GRAPHICS GROUP

Computer Graphics and Image Processing Ray Tracing II

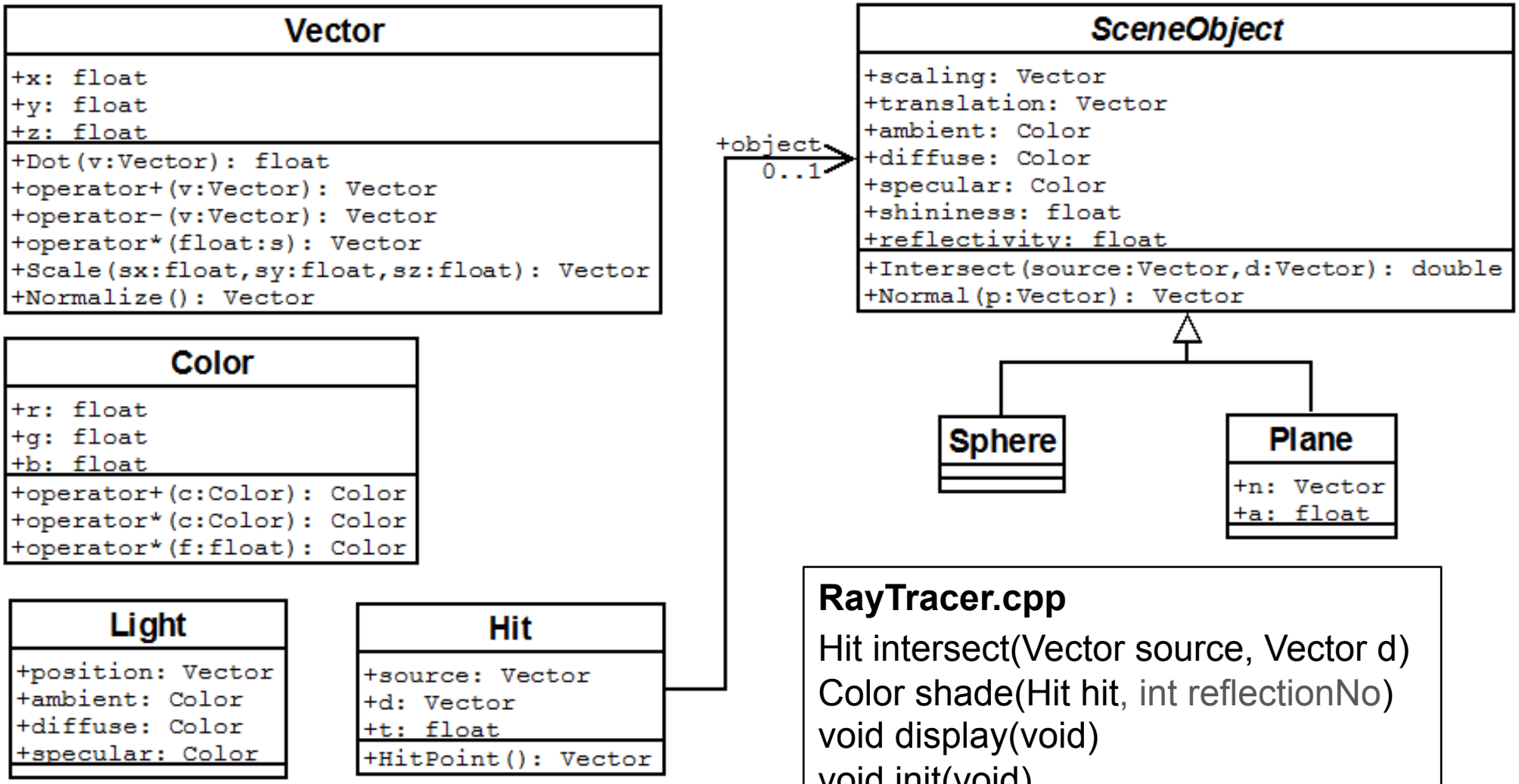
Part 1 – Lecture 10

Today's Outline

1. Structure of a Ray Tracer
2. Implementing a Ray Caster
3. Lights and Shadows

STRUCTURE OF A RAY TRACER

Ray Tracer Class Diagram



Class `Vector`

`Vector.h`

```
#pragma once
#include <cmath>

class Vector {
public:
    Vector();
    Vector(float x, float y, float z);
    ~Vector(void);

    float x, y, z;

    float Dot(Vector v); // dot product
    Vector operator+(Vector v);
    Vector operator-(Vector v);
    Vector operator*(float s);
    Vector Scale(float sx, float sy, float sz);
    Vector Normalize();
};
```

