surface:

$$ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2iz + j = 0$$

$$[x, y, z, 1]\begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & i \\ g & h & i & j \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

Using homogeneous coordinates.

$$X^T AX = 0$$

A is given by an input parameter:

float[16] coeffMatrix;
It represents A in the following sequence.

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Function: IntersectQuadratic()

- The parametric ray is

$$R(t) = S + Dt, t \geq 0$$

where S=(rayStart[0], rayStart[1], rayStart[2], 1)^T

D=(rayDir[0], rayDir[1], rayDir[2], 0)^T

- Substituting it in the quadric surface, we have

$$(S + Dt)^T A(S + Dt) = 0$$

$$(D^T AD)t^2 + 2(S^T AD)t + S^T AS = 0$$

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

- The determinant of this equation is

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

- If $\Delta > 0$, $t_0 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, t_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$.
- If $\Delta = 0$, $t = -\frac{b}{2a}$.
- Choose the smallest positive t.
- If there is no feasible t, return false. Otherwise, return true.
- The intersection point : S+Dt

Some useful functions

- void MatrixMultVector(float *m, float *v, float *rv);
  - Multiply a 4*4 matrix m by a 4*1 vector v.
  - rv=m*v
- void VectorMultMatrix(float *v, float *m, float *lv);
  - Multiply a 4*1 vector v by a 4*4 matrix m .
  - $lv=v^T m$
- float VectorMultVector(float *v1,float *v2);
  - Dot product of two 4*1 vectors.
  - return $v1^T v2$
- Don't use null pointer as parameter.
  - Correct parameter: float rv[16];
  - Wrong parameter :float * rv;

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Example:
Calculate $a=D^TAD$

```
float temp[4];
VectorMultMatrix(D,A,temp);
float a = VectorMultVector(temp,D);
```