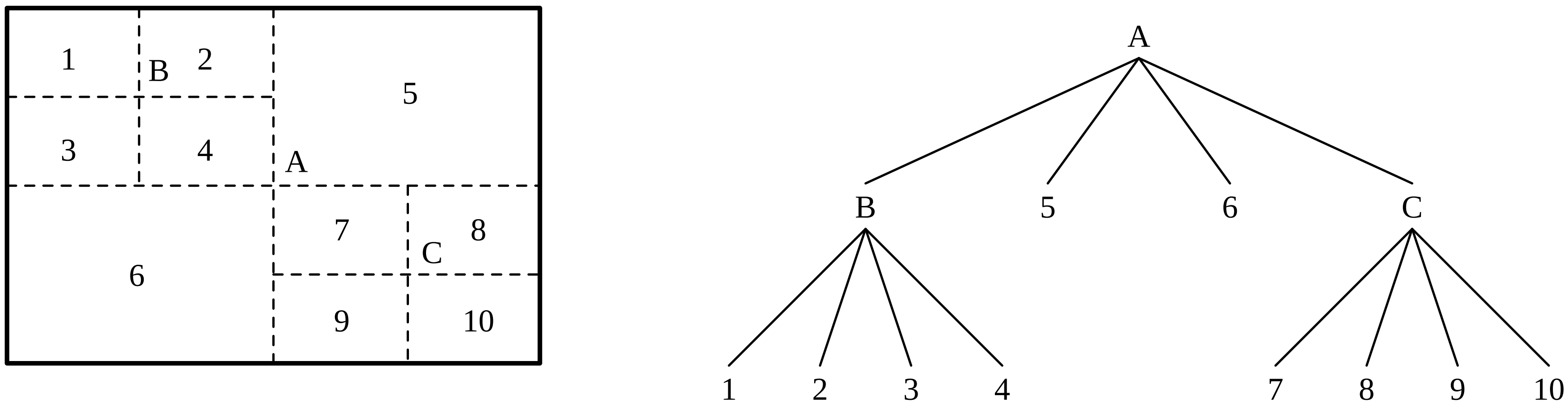
Space Partitioning

- **Quad Trees**. Partition 2-D space. Each node has 4 children.
- **Oct Trees**. Partion 3-D space. Each node has 8 children.
- **Binary Space Partitioning (BSP) Trees**.

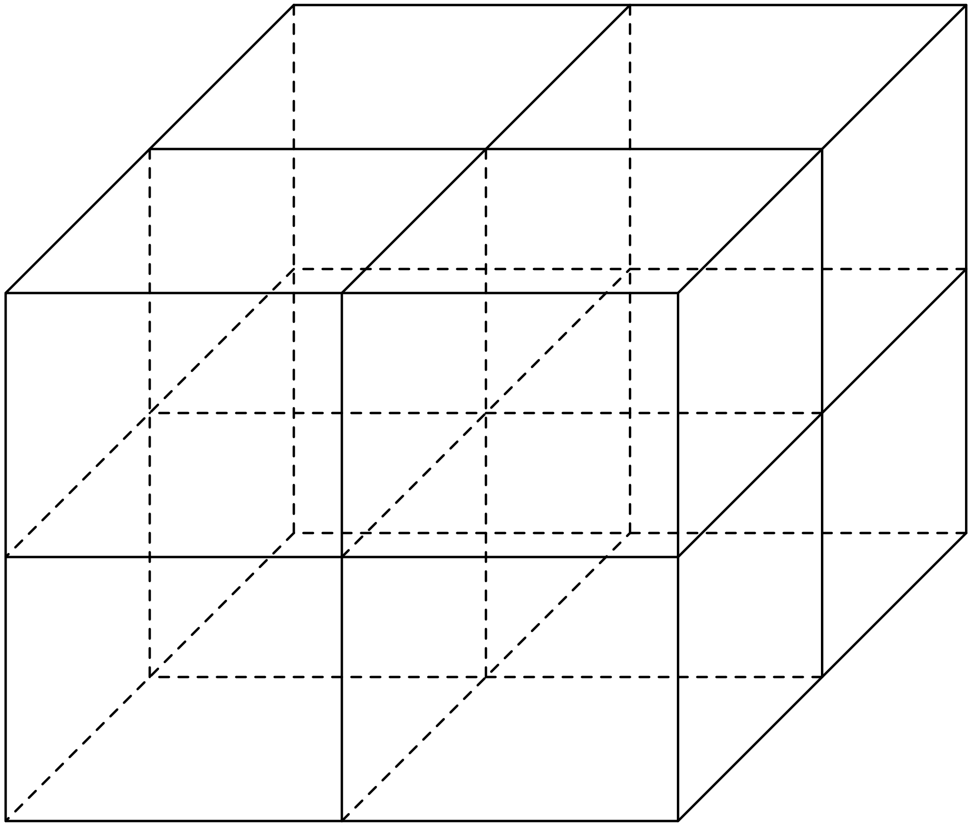
Quad Trees

Partition 2-D space. The Warnock algorithm uses this implicitly.

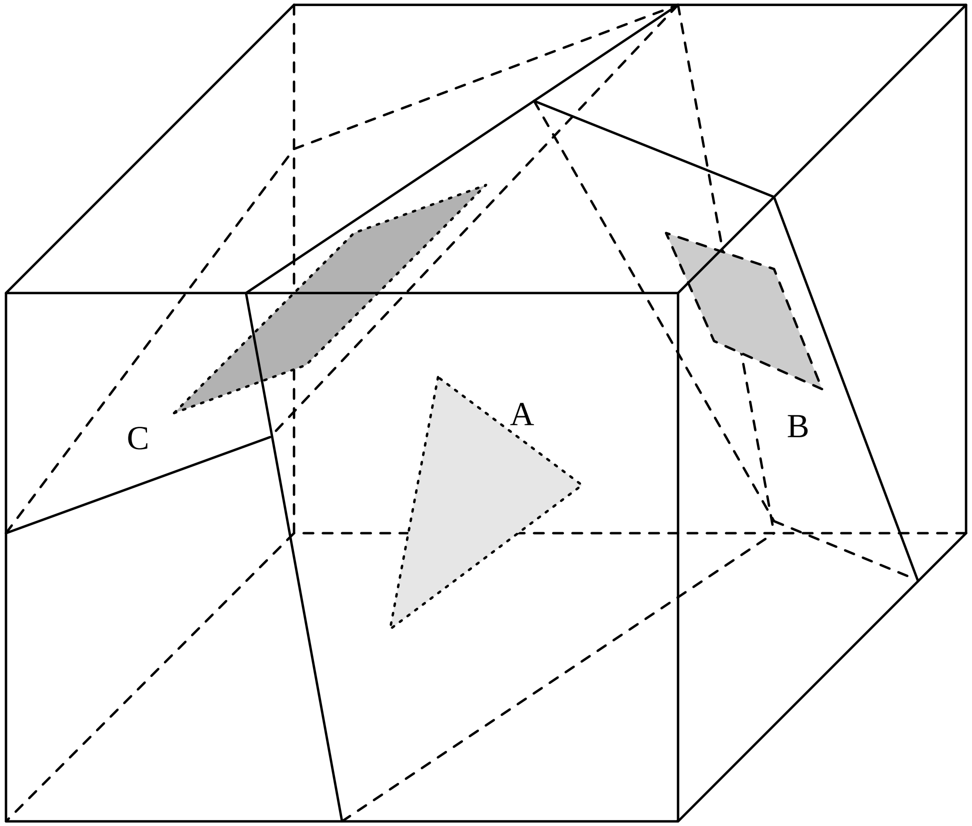


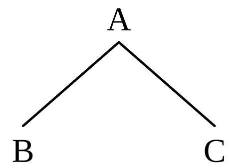
Oct-Trees

Partition 3-D space into octants.



