Computação Interactiva e em GPU

Interactive and GPU Computing

11494: Mestrado em Engenharia Informática

Chap. 2 — Ray Casting

Ray Casting

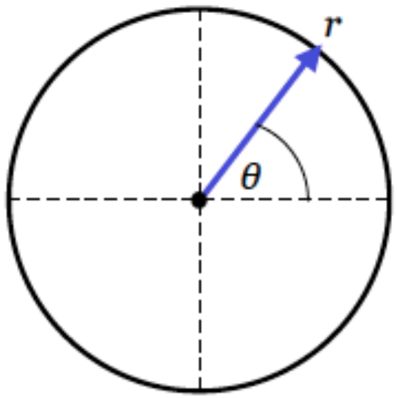
Outline

...:

- Parametric and implicit objects: a reminder.
- Implicit surfaces
- Ray casting: the basic idea
- Constructing rays through pixels
- Finding intersection points between rays and objects
- Pixel color computation: Lambertian model reminder

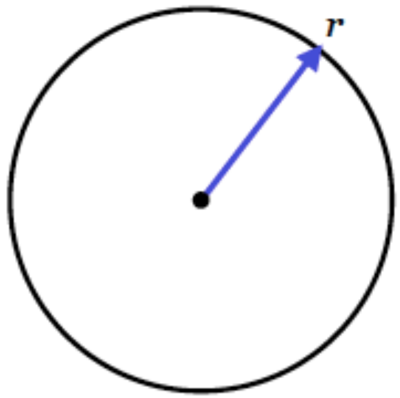
Parametric and implicit objects: a short reminder

parametric circle



$$\begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases}$$

implicit circle



$$f(x,y) = x^2 + y^2 - r^2 = 0$$

$$f(x,y) < 0 \qquad (inside)$$

$$f(x,y) > 0 \qquad (outside)$$

IMPLICIT SURFACES

Implicit surfaces

Definition:

- An implicit surface is a *zero* set of a function: $f(x,y,z)=0$
- *Example*: the radius-r sphere $f(x,y,z)=x^2+y^2+z^2-r^2=0$

Other designations:

- Isosurface / Level set

Unit surface normal:

- It is the *normalized* gradient vector

$$\vec{n}=\frac{\nabla f}{\|\nabla f\|} \quad where \quad \nabla f=\begin{bmatrix}\frac{\partial f}{\partial x}\\ \frac{\partial f}{\partial y}\\ \frac{\partial f}{\partial z}\end{bmatrix}$$

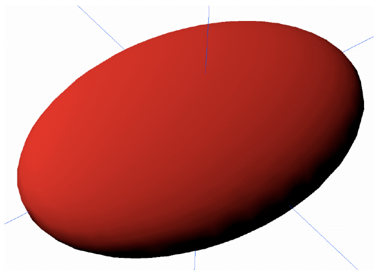
Advantages:

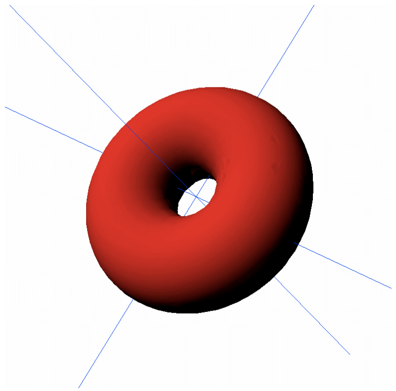
- The entire surface is represented by a single function.
- We can perform interesting operations with this function.
- Example: adding multiple surface functions together

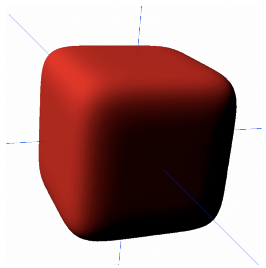
Implicit as solids

Representation of solids:

- Implicit functions represent important classes of solids, which are not necessarily bounded.
- *Example*:
  - Ellipsoid: is a closed, manifold surface that encloses a solid.
- The surface of such a solid is said to be its boundary, which separates the interior from the exterior of the solid.







