《Fundamentals of Computer Graphics》

Lecture 8、Ray tracing

Part 2: Intersection

Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

Material by S.M.Lea (UNC)

Shade()

- Shade() must determine whether the ray hits some object, and if so, which one. To do this, it uses a routine getFirstHit(ray, best), which puts data about the first hit into an *intersection record* named best.
- Once information about the first hit is available, shade() finds the color of the ray.
- If no object was hit this is simply the background color, which shade() returns.
- If the ray did hit an object, shade() accumulates the various contributions in the variable color. These are the colors emitted by the object if it is glowing, the ambient, diffuse, and specular components that are part of the classical shading model, and any light reflected from a shiny surface or refracted through a transparent surface.

Shade() Skeleton

```
Color3 Scene :: shade (Ray& ray)
{     // return the color of this ray
   Color3 color;               // total color to be returned
   Intersection best;          //data for the best hit so far
   getFirstHit(ray, best);     //fill the ‘best’ record
   if (best.numHits == 0)      //did the ray miss every object?
                               // return background;
   color.set(the emissive color of the object);
   color.add(ambient, diffuse and specular components);
   //add more contributions
   color.add(reflected and refracted components);
   return color;}
```

Intersection Class

- The routine getFirstHit() finds the object hit first by the ray and returns the information in the intersection record best.
- We implement intersection records using the class Intersection which is given in pseudo code by:

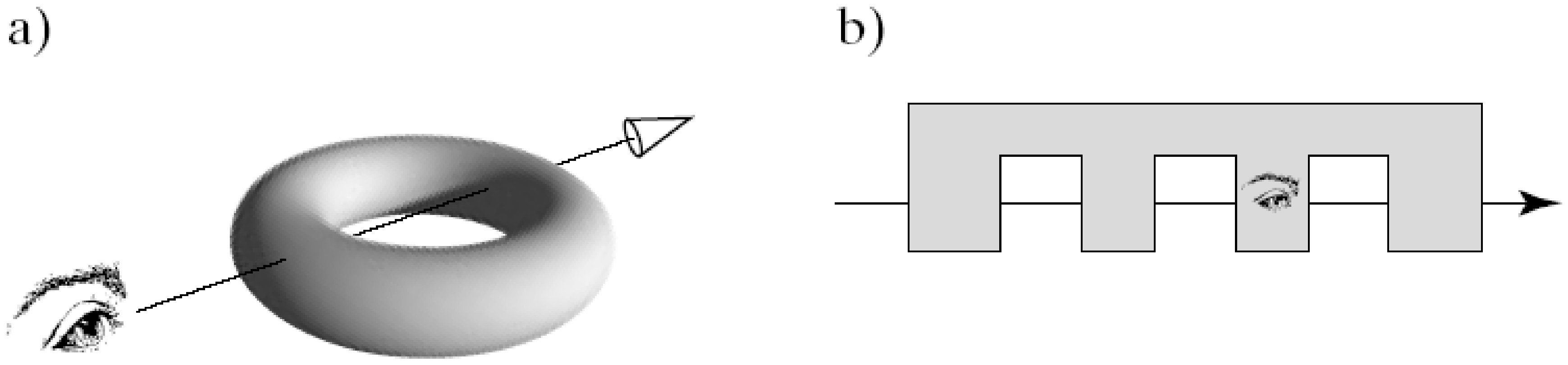
```
class Intersection{
 public:
   int numHits;        // # of hits at positive hit times
   HitInfo hit[8];     // list of hits – may need more than 8
                                                      later
   …various methods…
};
```

Intersections of Rays and Objects

- We are particularly interested in the first hit of the ray with an object. If inter is an intersection record and inter.numHits is greater than 0, information concerning the first hit is stored in inter.hit[0].
- Why keep information on *all* of the hits the ray makes with an object at positive hit times rather than just the first?
- One of the powerful advantages of the ray tracing approach is the ability to render *boolean* objects. To handle booleans, we must keep a record of all the hits a ray makes with an object so we take pains now to set things up properly.

Intersections of Rays and Objects (2)

- Normally the eye is outside of all objects and a ray hits just twice: once upon entering the object and once upon exiting it. In such cases inter.numHits is 2, inter.hit[0] describes where the ray enters the object, and inter.hit[1] describes where it exits.
- But some objects can have more than two hits.