`Vector.cpp`

```
#include "Vector.h"

Vector::Vector() { }
Vector::Vector(float x, float y, float z) { … }
Vector::~Vector(void) { }

float Vector::Dot(Vector v) {
    return x*v.x + y*v.y + z*v.z;
}

Vector Vector::operator+(Vector v) {
    return Vector(x+v.x, y+v.y, z+v.z);
}

Vector Vector::Normalize() {
    float l = sqrt(this->Dot(*this));
    return Vector(x/l, y/l, z/l);
}

…
```

Class `Color`

`Color.h`

```
#pragma once
class Color {
public:
    Color();
    Color(float r, float g, float b);
    ~Color(void);
    float r, g, b;
    Color operator+(Color c);
    Color operator*(Color c);
    Color operator*(float s);
};
```

`Color.cpp`

```
#include "Color.h"
Color::Color() { }
Color::Color(float r, float g, float b) {
    this->r = r; this->g = g; this->b = b;
}
Color::~Color(void) { }
Color Color::operator+(Color c) {
    return Color(r + c.r, g + c.g, b + c.b);
}
Color Color::operator*(Color c) {
    return Color(r * c.r, g * c.g, b * c.b);
}
Color Color::operator*(float s) {
    return Color(s*r, s*g, s*b);
}
```

The Main File: `RayTracer.cpp`

```
#include <windows.h>
#include <gl/glut.h>
#include <algorithm>
#include "Color.h“    …

int windowWidth = 400, windowHeight = 400;

const int numObjects = 1;
SceneObject* objects[numObjects];

const int numLights = 1;
Light* lights[numLights];

Color background = Color(0.5, 0.8, 1);

Vector eye = Vector(0, 0, 4);
Vector u = Vector(1, 0, 0);
Vector v = Vector(0, 1, 0);
Vector n = Vector(0, 0, 1);
float N = 1, W = 0.5, H = 0.5;
```

Headers

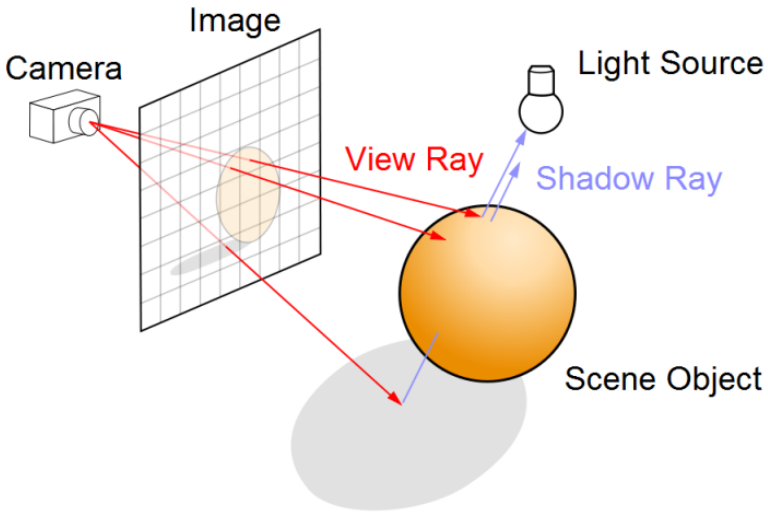
Scene objects

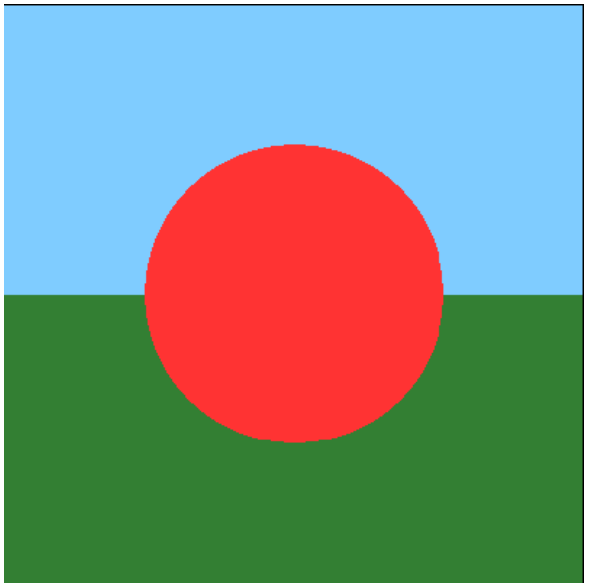
Lights

Camera

```
void init(void){
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, windowWidth,
       0, windowHeight);
    setupScene();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE |
    GLUT_RGB);
    glutInitWindowSize(
      windowWidth, windowHeight);
    …
}
```