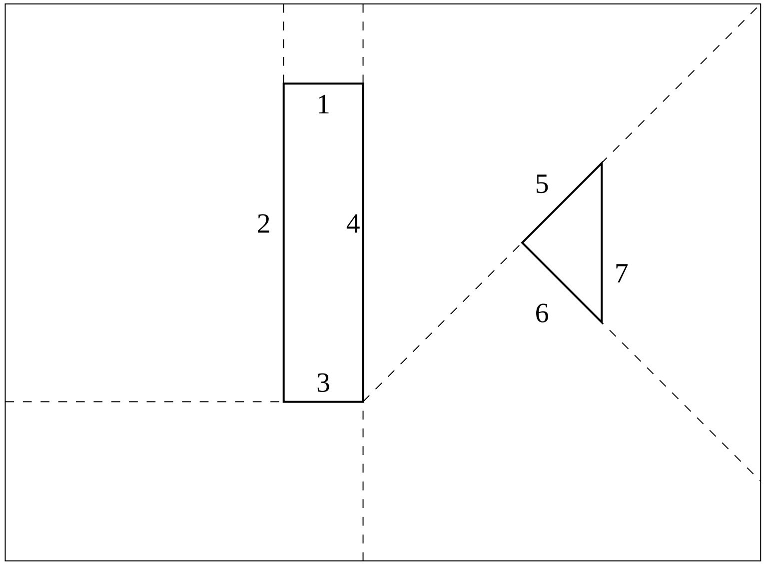
BSP Trees

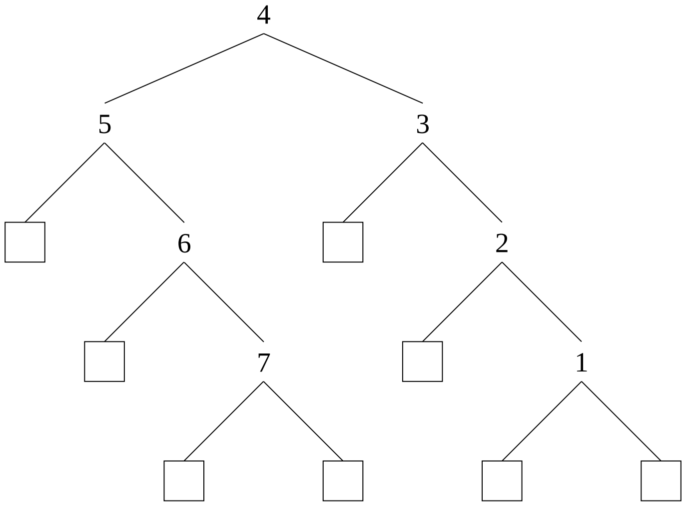




A BSP tree subdivides space along the planes defined by particular polygons.

A BSP Tree in Two Dimensions



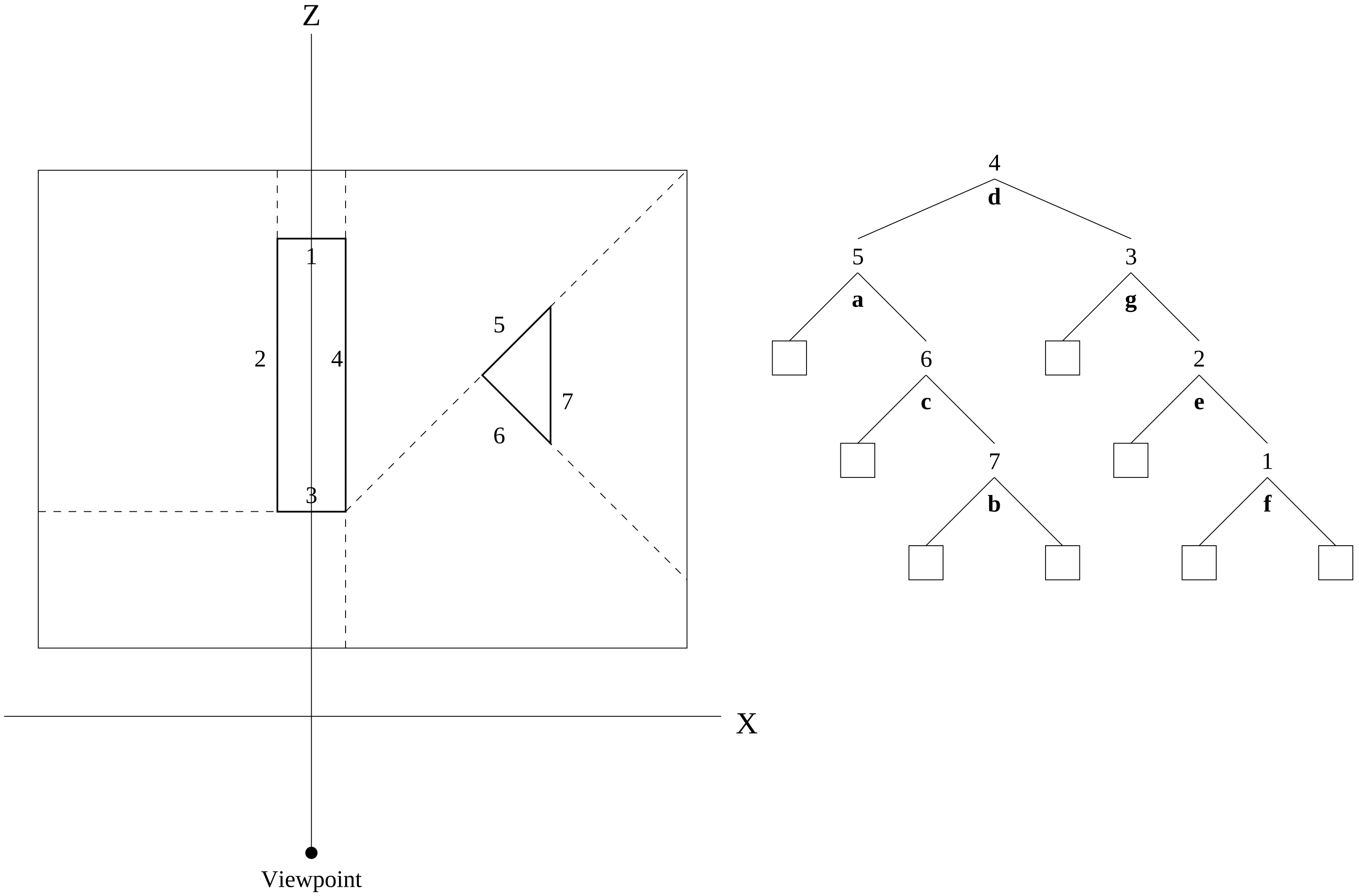


Left children are "in front of" the chosen line segment.