Intersections of Rays and Objects (3)

- In part a of the figure, there are four hits with positive hit times so inter.numHits is 4, and we store hit information in inter.hit[0], …, inter.hit[3].
- In part b, the ray hits the object eight times, but the eye is inside the object (assumed to be transparent) and only three of the hits occur with positive hit times (so inter.numHits is 3).

HitInfo Class

- Information about each hit is stored in a record of type HitInfo whose pseudo code form looks like:

```
class HitInfo { // data for each hit with a surface
public:
 double hitTime;                    // the hit time
 GeomObj* hitObject;                // the object hit
 bool isEntering;          // is the ray entering or exiting?
 int surface;                       // which surface is hit?
 Point3 hitPoint;          // hit point
 Vector3 hitNormal;                 // normal at hit point
 …other fields and methods …   };
```

getFirstHit()

- The routine getFirstHit() scans through the entire object list starting at pObj, a pointer to the first object in the list, testing whether the ray hits the object.
- To do this, it uses each object’s own hit() method. Each hit() method returns true if there is a legitimate hit and false otherwise.
- If there is a hit, the method builds an entire intersection record describing all of the hits with this object and places it in inter.
- Then getFirstHit() compares the first positive hit time in inter with that of best, the record of the best so far hit.
- If an earlier hit time is found, the data in inter is copied into best. The value of best.numHits is initialized to 0 so that the first real hit will be counted properly as pObj loops through the object list.

getFirstHit() Code

```
void Scene:: getFirstHit(Ray& ray, Intersection& best)
{  Intersection inter;            // make intersection record
   best.numHits = 0;                        // no hits yet
   for (GeomObj* pObj = obj; pObj != NULL; pObj =
        pObj->next)
  { // test each object in the scene
    if(!pObj->hit(ray, inter))    //does the ray hit pObj?
        continue;                 // miss: test the next object
    if(best.numHits == 0 ||                 // best has no hits yet
    inter.hit[0].hitTime < best.hit[0].hitTime)
    best.set(inter);     // copy inter into best}
}
```

getFirstHit() Execution

- Notice that getFirstHit() passes the burden of computing ray intersection onto the hit() routine that each object possesses.
- We shall develop a hit() method for each type of object. For the sake of efficiency, it will be finely tuned to exploit special knowledge of the shape of the generic object.
- This is an excellent example of using polymorphism to simplify code and make it more robust and efficient. The routine hit() is a virtual method of the GeomObj class from which all actual Shape classes are derived.

Hit() Method for a Sphere

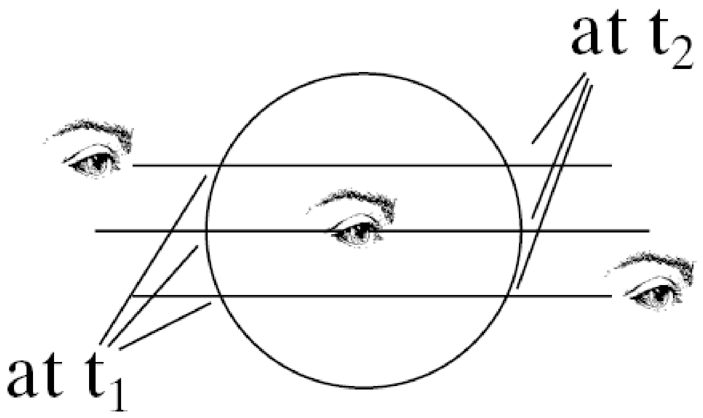
- Code for hit() is in Figure 12.15.
- The hit() method for the Sphere class first transforms the ray r into the generic coordinates of this sphere, using this sphere's particular inverse transformation and xfrmRay().
- Next the coefficients *A*, *B*, and *C* of the quadratic equation are found, and the discriminant $B^2 - AC$ is tested.
- If it is negative, there are no real solutions to the equation, and we know the ray must miss the sphere. Therefore, hit() returns false, and inter is never used.

Hit() Method for a Sphere (2)

- On the other hand, if the discriminant is positive, the two hit times are computed. Call the earlier hit time t_1 and the later one t_2.
- There are three possibilities:
  - The sphere can be in front of the eye, in which case both hit times are positive.
  - The eye can be inside the sphere, in which case t_1 is negative but t_2 is positive
  - The sphere can be behind the eye, so that both times are negative.