Orthogonal projection





IMPLEMENTING A RAY CASTER

Diagram thanks to Henrik

Casting Rays

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POINTS);
  for(int r=0; r<windowHeight; r++) {
    for(int c=0; c<windowWidth; c++) {
      // construct ray through (c, r)
      // using u,v,n and H,W
      Vector d = Vector( ? , ? , ? );
      // intersect ray with scene objects
      Hit hit = intersect(eye, d);
      // shade pixel accordingly
      Color color = shade(hit);
      glColor3f(color.r, color.g, color.b);
      glVertex2f((GLfloat)c, (GLfloat)r);
  } }
  glEnd(); glFlush ();  }
```

- We are drawing the points on the viewplane, starting at bottom left
- Ray starts at **eye** and has direction **d** (→ calculate **d**)
- `intersect(eye,d)` gives us info about first hit of the ray
- `shade(hit)` gives us the color of the point where the ray hit an object (or hit no object at all)

Use only floating point constants when calculating **d** to prevent rounding errors (sphere will show up as square)

Representing Objects

#pragma once … `SceneObject.h`

```
class SceneObject {
public:
    Vector scaling, translation;
    Color ambient, diffuse, specular;
    float shininess;
    float reflectivity;

    // returns the t value of the closest ray-
    // object intersection, or -1 otherwise
    virtual double Intersect(
            Vector source, Vector d) = 0;

    // returns normal at the given point p
    // if p not on object, result is undefined
    virtual Vector Normal(Vector p) = 0;
};
```

#pragma once … `Sphere.h`

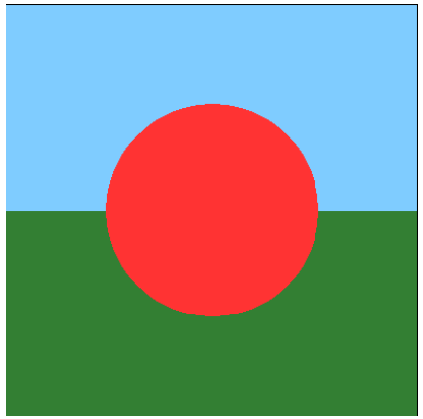
```
class Sphere : public SceneObject {
public:
    Sphere(void); ~Sphere(void);
    double Intersect(Vector source, Vector d);
    Vector Normal(Vector p);
};
```

For the ray caster we use only green parts

- `scaling` and `translation` for transformed objects
- `ambient`, `specular` colors and `shininess` for Phong illumination
- `reflectivity` for ray reflection

Setting Up Scene Objects

```
Sphere* s = new Sphere();
// no scaling or translation
s->scaling = Vector(1, 1, 1);
s->translation = Vector(0, 0, 0);
// Phong material parameters
s->ambient = Color(0.1, 0.1, 0.1);
s->diffuse = Color(1.0, 0.2, 0.2);
s->specular = Color(0.7, 0.7, 0.7);
s->shininess =  50;
// ray reflection parameter
s->reflectivity = 0.4;
// put into objects array
objects[0] = s;
// similar for plane…
```



Set up objects in global variable:

```
const int numObjects = 2;
SceneObject* objects[numObjects];
```

Set up background color:

```
Color background =
        Color(0.5, 0.8, 1);
```

- Analogous for planes, but need to set more parameters: n and a
- For now we ignore scaling and translation, but later we support scaled and translated objects

Sphere.cpp