Quadric surfaces

They are a particular case of implicit surfaces.

http://en.wikipedia.org/wiki/Quadric

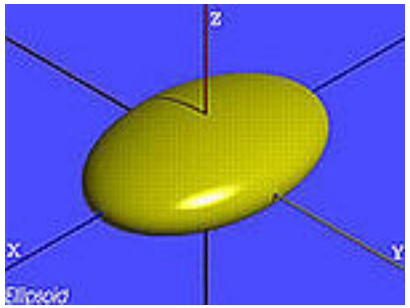
Definition:

– Every quadric surface is defined by the 2nd degree polynomial:

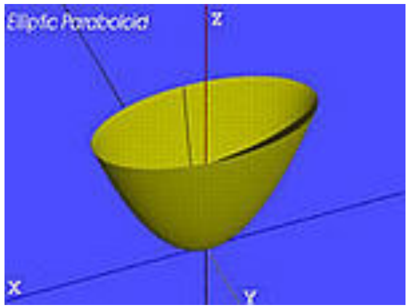
$$f(x,y,z) = Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 + 2Fyz + 2Gy + Hz^2 + 2Iz + J = 0$$

Matrix form:

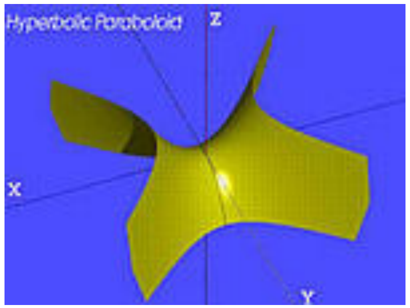
$$f(x,y,z) = v^T M v = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$



$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - z = 0$$



$$\frac{x^2}{a^2} - \frac{y^2}{b^2} - z = 0$$

Gaussian blob surfaces

They are another particular case of implicit surfaces.

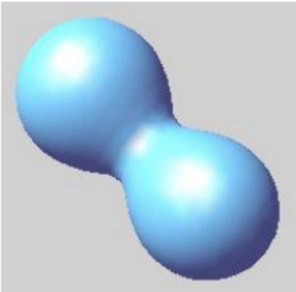
http://en.wikipedia.org/wiki/Gaussian_function

Definition:

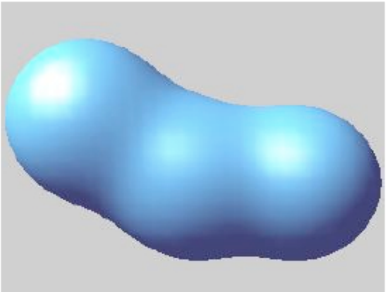
– A Gaussian blob surface is defined by summing up Gaussian functions for a given threshold *T*, each one of which is associated to a point (e.g., center of an atom) in 3D space

– $$f(x,y,z) = \sum_{i=1}^{N} f_i - T = 0 \quad \text{with} \quad f_i(x,y,z) = ae^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2}+\frac{(y-y_0)^2}{2\sigma_y^2}+\frac{(z-z_0)^2}{2\sigma_z^2}\right)}$$

$N=1$



$N=2$

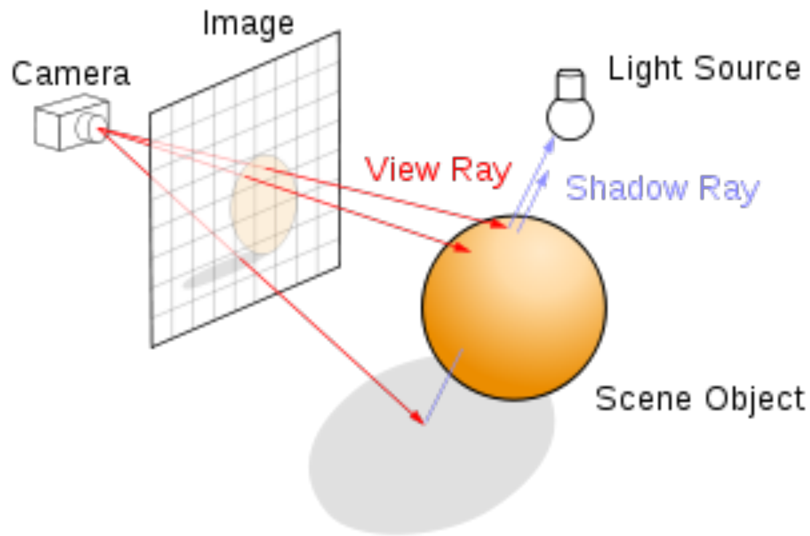


$N=3$

RAY CASTING

Ray casting

*Arthur **Appel** (1968). Some techniques for shading machine rendering of solids. AFIPS Conference Proc. 32, pp. 37-45.*



http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Key idea:

- The idea behind ray casting is to shoot rays from the eye, one per pixel, and find the closest object blocking the path of that ray.
- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces.