Hit() Method for a Sphere (3)

- If t_1 is strictly positive, data for the first hit are placed in inter.hit[0], and variable num is set to one to indicate there has been a hit.
- If t_2 is positive, data for the next hit are placed in inter.hit[num]. (If the first hit time is negative, the second hit time data are automatically placed in hit[0].)



Hit() Method for a Sphere (4)

- The data placed in each hit record use knowledge about a sphere.
- For instance, because the sphere is a convex object, the ray must be entering at the earlier hit time, and exiting at the later hit time.
- The value of surface is set to 0, as there is only one surface for a sphere; we address the issue wherein a ray can hit several possible surfaces later.

Hit() Method for a Sphere (5)

- The points in generic coordinates where the ray hits the sphere are also recorded in the hitPoint field.
- The hit spot is always (by definition) the same as the position of the ray at the given hit time, which is found by the function Point3 rayPos (Ray& r, float t);//returns the ray’s location at time t.
- The calculation of the normal vector at the hit spot is also simple for a sphere; because the normal points outward radially from the center of the sphere, it has coordinates *identical* to the hit point itself.

Complete Ray tracer for Emissive Sphere Scenes

- We have enough tools in place to put together a simple ray tracer for scenes composed of spheres and ellipsoids.
- It is very useful to get this much working before things get more complicated to see how all of the ingredients discussed go together.

Emissive Sphere Raytracer (2)

- To make an object glow, we set its emissive component in the SDL file to some nonzero color, as in emissive 0.3 0.6 0.2
- To adjust Scene:: shade() to handle only emissive light, just remove the color.add() lines,

```
Color3 Scene :: shade(Ray& ray)
{   Color3 color;
    Intersection best;
    getFirstHit(ray, best);
    if (best.numHits == 0) return background;
    Shape* myObj = (Shape*)best.hit[0].hitObject; //the hit
    object
    color.set (myObj->mtrl.emissive);
    return color;  }
```

Emissive Sphere Ray tracer (3)

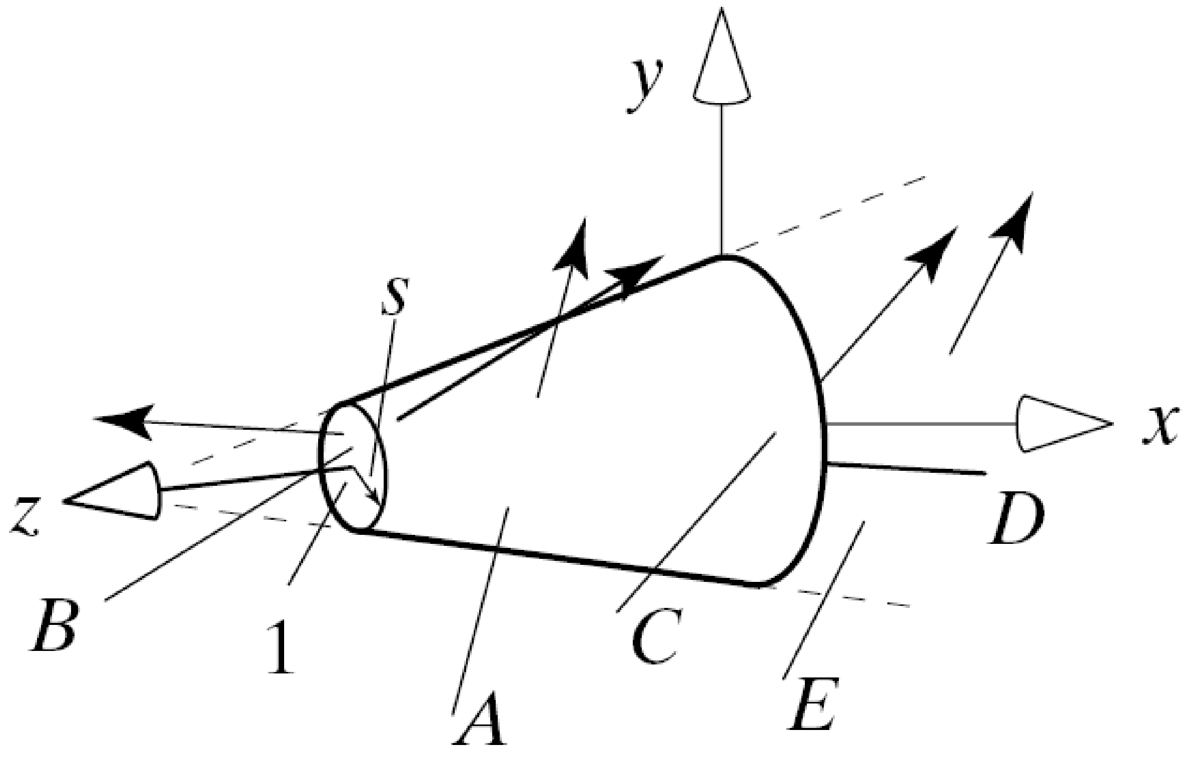
- The only parts of Camera :: raytrace() that need fleshing out are the computation of the ray’s direction for each pixel block.
- Find the parametric form for the ray that passes through the lower left corner of the *i-kth* pixel block; see Equation 12.2.