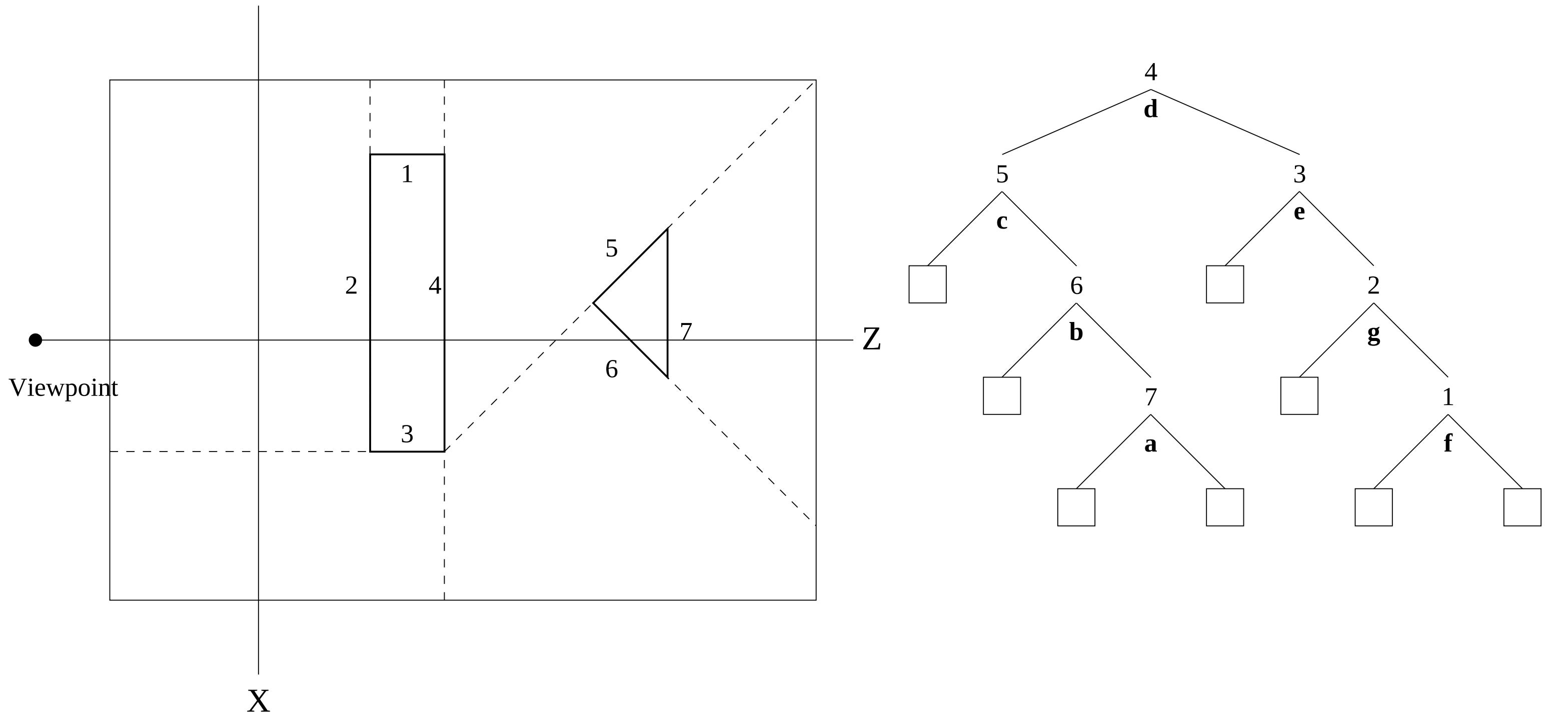
Right children are "behind" the chosen line segment.

Each node stores any line segments "colinear" with the chosen segment.

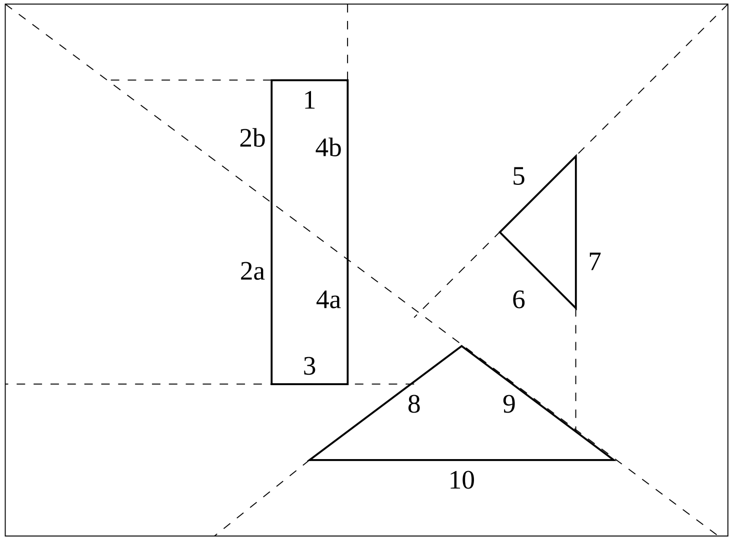
Hidden Surface Elimination using BSP Trees

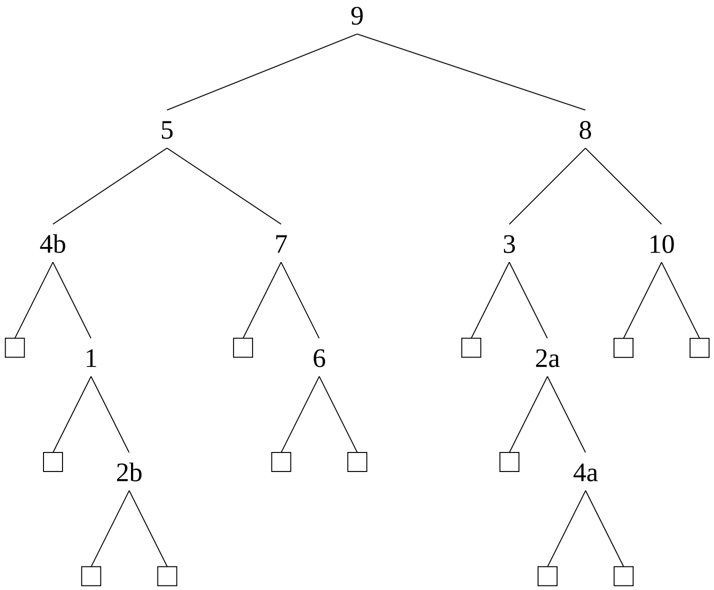


Moving the Viewpoint—The Tree Does Not Change



Some Segments May Need to be Split to Grow the Tree





BSP Classes

```
class bspnode_t                              // node of bsp tree
{
public:
  bspnode_t(plane_t & p, bspnode_t * f, bspnode_t * b, bsptriangle_t* c):
    plane(p), front(f), back(b), contents(c) {};
  int infront(point3_t & p);     // is p in front of the plane?
protected:
  plane_t plane ;                   // plane eq of separating plane
  bspnode_t * front;                         // front child node
  bspnode_t * back;                  // back child node
  bsptriangle_t * contents;        // triangles that lie in this plane
} ;
```