Algorithm:

- For each pixel
  - Calculate ray from viewer point through pixel
  - Find intersection points with scene objects (e.g., a sphere)
  - Calculate the color at the intersection point near to viewer (e.g., Phong illumination model)

Step 1: Constructing a ray through each pixel

Equation of the ray passing through a pixel:

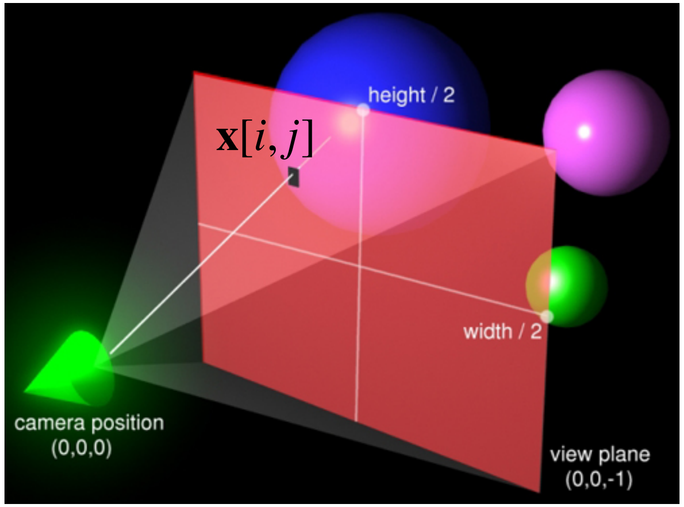
$$\mathbf{x}(t) = \mathbf{o} + t\mathbf{v}$$

where:

- **o** is the camera (eye) position;
- **v** is the vector that stands for the direction of the ray starting at **o** and passing through pixel (i,j):

$$\mathbf{v} = \frac{\mathbf{x} - \mathbf{o}}{\|\mathbf{x} - \mathbf{o}\|}$$

where **x** is the float-point location of the window corresponding to the pixel (i,j) of a discrete view screen (in the view plane) of resolution (W,H):



http://www.unknownroad.com/rtfm/graphics/rt_eyerays.html

Step 1: Constructing a ray through each pixel (cont'd)

<u>Side</u> view of camera at o:

- Position of the i-th pixel **x**[i]?
- Let us first agree that:

$$P_0 = \vec{o} + d\,\vec{l} - d\tan(\theta)\vec{u}$$

$$P_1 = \vec{o} + d\,\vec{l} + d\tan(\theta)\vec{u}$$

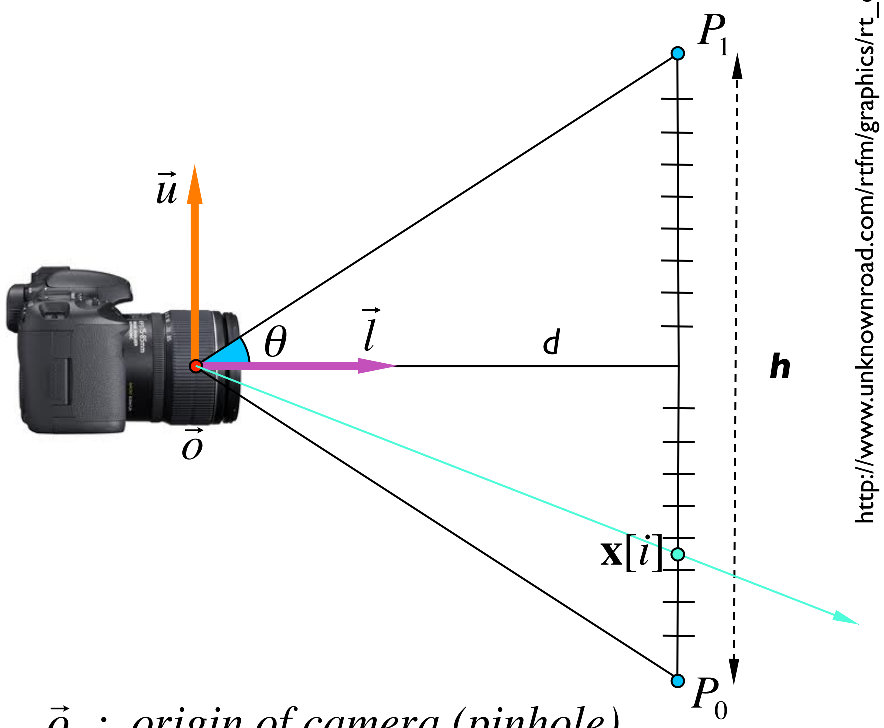
$$h = 2d\tan(\theta)$$

- Also:

$$\mathrm{x}[i] = \frac{i+0.5}{H}(P_1 - P_0)$$

so

$$\mathrm{x}[i] = \frac{i+0.5}{H}h\vec{u}$$



$\vec{o}$: *origin of camera (pinhole)*

$\vec{l}$: *look vector*

$\vec{u}$: *up vector*

H : *discrete height of screen (in pixels)*

h : *height of screen*

d : *distance to screen*

θ : *field of view (FOV) or frustum halfangle*

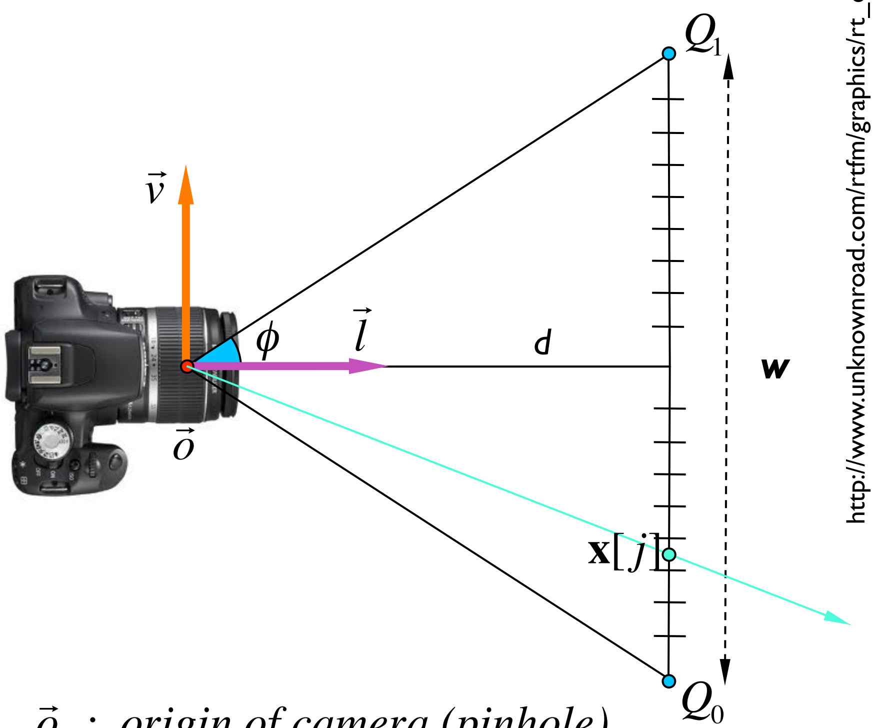
http://www.unknownroad.com/rtfm/graphics/rt_eyerays.html

Step 1: Constructing a ray through each pixel (cont'd)

Top view of camera at o:

- Position of the j-th pixel **x**[j]?
- Analogously, we have:

$$\mathrm{x}[j] = \frac{j+0.5}{W} w\vec{v}$$



$\vec{o}$: *origin of camera (pinhole)*

$\vec{l}$: *look vector*

$\vec{u}$: *up/side vector*

W : *discrete width of screen (in pixels)*

w : *width of screen*

d : *distance to screen*

ϕ : *field of view (FOV) or frustum halfangle*

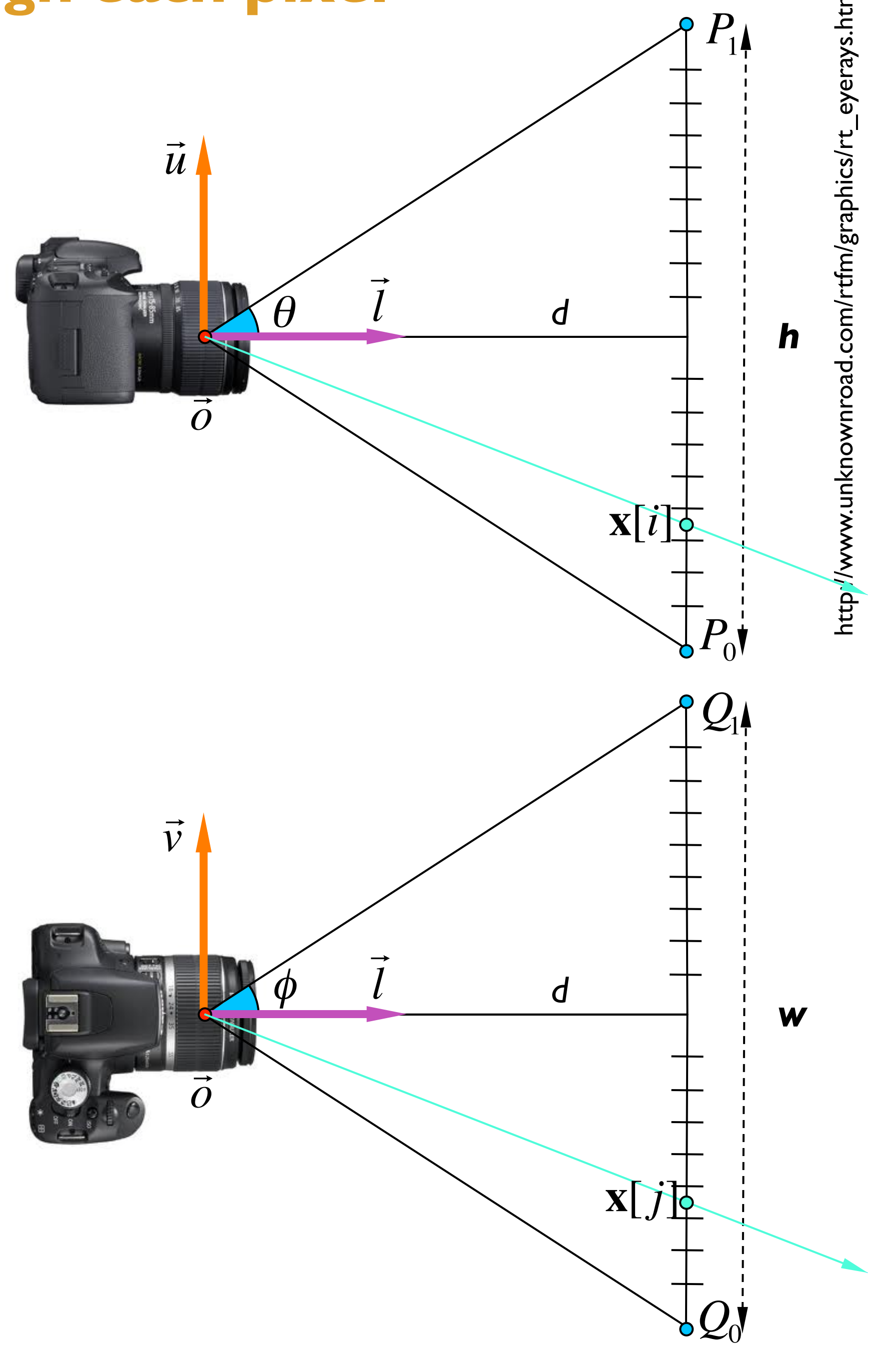
http://www.unknownroad.com/rtfm/graphics/rt_eyerays.html

Step 1: Constructing a ray through each pixel (cont'd)

In conclusion:

- The equation of the ray through each pixel (i,j) is given by:

$$x[i,j] = \vec{o} + \frac{i+0.5}{H} h\vec{u} + \frac{j+0.5}{W} w\vec{v}$$



Step 2: Finding intersection points between rays and implicitly-defined objects

General algorithm:

- Given the equation of the ray:

$$\mathbf{x}(t) = \mathbf{o} + t\mathbf{v}$$

- Given a surface in implicit form *f(x,y,z)*=0:
  - Plane $f(\mathrm{x}) = ax + by + cz + d = \mathbf{n} \bullet \mathbf{x} + d = 0, \quad with \quad \mathbf{n} = (a,b,c) \quad and \quad \mathbf{x} = (x,y,z)$
  - Sphere $f(\mathbf{x}) = x^2 + y^2 + z^2 - 1 = 0$
  - Cylinder $f(\mathrm{x}) = x^2 + y^2 - 1 = 0 \quad and \quad 0 < z < 1$

- We know that all points on the surface satisfy *f(x,y,z)*=0.