Intersecting Rays with Squares

- The generic square lies in the *z* = 0 plane and extends from –1 to 1 in both *x* and *y*.
- Its implicit form is $F(P) = P_z \; for \; |P_x| \le 1 \; and \; |P_y| \le 1$
- hit() first finds where the ray hits the generic plane (Equation 12.10) and then tests whether this hit spot also lies within the square.
- Code is in Figure 12.17.

Intersecting Rays with Tapered Cylinders

- The generic tapered cylinder is shown in the figure, along with several rays. A ray can intersect the cylinder in many ways.



Hit() for Tapered Cylinders

- The side of the cylinder is part of an infinitely long wall with a radius of 1 at *z* = 0 and a small radius of *s* at *z* = 1.
- Implicit form: for 0 < z < 1,

$$F(x,y,z) = x^2 + y^2 - (1 + (s-1)z)^2$$

- If *s* = 1 the shape becomes the generic cylinder; if *s* = 0 it becomes the generic cone.
- We develop a hit() method for the tapered cylinder, which provides as well a hit() method for the cylinder and cone.

Hit() for Tapered Cylinders (2)

- With this many possibilities, we need an organized approach that avoids an unwieldy number of if()..else tests.
- The solution is to identify hits in whatever order is convenient and to put them in the inter.hit[] list regardless of order.
- At the end, if inter.hit[] holds two hits out of order, the two items are swapped.

Hit() for Tapered Cylinders (3)

- To determine whether the ray strikes the infinite wall, substitute *S* + **c***t* into the implicit form for the tapered cylinder to obtain the quadratic equation, $At^2 + 2Bt + C = 0$. It is straightforward to show that [$d = (s - 1)c_z$ and $F = 1 + (s - 1)S_z$]

$$A = c_x^2 + c_y^2 - d^2$$

$$B = S_x c_x + S_y c_y - Fd$$

$$C = S_x^2 + S_y^2 - F^2$$

Hit() for Tapered Cylinders (4)

- If the discriminant B^2 - AC is negative, the ray passes by the tapered cylinder’s wall.
- If the discriminant is not negative, the ray does strike the wall, and the hit times can be found by solving the quadratic equation.
- To test whether each hit is on the actual cylinder wall, find the z-component of the hit spot. The ray hits the cylinder only if the z-component lies between 0 and 1.

Hit() for Tapered Cylinders (5)

- To test for an intersection with the base, intersect the ray with the plane $z = 0$. Suppose it hits at the point $(x, y, 0)$. The hit spot lies within the cap if $x^2+y^2 < 1$.
- Similarly, to test for an intersection with the cap, intersect the ray with the plane $z = 1$. Suppose it hits at the point $(x, y, 1)$. The hit spot lies within the cap if $x^2+y^2 < s^2$.

Hit() for Tapered Cylinders (6)

- A cylinder has more than one surface, and we will later want to know which surface is hit.
  - For instance, we may want to paste a different texture on the wall than on the cap.
- Therefore we adopt the following numbering: the wall is surface 0, the base is surface 1, and the cap is surface 2.
- The appropriate value is placed in the surface field of each hit record.
- Code is in Figure 12.19.

Hit() for Tapered Cylinders (7)

- The normal vector must be found at the two hit points.
- The normal to the cylinder wall at point (x, y, z) is simply (x, y, $-(s - 1)(1+ (s - 1)z)$).
- The normals to the cap and base are (0, 0, 1) and (0, 0, -1), respectively. The hitNormal fields are filled with the appropriate values by hit().