BSP Classes Continued

```
enum bspresult_t {INFRONT, INBACK, SPLIT, COPLANAR};
class bsptriangle_t : public triangle_t     // triangle for BSP trees
{
public:
  void display(bspscene_t *scene, point3_t & camera, int & col); // display this triangle
  void split(bsptriangle_t * splitter,
        bsptriangle_t * & front,
        bsptriangle_t * & back);
   bspresult_t compare(bsptriangle_t * other);
protected:
  bsptriangle_t * next;
  void vertexsplit(bsptriangle_t *, vertex3_t &, vertex3_t &, vertex3_t &,
          double, bsptriangle_t * &, bsptriangle_t * &);
  void sidesplit(bsptriangle_t *,
        vertex3_t &, vertex3_t &, vertex3_t &,
        bsptriangle_t * &, bsptriangle_t * &);
} ;
```

Building BSP Trees

BSP trees are constructed much like Quicksort. We use the first triangle to split the space into triangles on one side and triangles on the other side. Triangles must be split if they span the plane.

```
bspnode_t * bspscene_t::buildTreeTopDown(bsptriangle_t * faces)
{
 if (faces) {
   // all faces lie in a single plane, we are done
   plane_t p = faces→givp();
   int allSame = 1;
   for (bsptriangle_t * face = faces→next; face; face = face→next) {
     if (! (p == face→givp())) { allSame = 0; break; }
   }
   if (allSame) return new bspnode_t(p, 0, 0, faces);
```

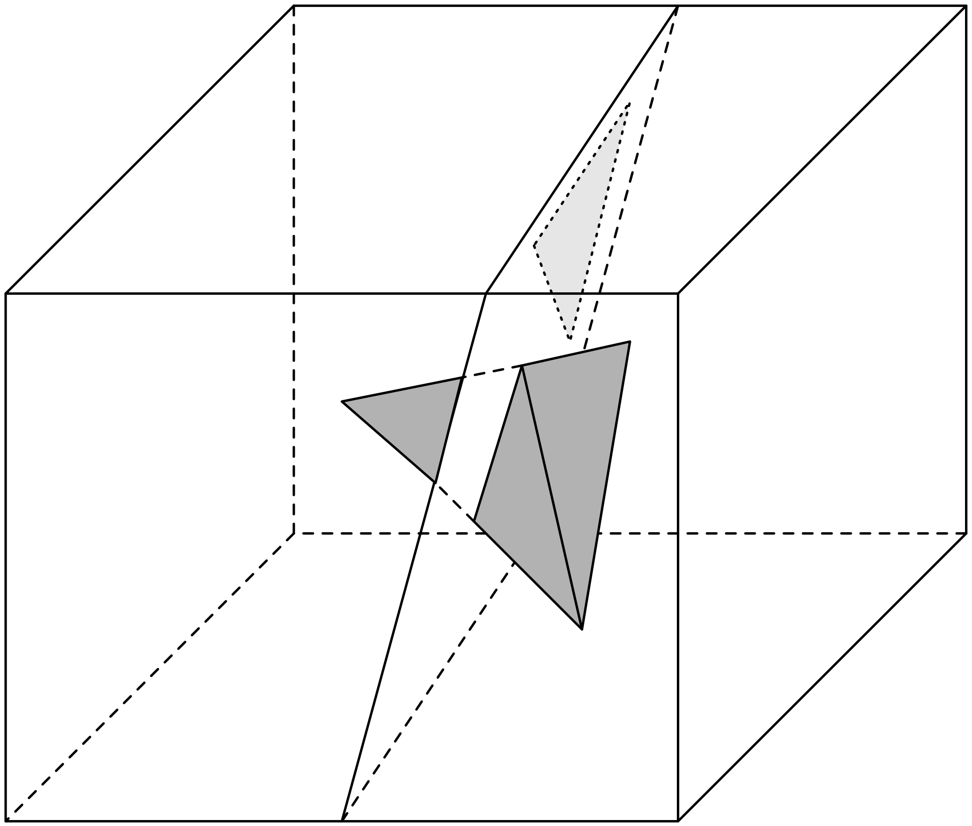
Building BSP Trees (2)

```
    else {
      bsptriangle_t * best = bestSplit(faces);
      bsptriangle_t * front = 0, back = 0, coplanar = 0, next = 0;
      for (bsptriangle_t * face = faces; face; face = next) {
        next = face→next;
        switch (best→compare(face)) {
          case INFRONT: face→next = front; front = face; break;
          case INBACK:  face→next = back; back = face; break;
          case COPLANAR: face→next = coplanar; coplanar = face; break;
          case SPLIT: face→split(best, front, back); break;
          }
      }
      return new bspnode_t(best→givp(), buildTreeTopDown(front),
                          buildTreeTopDown(back), coplanar);
    }}
  else return 0;
}
```

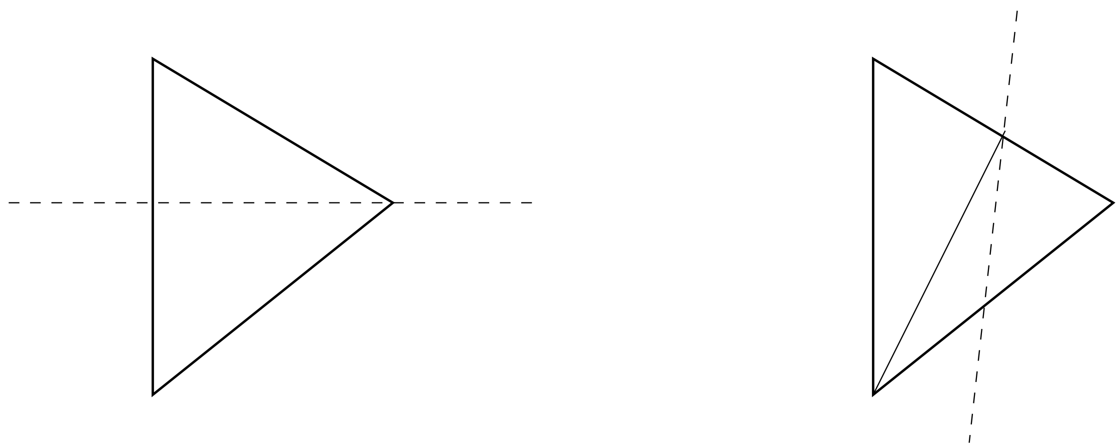
Deciding Whether a Triangle is Split by a Plane

```
bspresult_t bsptriangle_t::compare(bsptriangle_t * other)
{
  if (p == other→p) return COPLANAR;    // overloaded ==
  else {
    double side1 = p.planeside(other→giva());
    double side2 = p.planeside(other→givb());
    double side3 = p.planeside(other→givc());
    if (side1 ≥ 0 && side2 ≥ 0 && side3 ≥ 0) return INFRONT;
    else if (side1 ≤ 0 && side2 ≤ 0 && side3 ≤ 0) return INBACK;
    else return SPLIT;
  }
}
```

Splitting Spanning Triangles



Splitting Cases



If the splitting plane intersects one of the vertexes, we get two triangles. Otherwise, we get a triangle and a quadrilateral. We must split the quadrilateral into two triangles. Care must be taken to keep the vertexes in clock-wise order.