- Therefore, for a ray **x**(*t*) to intersect the surface, we have to solve:

$$f(\mathbf{x}(t)) = f(\mathbf{o} + t\mathbf{v}) = 0$$

Ray-plane intersection

Algorithm:

- Given the equation of a generic ray:

$$\mathbf{x}(t) = \mathbf{x}_0 + t\mathbf{v}$$

- Given the equation of the plane:

$$f(\mathbf{x}) = ax + by + cz + d = \mathbf{n} \bullet \mathbf{x} + d = 0$$

where

- **n** is the normal to the plane
- *d* is the distance of the plane from the origin

- Substituting and solving for t, we obtain:

$$f(\mathbf{x}(t)) = f(\mathbf{x}_0 + t\mathbf{v}) = 0 \quad \text{or} \quad f(\mathbf{x}(t)) = \mathbf{n} \bullet (\mathbf{x}_0 + t\mathbf{v}) + d = 0$$

so, the ray hits the plane at

$$t = \frac{-(\mathbf{n} \bullet \mathbf{x}_0 + d)}{\mathbf{n} \bullet \mathbf{v}}$$

Ray-triangle intersection

Tomas Möller and Ben Trumbore, "Fast, minimum storage ray-triangle intersection", Journal of Graphics Tools, 2(1):21-28, 1997

Algorithm:

– Given the equation of a generic ray:

$$\mathbf{x}(t) = \mathbf{o} + t\mathbf{v}$$

– Given the equation:

$$\mathbf{x} = (1 - u - v)\mathbf{x}_0 + u\mathbf{x}_1 + v\mathbf{x}_2 \qquad u,v \geq 0, \;\; u + v \leq 1$$

that expresses $\mathbf{x}$ in *barycentric coordinates* (u,v) as a point in a triangle with vertices $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$

– If the intersection point belongs to both ray line and triangle, we have:

$$\mathbf{x}(t) = \mathbf{o} + t\mathbf{v} = (1 - u - v)\mathbf{x}_0 + u\mathbf{x}_1 + v\mathbf{x}_2$$

– Thus, solve the previous equation system for (t,u,v) in terms of (x,y,z).

Testing ray-triangle intersection

- *R. J. Segura, F. R. Feito,"Algorithms to test Ray-triangle Intersection Comparative Study", WSCG 2001.*

```
#define crossProduct(a,b,c) \
    (a)[0] = (b)[1] * (c)[2] - (c)[1] * (b)[2]; \
    (a)[1] = (b)[2] * (c)[0] - (c)[2] * (b)[0]; \
    (a)[2] = (b)[0] * (c)[1] - (c)[0] * (b)[1];
```

```
#define innerProduct(v,q) \
        ((v)[0] * (q)[0] + \
        (v)[1] * (q)[1] + \
        (v)[2] * (q)[2])
```

```
/* a = b - c */
#define vector(a,b,c) \
        (a)[0] = (b)[0] - (c)[0];     \
        (a)[1] = (b)[1] - (c)[1];     \
        (a)[2] = (b)[2] - (c)[2];


int rayIntersectsTriangle(float *p, float *d,
                          float *v0, float *v1, float *v2) {

    float e1[3],e2[3],h[3],s[3],q[3];
    float a,f,u,v;

    vector(e1,v1,v0);
    vector(e2,v2,v0);
    crossProduct(h,d,e2);
    a = innerProduct(e1,h);

    if (a > -0.00001 && a < 0.00001)
        return(false);

    f = 1/a;
    vector(s,p,v0);
    u = f * (innerProduct(s,h));

    if (u < 0.0 || u > 1.0)
        return(false);

    crossProduct(q,s,e1);
    v = f * innerProduct(d,q);
    if (v < 0.0 || u + v > 1.0)
        return(false);

    return(true);
}
```

Ray-sphere intersection

Algorithm:

- Implicit form of sphere given center *(a,b,c)* and radius **r**:

$$\|\mathbf{x}-\mathbf{c}\|^2 = r^2 \qquad \mathbf{x} = (x,y,z), \quad \mathbf{c} = (a,b,c)$$