Intersecting a Ray with a Cube or Any Convex Polyhedron

- The **generic cube** is centered at the origin and has corners at ($\pm$1, $\pm$1, $\pm$1), using all eight combinations of +1 and -1.
- Thus its edges are aligned with the coordinate axes, and its six faces lie in the planes specified in the table (next slide).
- The figure also shows the outward pointing normal vector to each plane and a typical point, *spot*, that lies in the plane.

The Six Planes That Define the Generic Cube

Plane	Name	Eqn.	Out Normal	Spot
0	top	y = 1	(0, 1, 0)	(0, 1,0)
1	bottom	y = -1	(0, -1, 0)	(0, -1, 0)
2	right	x = 1	(1, 0, 0)	(1, 0, 0)
3	left	x = -1	(-1, 0, 0)	(-1, 0, 0)
4	front	z = 1	(0, 0, 1)	(0, 0, 1)
5	back	z = -1	(0, 0, -1)	(0, 0, -1)

Importance of the Generic Cube

- A large variety of boxes can be modeled and placed in a scene by applying an affine transformation to a generic cube.
- When ray tracing, each ray can be inverse transformed into the generic cube's coordinate system, and we can use a *ray-with-generic cube* intersection routine which can be made very efficient.
- The generic cube can be used as an **extent** for the other generic primitives in the sense of a **bounding box**: each generic primitive like the cylinder fits snugly inside.
- It is often efficient to test whether a ray intersects the extent of an object before testing whether it hits the object itself, particularly if the ray-with-object intersection is computationally expensive. If a ray misses the extent, it *must* miss the object.

Hit() for the Generic Cube

- The process of intersecting a ray with a cube is essentially the Cyrus-Beck clipping algorithm, in which a line is clipped against a convex window in 2D space.
- The basic idea is that each plane of the cube defines an inside half space and an outside half space. A point on a ray lies inside the cube if and only if it lies on the inside of every half-space of the cube.
- So intersecting a ray with a cube is a matter of finding the interval of time in which the ray lies inside all of the planes of the cube.

Hit() for the Generic Cube (2)

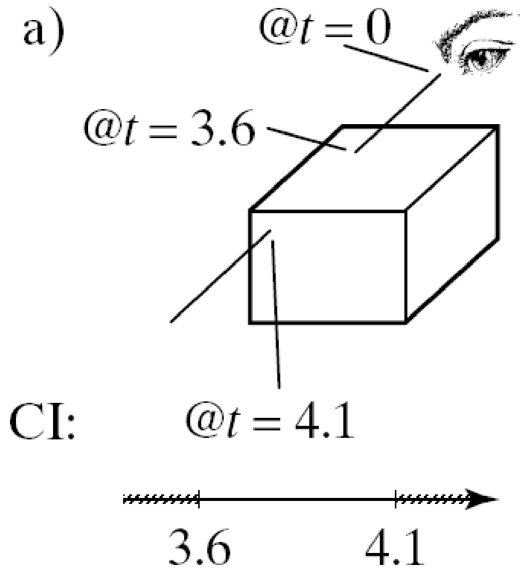
- We test the ray against each of the planes of cube *P* in turn, in some chosen order, computing the time at which the ray either enters or exits the inside half space for that plane.
- We keep track of a *candidate interval*, *CI*: the interval of time in which, based on our tests so far, the ray *could* be inside the object. It is bracketed by the values t_{in} and t_{out}: $CI = [t_{\text{in}}, t_{\text{out}}]$.
- The values t_{in} and t_{out} are “must be outside times”. The understanding is that the ray if it is entering must be outside the relevant plane for all $t < t_{\text{in}}$ and, if it is exiting for all $t > t_{\text{out}}$.

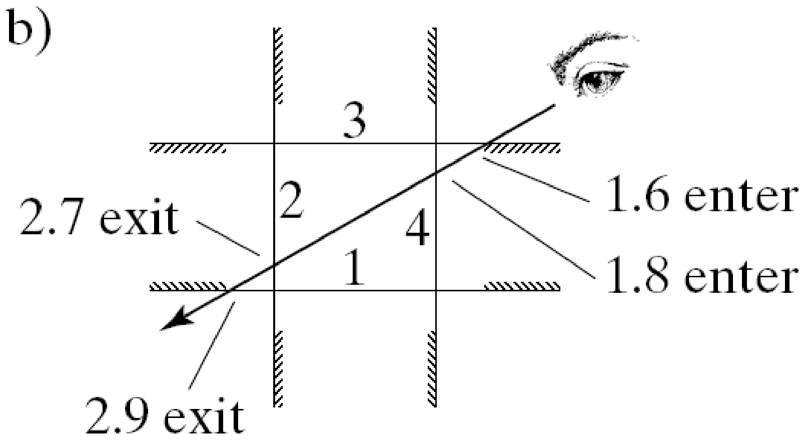
Hit() for the Generic Cube (3)