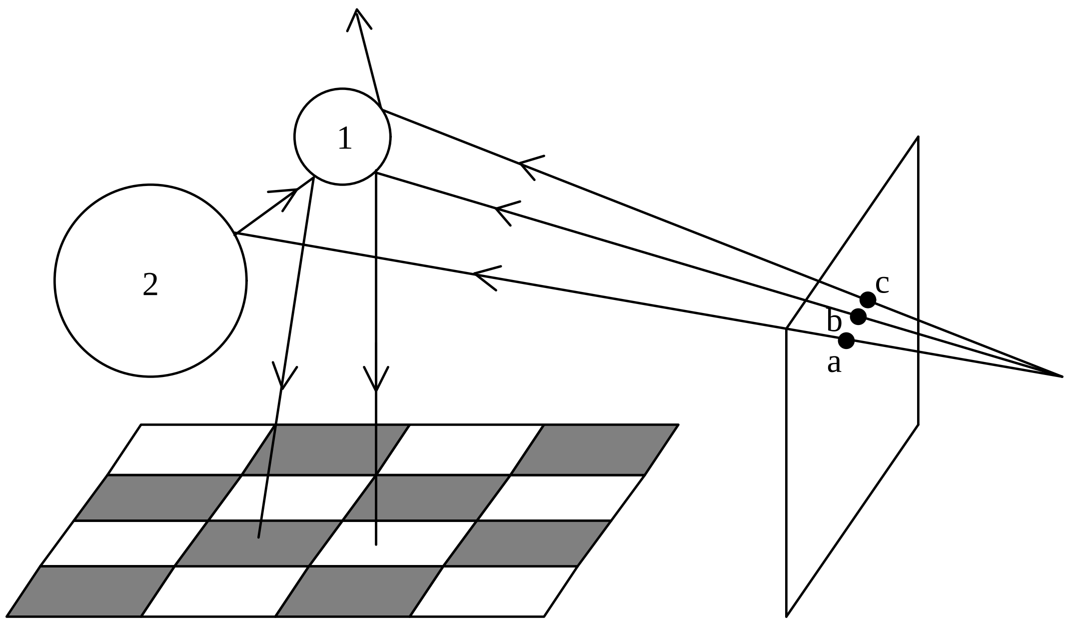
Implementing the Painter's Algorithm

```
void bspscene_t::walk(bspnode_t * node, point3_t & camera, int & col)
{
  if (node) {
    if (node→infront(camera)) {
      walk(node→back, camera, col);
      node→contents→display(this, camera, col);
      walk(node→front, camera, col);
    }
    else {
      walk(node→front, camera, col);
      node→contents→display(this, camera, col);
      walk(node→back, camera, col);
    }
  }
}
```

Ray Tracing

The more accurately we simulate the physics of image formation, the more realistic our images will be. Real images are formed when light rays exit a light source, bounce off various objects in the scene, and are reflected to the eye.

Ray Tracing simulates this by **working backward** from the eye. We follow a ray from the eye until it strikes an object. If the object is shiny, we compute a reflection angle and follow the ray until it hits something else. Eventually, the ray will hit a light source or an object with only diffuse reflection.



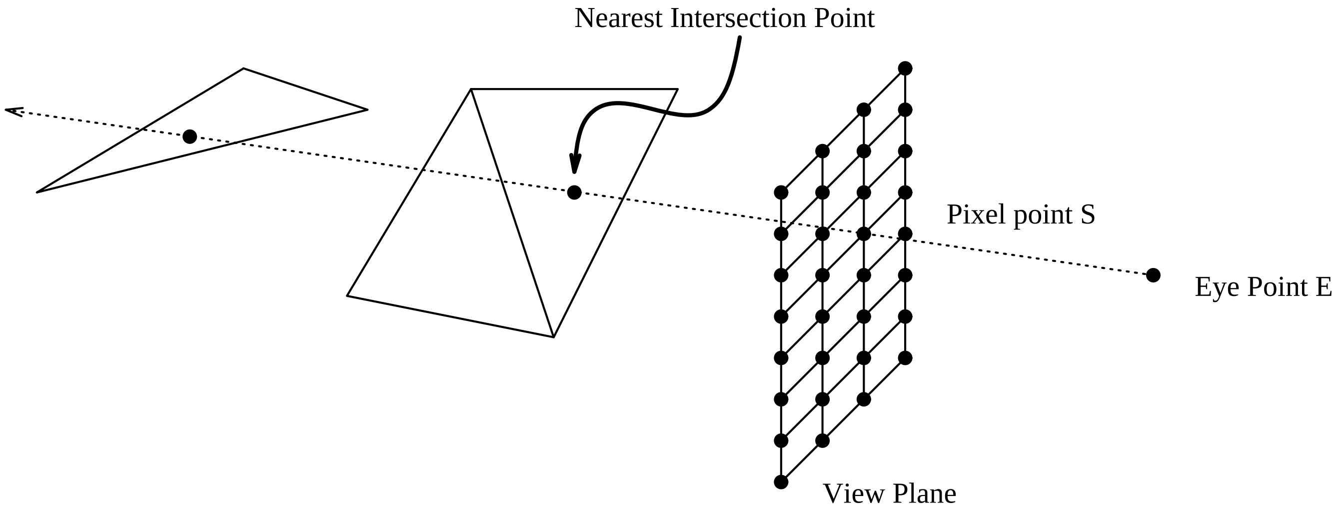
Advantages and Disadvantages of Ray Tracing

- **Can correctly model shadows and multiple reflected images.**
- **Can handle refraction and transparent objects.**
- **Can handle boolean combinations of objects.**

  Constructive Solid Geometry: CSG.
- **Can handle non-polygonal objects.**

  Still requires computing normal vectors.
- **Very Expensive.**

Ray Tracing Polygons with Diffuse Reflection



Starting at the eye point, we send a ray through one of the pixels on the screen and follow it until it hits a polygon. Let $f(P) = 0$ be the equation defining the polygon.