- Intersection with the ray $\mathbf{x}(t)=\mathbf{o}+t\mathbf{v}$ gives:

$$\|\mathbf{o}+t\mathbf{v}-\mathbf{c}\|^2 = r^2$$

- Taking into account the identity $\|\mathbf{a}+\mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 + 2(\mathbf{a}\bullet\mathbf{b})$
  - the intersection is a quadratic equation in *t*:

$$\|\mathbf{o}+t\mathbf{v}-\mathbf{c}\|^2 - \mathbf{r}^2 = t^2\|\mathbf{v}\|^2 + 2t\mathbf{v}\bullet(\mathbf{o}-\mathbf{c}) + (\|\mathbf{o}-\mathbf{c}\|^2 - r^2)$$

- Solving for *t*:

$$t = \frac{-\mathbf{v}\bullet(\mathbf{o}-\mathbf{c}) \pm \sqrt{(\mathbf{v}\bullet(\mathbf{o}-\mathbf{c}))^2 - \|\mathbf{v}\|^2(\|\mathbf{o}-\mathbf{c}\|^2 - r^2)}}{\|\mathbf{v}\|^2}$$

  - Real solutions indicate one point (tangent point) or two intersection points
  - Negative solutions are behind the eye
  - If discriminant is negative, the ray misses the sphere

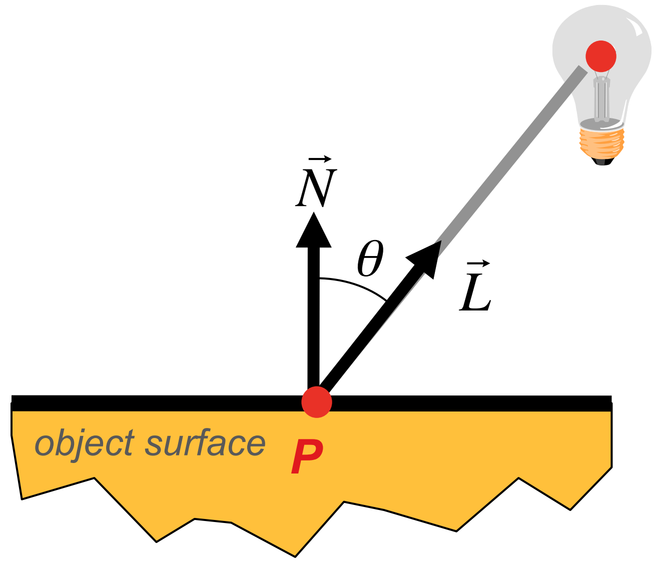
Pixel color computation

Two major possibilities:

- The Lambertian/Phong illumination model of OpenGL.
  - It does not require implementation because it already comes with OpenGL.
- The Lambertian/Phong illumination output to a pixel matrix of some image file format (e.g., TGA)
  - It requires implementation.

Pixel color computation: diffuse light

$$I_D = K_D(\vec{N} \bullet \vec{L})I$$



Lambertian model for diffuse light:

- Specify how material reflects light:
- Bind material to each object in a scene
- Calculates the Lambert coefficient at each hit point of the object surface.
- Calculates the color at each hit point of the object surface:

```
// list of lights
Light
{
  Position = 0.0, 240.0, -100.0;
  Intensity = 1.0, 1.0, 1.0 ;
}
```

```
// list of materials
Material
{
  Id = 0;
  Diffuse = 1.0, 1.0, 0.0;
  Reflection = 0.5;
}
```

```
// list of objects
Sphere
{
  Center = 20.0, 20.0, 0.0;
  Radius = 9.0;
  Material.Id = 0;
}
```

Reflection coefficient

Normal vector

Light vector

```
// calculates the Lambert coefficient
float K = K_D * (N•L);
// calculates the RGB color
red += K * Light.red * Material.red;
green += K * Light.green * Material.green;
blue += K * Light.blue * Material.blue;
```

Ray casting algorithm: review

Algorithm:

- Define the objects and light sources in the scene
- Set up the camera
- `for (int i=0; i<nRows; i++)`
- `    for (int j=0; j<nCols; j++)`

1. Build the (i,j)th ray
2. Find intersections of the (i,j)th ray with the objects in the scene
3. Identify the intersection that lies closest to, and in front of, the eye
4. Compute the hit point where the ray hits this object, and the normal at that point
5. Find the color of the light returning to the eye along the ray from the hit point
6. Place the color at the (i,j)th pixel

Courtesy F.S. Hill, "Computer Graphics using OpenGL"

Summary:

….:

- Parametric and implicit objects: a reminder.
- Implicit surfaces
- Ray casting: the basic idea
- Constructing rays through pixels
- Finding intersection points between rays and objects
- Pixel color computation: Lambertian model reminder