- As each plane of *P* is tested, we chop away at this interval, either increasing t_{in}, or reducing t_{out}.
- If at any point the *CI* becomes empty, the ray must miss the object, giving an early out.
- If after testing all of the planes of *P* the remaining *CI* is non-empty, the ray enters the object at t_{in} and exits at t_{out}.

Example

- The figure shows the example of a ray entering the generic cube at *t* = 3.6 and exiting at *t* = 4.1. In this example, when testing is complete, the remaining *CI* is [3.6, 4.1] and the ray definitely hits the cube. Part b shows a 2D version for simplicity.





Hit() Pseudocode

- At each step, t_{in} and t_{out} are adjusted according to the following pseudocode algorithm:

```
initialize $t_{in}$ at $-\infty$, and $t_{out}$ at $\infty$ (for each CI)
If (the ray is entering at $t_{hit}$)
    $t_{in} = \max(t_{in}, t_{hit})$
else if (the ray is exiting at $t_{hit}$)
    $t_{out} = \min(t_{out}, t_{hit})$
```

Hit() for the Generic Cube (4)

- Let the ray be *S* + **c***t* and suppose the plane in question has outward-pointing normal **m** and contains the point *B*.
- The implicit form for this plane is

$$F(P) = \mathbf{m} \cdot (P - B)$$

- so the ray hits it when $\mathbf{m} \cdot (S + \mathbf{c}t - B) = 0$
- or at the hit time $t = \dfrac{numer}{denom}$
- where $numer = \mathbf{m} \cdot (B - S)$

$$denom = \mathbf{m} \cdot \mathbf{c}$$

Hit() for the Generic Cube (4)

- The ray is passing into the outside half space of the plane if *denom* > 0 (since then **m** and **c** are less than 90° apart), and passing into the inside half space if *denom* < 0.
- If *denom* $= 0$ the ray is parallel to the plane, and the value of *numer* determines whether it lies wholly inside or wholly outside.
- Code is in Fig. 12.23.

Hit() Conditions (2D): Summary

Situation	**Condition**
Pass to inside	Denom < 0
Pass to outside	Denom > 0
Wholly inside	Denom = 0, numer > 0
Wholly outside	Denom = 0, numer < 0

Hit() Conditions (3D): Summary

Plane	numer	denom
0	$1 - S_y$	c_y
1	$1 + S_y$	$-c_y$
2	$1 – S_z$	c_z
3	$1 + S_z$	$-c_z$
4	$1 - S_x$	cx
5	$1 + S_x$	$-c_x$

Hit() for the Generic Cube (5)

- When all planes have been tested, we know the hit times t_{In} and t_{Out}.
- If t_{In} is positive, the data for the hit at t_{In} is loaded into inter.hit[0], and the data for the hit at t_{out} is loaded into inter.hit[1]. If only t_{out} is positive, the data for its hit are loaded into inter.hit[0].
- The normal vector to each hit surface is set using a helper function cubeNormal(i), which returns the outward normal vector.
- For instance, cubeNormal(0) returns (0,1,0), and cubeNormal(3) returns (-1,0,0).

Intersection for Any Convex Polyhedron

- The extension of hit() for any convex polyhedron is very straightforward.
- Suppose there are *N* bounding planes, and the *i*-th plane contains point B_i and has (outward) normal vector $\mathbf{m}_i$.
- Everything in hit() for the cube remains the same except that numer and denom are found using

$$numer = \mathbf{m} \cdot (B - S)$$

$$denom = \mathbf{m} \cdot \mathbf{c}$$

Intersection for Any Convex Polyhedron (2)

- Now two expensive dot products must be performed, and the for loop becomes:

```
for (int i = 0; i < N; i++) //for each plane of the
   polyhedron
{   numer = dot3(mi, Bi – S);
    denom = dot3(mi, c);
    if(fabs(denom) < eps) … as before
    … same as before ..
}
```