```
double Sphere::Intersect(
    Vector source, Vector d) {
    float A = d.Dot(d);
    float B = 2*source.Dot(d);
    float C = source.Dot(source) - 1;
    if(B*B - 4*A*C <= 0) return -1;  // no hit
    float t1;
    if(B>0)   // for numerical precision
        t1 = (-B - sqrt(B*B - 4*A*C)) / (2*A);
    else
        t1 = (-B + sqrt(B*B - 4*A*C)) / (2*A);
    float t2 = C/(A*t1); // easier way to get t2
    if(t1<t2) return t1;  // need only closer t
    else return t2;
}
```

```
Vector Sphere::Normal(Vector p) {
    return ? ;
}
```

- Class `Sphere` implements intersection and normal for sphere primitive (at origin with radius 1)
- If no intersection: return -1

Also need to create class `Plane`

- Fields: `Vector n` (normal) and `float a` (distance from origin)
- Implement `Intersect`

Class `Hit`

`Hit.h`

```
#pragma once
#include "Vector.h"
#include "SceneObject.h"

class Hit {
public:
    Hit(void);
    Hit(Vector source, Vector d, float t,
        SceneObject* object);
    ~Hit(void);

    Vector source;
    Vector d;
    float t;
    SceneObject* object;

    Vector HitPoint();
};
```

`Hit.cpp`

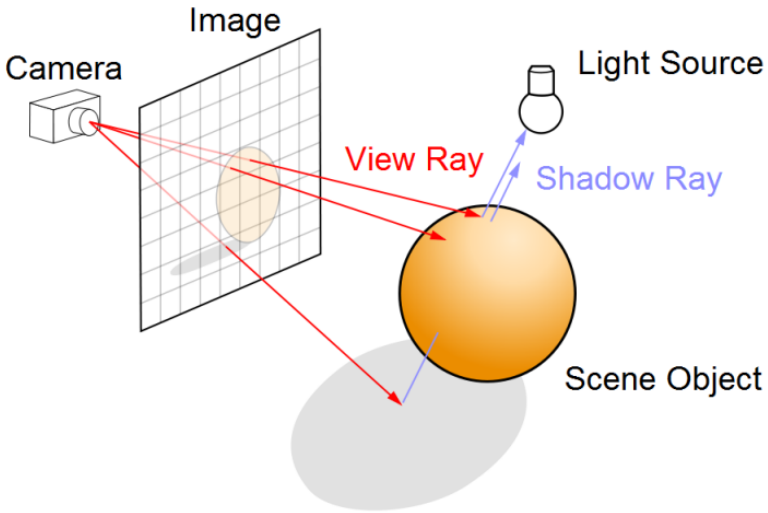
```
#include "Hit.h"

Hit::Hit(void) { }

Hit::Hit(Vector source, Vector d, float t,
    SceneObject* object) {
    this->source = source;
    this->d = d;
    this->t = t;
    this->object = object;
}

Hit::~Hit(void) { }

Vector Hit::HitPoint() {
    // calculate intersection point
    // using source, d and t
    return ? ;
}
```

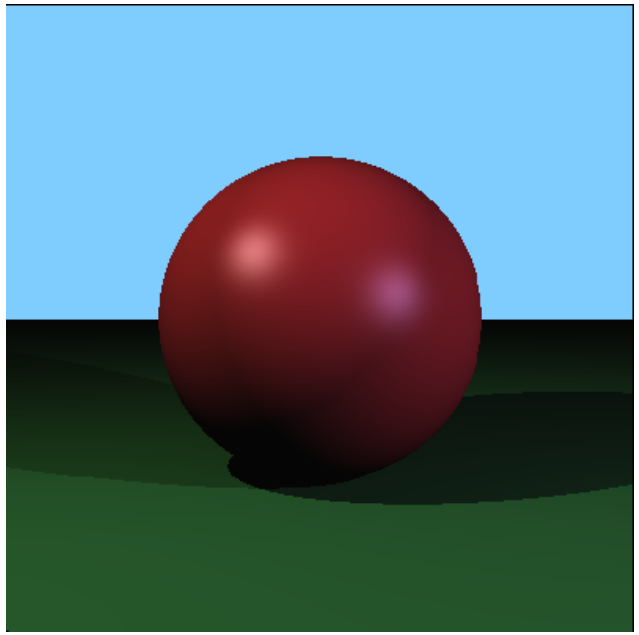
`intersect` and `shade`