- **Map the device coordinates of the pixel into world coordinates:** S.
- **Compute the ray:** Starting point E, normalized direction $D = (S - E)/\|S - E\|$.
- **Compute intersection points:**. Find smallest u such that $f(E + uD) = 0$. Entry point $P_{nearest} = E + uD$.
- **Compute surface normal and RGB values for diffuse reflection at** $P_{nearest}$**.**

Comparison to Z Buffering

Different order of loops:

```
// Z buffering
fill Z with 1.0;
for each front face polygon T {
   for each pixel P in T {
     if (P.z < Z[P.x][P.y]) {
        Z[P.x][P.y] = P.z;
        col(illum(P,T));
        setpix(P.x,P.y);
        }
     }
   }
```

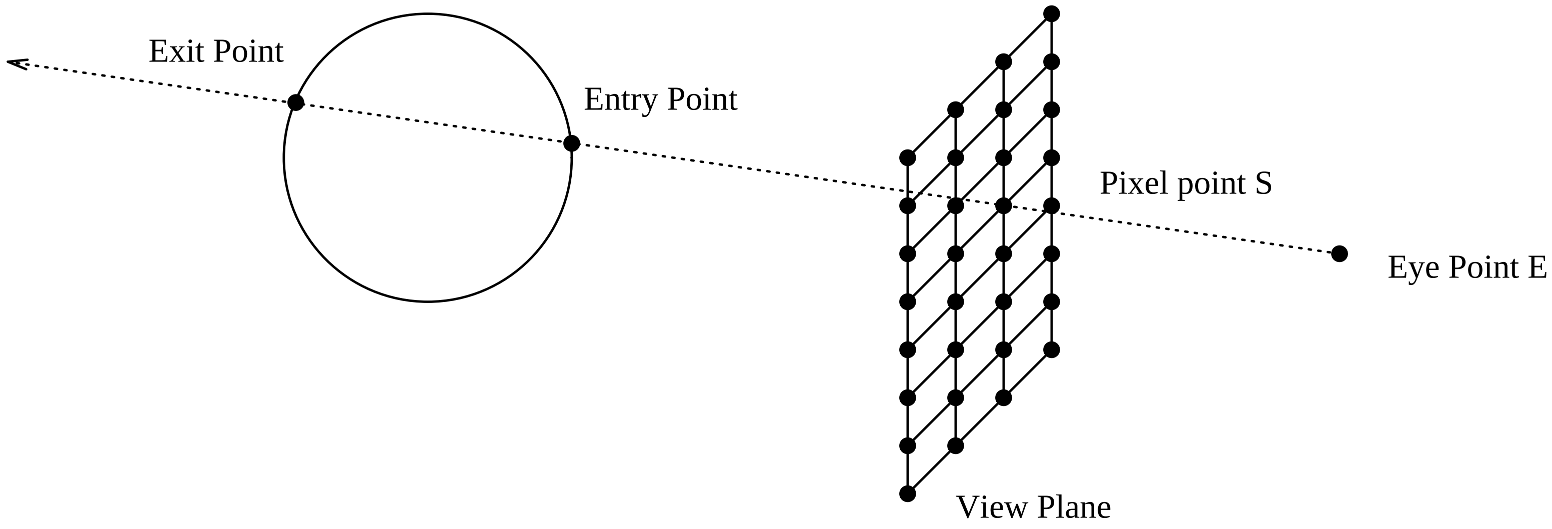
```
// Ray Tracing
for each pixel P in image {
   R = ray from eyepoint E through P
   H = NULL;            // hit triangle
   u = infinity;
   for each front face polygon T {
     thisu = u value where R hits T;
     if (thisu < u) {
        H = T;
        u = thisu;
        }
     }
   col(illum(R(u),T));
   setpix(H.x,H.y);
   }
```

Pruning Useless Front Faces

How can we avoid testing for intersections against useless faces? Use BSP trees.

```
triangle_t * BSPtree::trace(ray_t ray)
{
  triangle_t * result = 0;
  ray_t continuation = ray_t(intersection(ray,splitter),ray.direction);
  if (side(ray.start()) == Behind ) {
     if (result = behind()→trace(ray)) { return result;}
     if (result = splitter.trace(ray)) { return result; }
     return infront()→trace(continuation);
     }
  else {
     if (result = infront()→trace(ray)) { return result;}
     if (result = splitter.trace(ray)) { return result; }
     return behind()→trace(continuation);
     }
}
```

A Simple Example: Sphere with Diffuse Reflection



Starting at the eye point, we send a ray through one of the pixels on the screen and follow it until it hits the sphere. Let $f(P) = 0$ be the equation defining the sphere.

- **Map the device coordinates of the pixel into world coordinates:** S.
- **Compute the ray:** Starting point E, normalized direction $D = (S - E)/\|S - E\|$.
- **Compute intersection points:**. Find smallest u such that $f(E + uD) = 0$. Entry point $P_{ent} = E + uD$.
- **Compute surface normal and RGB values for diffuse reflection at** P_{ent}**.**

Algebra

Equation of Sphere: $(x - c_x)^2 + (y - c_z)^2 + (z - c_z)^2 - r^2 = 0$.

Center of Sphere: $C = (c_x, c_y, c_z)$.

World coordinates of pixel: S.

World coordinates of eye: E.

Ray direction: $D = \frac{S-E}{\|S-E\|}$

Translate sphere to the origin (i.e., by $-C$). Its equation becomes $x^2 + y^2 + z^2 - r^2 = 0$.

Translate starting point of ray to origin: $E - C$.

Equation of ray is therefore: $P = E - C + uD$.

Plug this into the sphere equation:

$$(E_x - C_x + uD_x)^2 + (E_y - C_y + uD_y)^2 + (E_z - C_z + uD_z)^2 - r^2 = 0$$

Solve this using the quadratic formula.

Algebra (2)

Express this as

$$a_2 u^2 + 2a_1 u + a_0 = 0$$

where

$$a_2 = D_x^2 + D_y^2 + D_z^2 = 1 \qquad a_1 = D \cdot (E - C) \qquad a_0 = (E - C) \cdot (E - C) - r^2$$

$$u = \frac{-2a_1 \pm \sqrt{(2a_1)^2 - 4a_2 a_0}}{2a_2}$$

This can be simplified by dividing through by 2 and removing a_2:

$$u = -a_1 \pm \sqrt{a_1^2 - a_0}$$

The discriminant (inside the square root) determines the number of solutions:

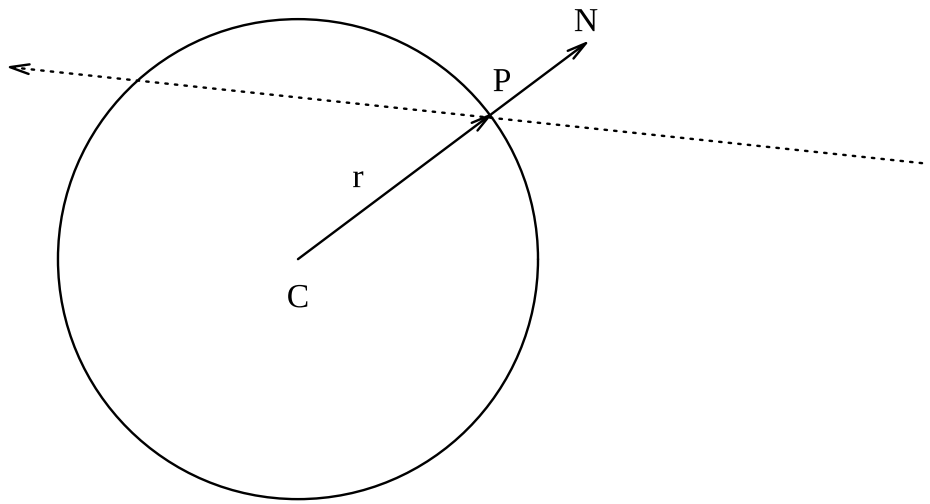
$a_1^2 - a_0 < 0$	No solutions (imaginary)
$a_1^2 - a_0 = 0$	One solution ("grazing")
$a_1^2 - a_0 > 0$	Two solutions

Surface Normal at Hit Point

On a sphere, normal lies along the radius.