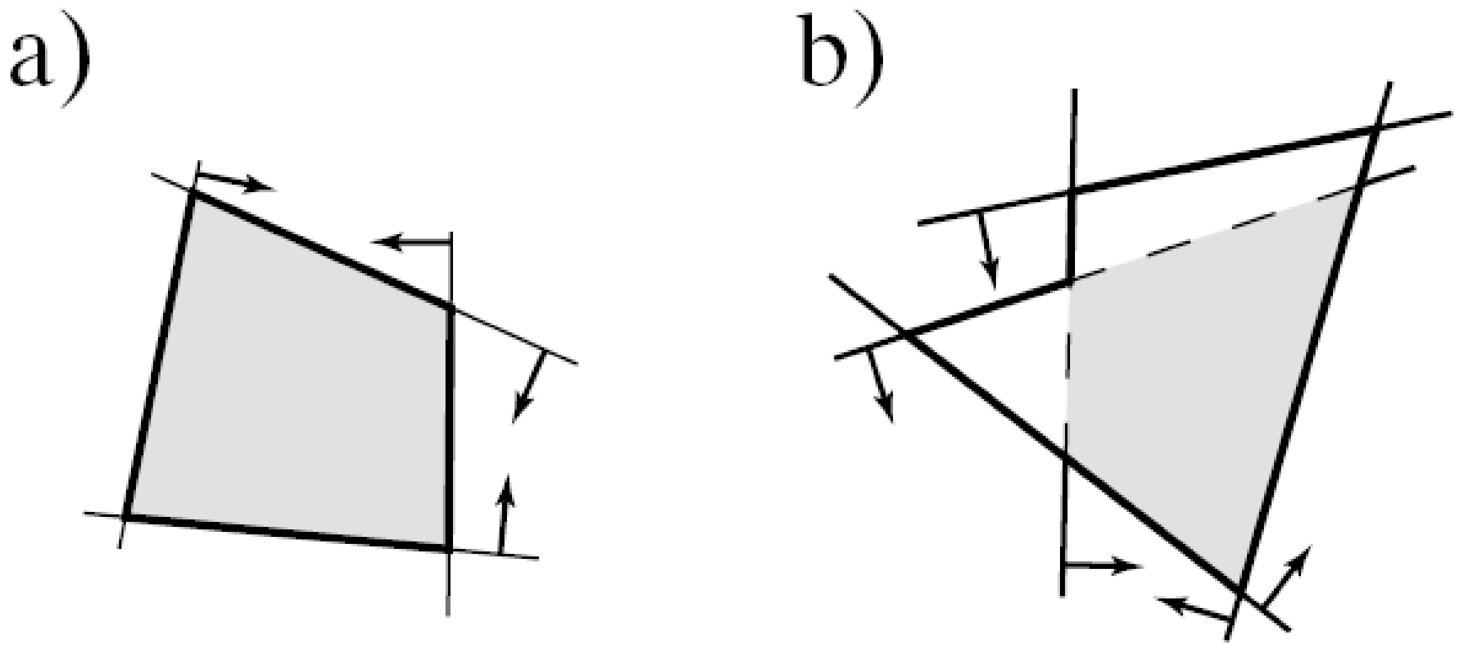
Intersection for a Mesh Object

- Mesh objects are described by a list of faces, and each face is a list of vertices along with a normal vector at each vertex.
- We shall take each face of the mesh in turn and treat it as a bounding plane. Call the plane associated with each face its *face plane*.
- The object that is ray traced, therefore, is the shape that is the intersection of the inside half spaces of each of its face planes.

Intersection for a Mesh Object (2)

- The figure shows two shapes (2D for simplicity) with their face planes marked.
- The object in part a is convex so its face planes are the same as its bounding planes. This shape will be ray traced correctly.
- Part b shows a non-convex object. The portion that is inside all of its face planes is shown shaded. This is what the ray tracer will display!
- So using this method for a mesh will work only if the mesh represents a truly convex object.

Intersection for a Mesh Object (3)



Hit() for a Mesh Object

- To build hit() for a mesh, we form numer and denom for each face in turn.
- We use as the representative point on the face plane the 0-th vertex of the face: pt[face[f].vert[0].vertIndex], and use as the normal to the face plane the normal associated with this 0-th vertex: norm[face[f].vert[0].normIndex].