```
Hit intersect(Vector source, Vector d) {
    // initially hit object==NULL (→ no hit)
    Hit hit = Hit(source, d, -1, NULL);
    // for every object, check if ray hits it
    for(int i=0; i<numObjects; i++) {
        float t = objects[i]->Intersect(source, d);
        // 1. only use hits visible for the camera
        // 2. only overwrite hit if either there is
        //     no hit yet, or the hit is closer
        if(t>0.00001
              && (hit.object==NULL || t<hit.t))
            hit = Hit(source, d, t, objects[i]);
    }
    return hit;
}
```

```
Color shade(Hit hit) {
    // if no object was hit,
    // return background color
    if(hit.object==NULL)
        return background;
    // otherwise use diffuse object color
    return hit.object->diffuse;
}
```

- Both in `RayTracer.cpp`
- `intersect` finds the hit of a ray with an object closest to the camera
- If hit.object==NULL, then the ray did not hit anything





LIGHTS AND SHADOWS

Diagram thanks to Henrik

Setting Up Point Lights

Light.h

```
#pragma once
#include "Color.h"
#include "Vector.h"

class Light {
public:
    Light(void);  ~Light(void);
    Vector position;
    Color ambient, diffuse, specular;
};
```

Light.cpp

```
#include "Light.h"

Light::Light(void) { }

Light::~Light(void) { }
```

Set up lights in global variable:

```
const int numLights = 2;
Light* lights[numLights];
```

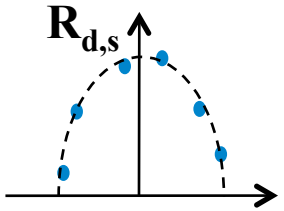
Code for setting up a light:

```
Light* l = new Light();
l->position = Vector(-2, 2, 2);
l->ambient = Color(0.1, 0.1, 0.1);
l->diffuse = Color(0.5, 0.5, 0.5);
l->specular = Color(0.5, 0.5, 0.5);
lights[0] = l;

// similar for second light…
```

Adding Phong Illumination to `shade`

$$\mathbf{R} = \mathbf{I_a}\,\rho_a + \left(\mathbf{I_d}\,\rho_d \frac{s \cdot m}{|s||m|} + \mathbf{I_s}\,\rho_s \left(\frac{h \cdot m}{|h||m|} \right)^{\alpha} \right) \Big/ \left(k_c + k_l d + k_q d^2 \right)$$

```
Color shade(Hit hit) {
    if(hit.object==NULL) return background;
    Color color = Color(0,0,0);
    for(int i=0; i<numLights; i++) {
        // ambient reflection
        color = color + hit.object->ambient
                      * lights[i]->ambient;

        Vector p = hit.HitPoint();
        Vector v = hit.source - p;
        Vector s = ? ;    Vector m = ? ;

        // make sure light hits the front face
        if (s.Dot(m) < 0)  continue;

        // diffuse reflection
        color = color + ? ;