Let $P = E + uD$ be the hit point. Then the surface normal is just $P - C$. This has length r, so we can normalize it by dividing by r:

$$N = \frac{P - C}{r}$$



Worked Out Example

Equation of Sphere: $x^2 + y^2 + z^2 - r^2 = 0$.

Center of Sphere: (0.1, 0.3, 0.8); Radius: 0.4

Eye point: $E = (0, 0, -3)$

Pixel size of screen: width = 640, height = 400.

World coordinates of screen:

$$left = -1 \qquad right = 1 \qquad top = 0.625 \qquad bottom = -0.625$$

Chosen Pixel: (307,123) In world coordinates this is $(-0.04062, 0.24062)$.

Normalized Ray Direction $D = (-0.01349, 0.07994, 0.99671)$.

Computing Sphere/Ray Intersection

Translate sphere to origin (i.e., by $(-0.1, -0.3, -0.8)$).

Translate ray starting point by same amount. It becomes $(-0.1, -0.3, -3.8)$.

Compute discriminant: $a_1^2 - a_0$:

$$a_1 = -3.81013 \qquad a_0 = 14.38 \qquad a_1^2 - a_0 = 0.13709$$

This is greater than zero, so compute u values:

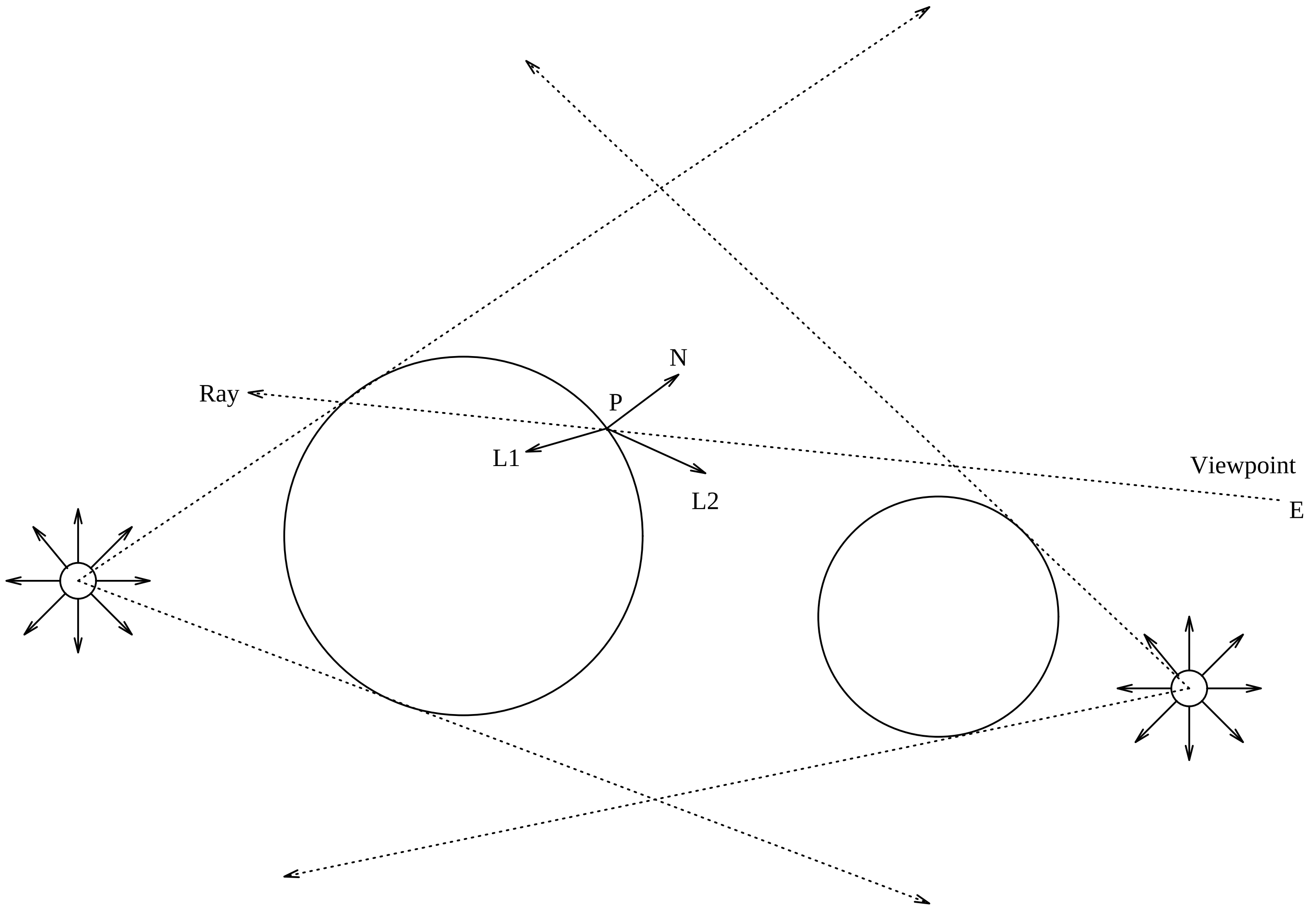
$$u_{enter} = 3.43987 \qquad u_{exit} = 4.18038$$

Plug in u to compute hit point: $(-0.04642, 0.27499, 0.42855)$.