Intersection with Other Primitives

- We only need to know the implicit form *F*(*P*) of the shape.
- Then, as before, to find where the ray *S* + **c** *t* intersects the surface, we substitute *S* + **c** *t* for *P* in *F*(*P*) forming a function of time *t*: d(t) = F(S + **c**t), which is
  - positive at those values of *t* for which the point on the ray is outside the object,
  - zero when the ray coincides with the surface of the object,
  - negative when the ray is inside the surface.

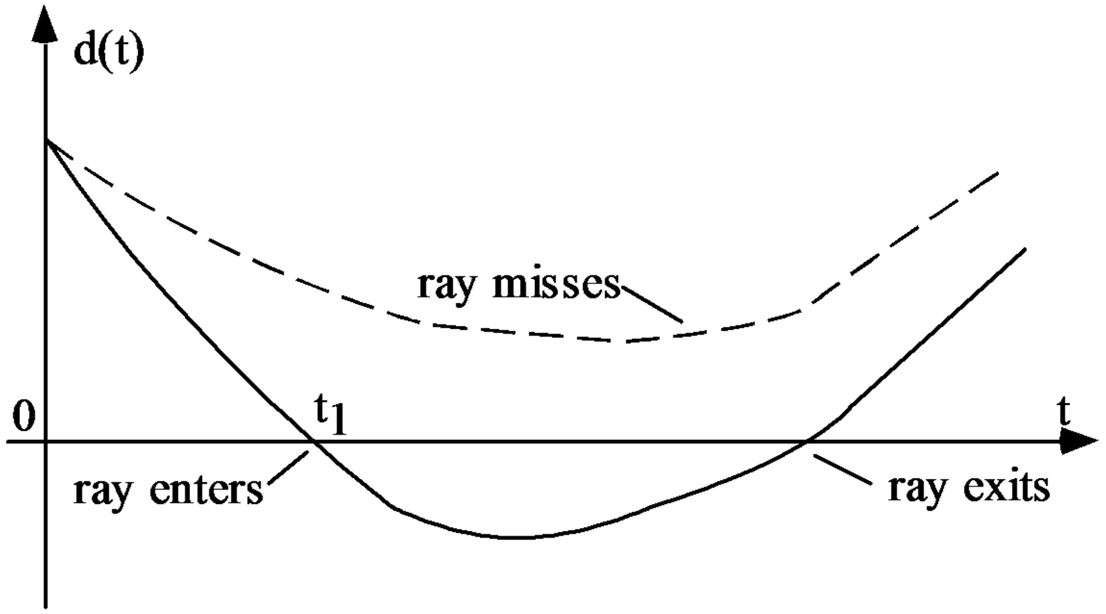
Intersection with Other Primitives (2)

- When seeking intersections, we look for values of *t* that make *d*(*t*) = 0, so intersecting a ray is equivalent to solving this equation.
- For the sphere and other quadric surfaces, we have seen that this leads to a simple quadratic equation.
- A *torus* is a different matter. The generic torus has a quartic (4th order) implicit function,

$$F(P) = (\sqrt{P_x^2 + P_y^2} - d)^2 + P_z^2 - 1$$

Intersection with Other Primitives (3)

- More generally, consider the general shape of *d*(*t*) as *t* increases. If the ray is aimed towards the object in question, and the start point *S* lies outside the object, we get shapes similar to those in the figure.



Intersection with Other Primitives (4)

- The value of $d(0)$ is positive since the ray starts outside the object.
- As the ray approaches the surface, $d(t)$ decreases, reaching 0 if the ray intersects the surface at t_1 in the figure.
- There is a period of time during which the ray is inside the object and $d(t) < 0$.
- When the ray emerges again, $d(t)$ passes through 0 and increases thereafter.
- If instead the ray misses the object (as in the dashed curve), $d(t)$ decreases for a while, but then increases forever without reaching 0.

Intersection with Other Primitives (5)

- For quadrics such as the sphere, $d(t)$ has a parabolic shape, and for the torus, a quartic shape.
- For other surfaces, $d(t)$ may be so complicated that we have to search numerically to locate t's for which $d(.)$ equals 0.
- To find the smallest positive value of t that yields 0, we evaluate $d(t)$ at a sequence of t-values, searching for one that makes $d(t)$ very small. Techniques such as Newton's method provide clever ways to progress towards better and better t-values, but in general these numerical techniques require many iterations.
- This, of course, significantly slows down the ray tracing process.