        // specular reflection
        Vector h = ? ;
        color = color + ? ;
    }
    return color;
}
```



Angle between **s** and **m**

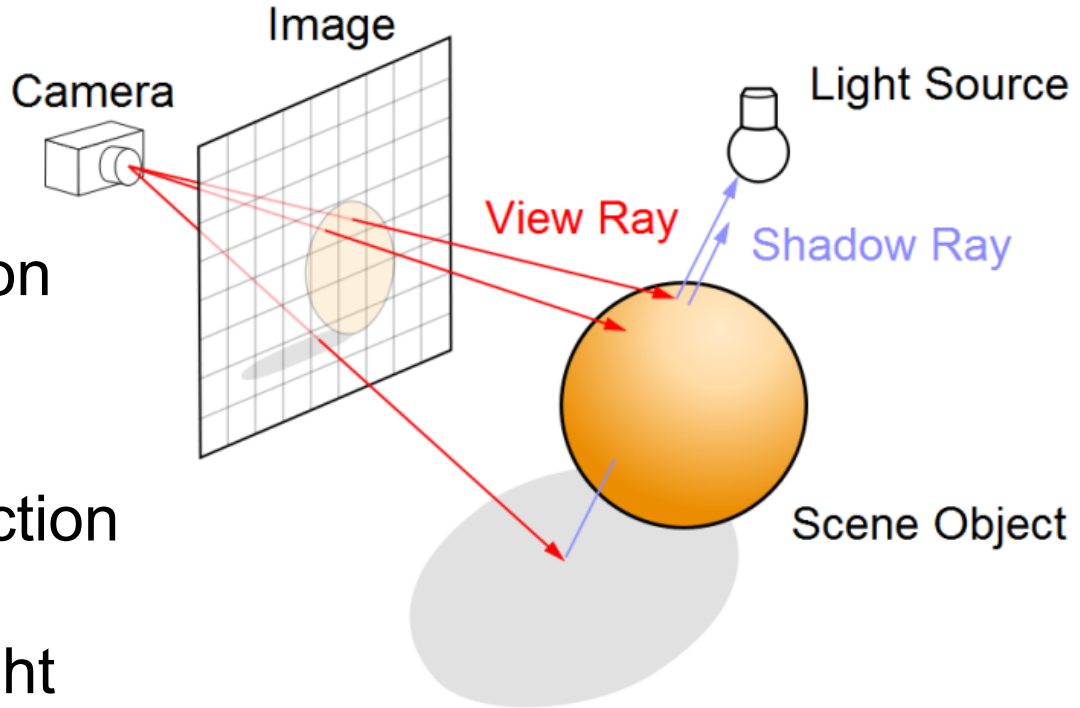
For the assignment, you can omit the division by distance.

Shadow Feelers

Problem: How do we know if a point **p** is in shadow of a light **l** ?

Solution: Check if there is something between **p** and **l**

1. Calculate ($\mathbf{s}_{ource}$, **d**) for a ray that starts at **p** and goes to **l** (a “shadow feeler”)
2. Check if there is an intersection with any scene object (→ use `intersect`)
3. If there is a ray-object intersection between **p** and **l** then:
   do not illuminate **p** with the light
   i.e. do not add R_d and R_s
   Otherwise: normal illumination

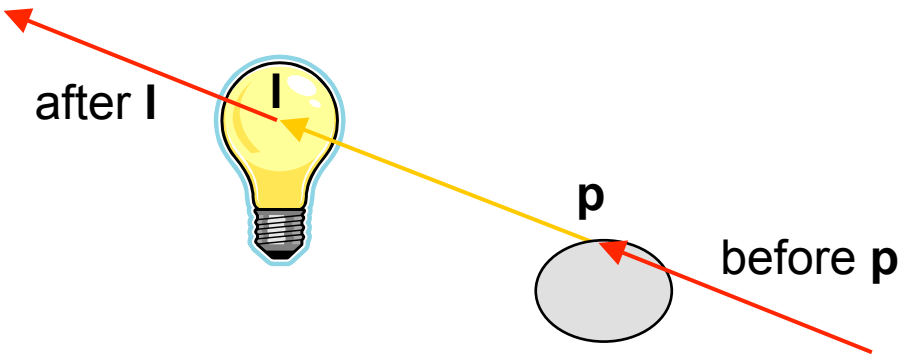


Adding Shadows to `shade`

```
Color shade(Hit hit) {
    if(hit.object==NULL) return background;
    Color color = Color(0,0,0);
    for(int i=0; i<numLights; i++) {
        // ambient reflection …
        // make sure light hits the front face …
        // cast a "shadow feeler“
        Hit feeler = intersect( ? , ? );
        if(“no hit” || “hit before p” || “hit after l”) {
            // diffuse reflection
            color = color + … ;
            // specular reflection
            color = color + … ;
        }
        return color;
}
```

Three different non-shadow cases:

1. “no hit”: no object was hit at all
2. “hit before **p**”: the closest hit is on the other side of p
   (→ no object between **l** and **p**)
3. “hit after **l**”:
   the closest hit is after **l**
   (→ no object between **l** and **p**)



SUMMARY

Summary

1. Define good helper classes (`Vector`, `Color`, `Hit`)
2. Start with basic ray casting algorithm
3. The two most important functions: `intersect` and `shade`
4. Get lights & shadows by using Phong illumination model and "shadow feelers" in function `shade`

References:

- Structure of a Ray Tracer: Hill, Chapter 12.5
- Illuminating Pixels: Hill, Chapter 12.7
- Shadows: Hill, Chapter 12.11

Quiz

1. How does the intersect function in a ray tracer work?
2. What does the shade function in a ray tracer do?
3. How do we integrate Phong illumination into a ray tracer?
4. What is a “shadow feeler” and how can it be used to ray trace shadows?

Cover your right eye and look at the black star. Move slowly towards the screen.
The colored star DISAPPEARS!

And you thought only cars had blindspots!