Normal vector at hit point: $(-0.36605, -0.062525, -0.92862)$

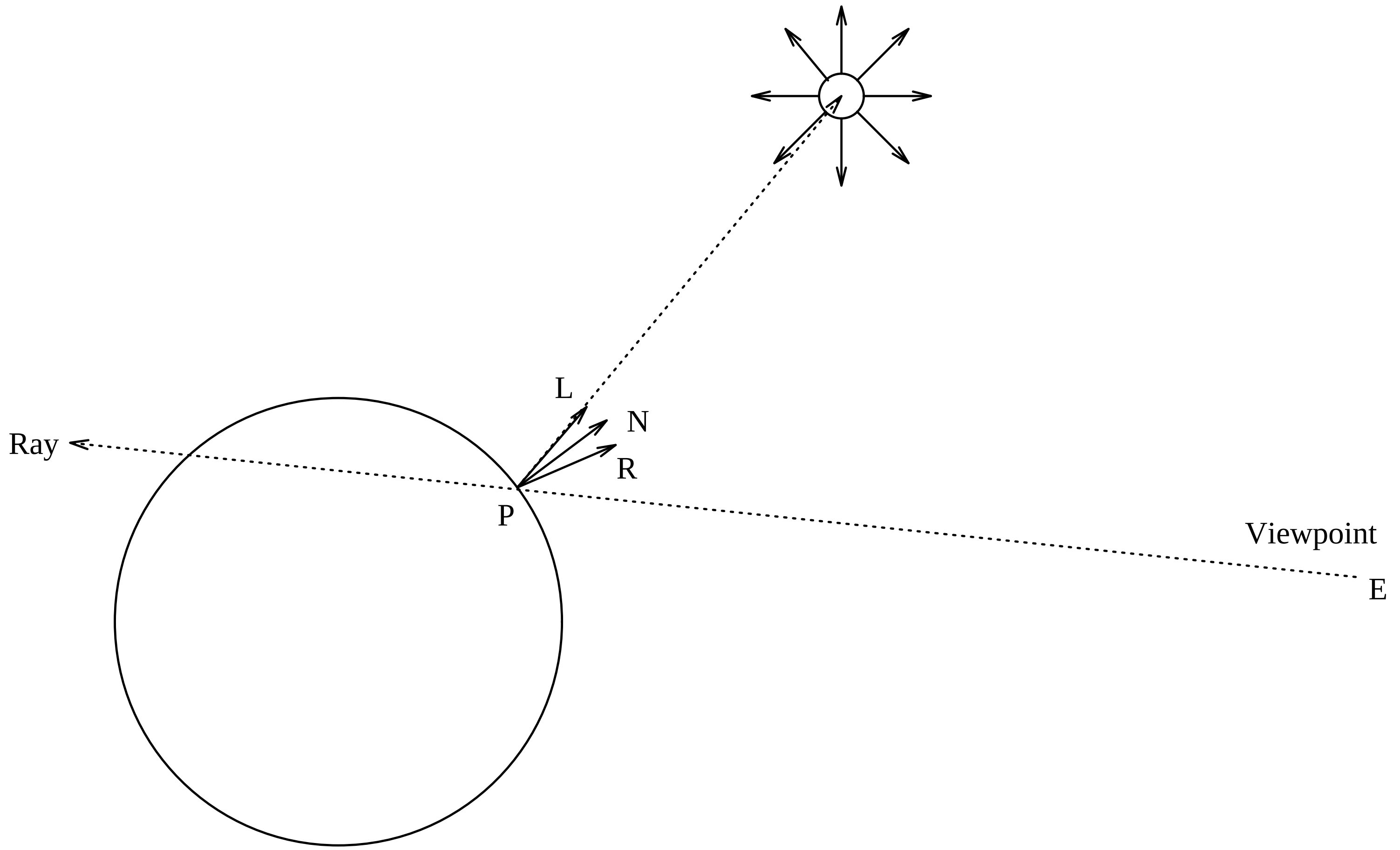
Shadows

To check for shadows, we can define secondary rays from point P to each of the point light sources L_1 and L_2. If a ray hits some object, then the corresponding light source does not strike P.



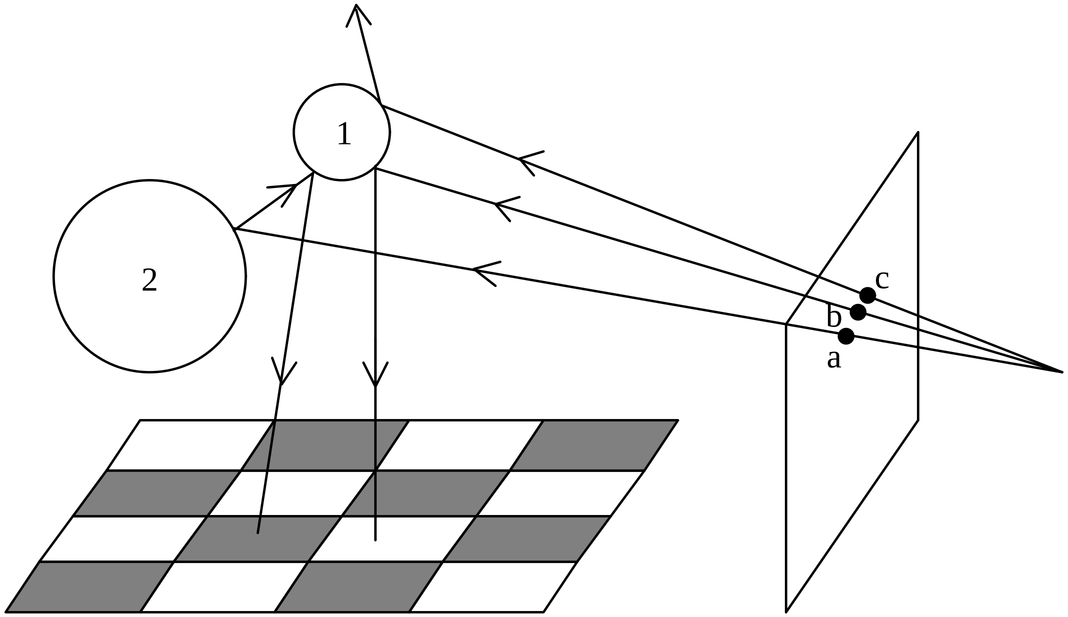
Soft Highlights from Partial Specular Reflection

If the ray reaches the light source, then apply Phong's rule to compute "soft" specular highlight.



Specular Reflections of Other Objects

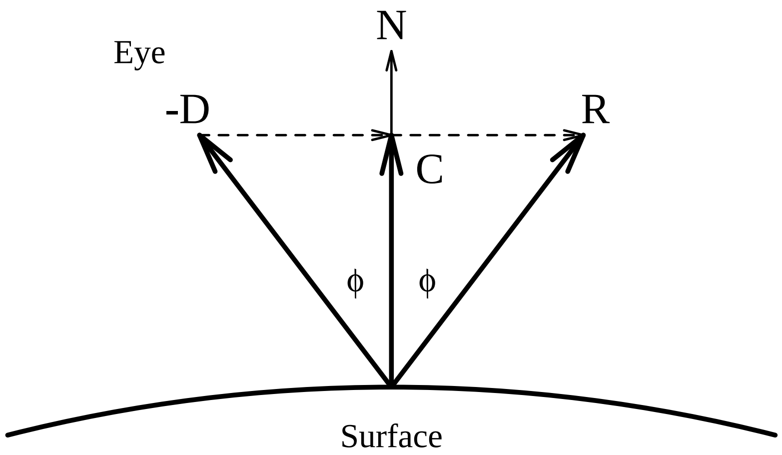
For shiny objects, we must "bounce" the ray off the object to see what it strikes next. Recursion terminates when ray hits non-specular surface or nothing.



Computing the Reflection Vector

Very similar to the reflection vector for light sources:

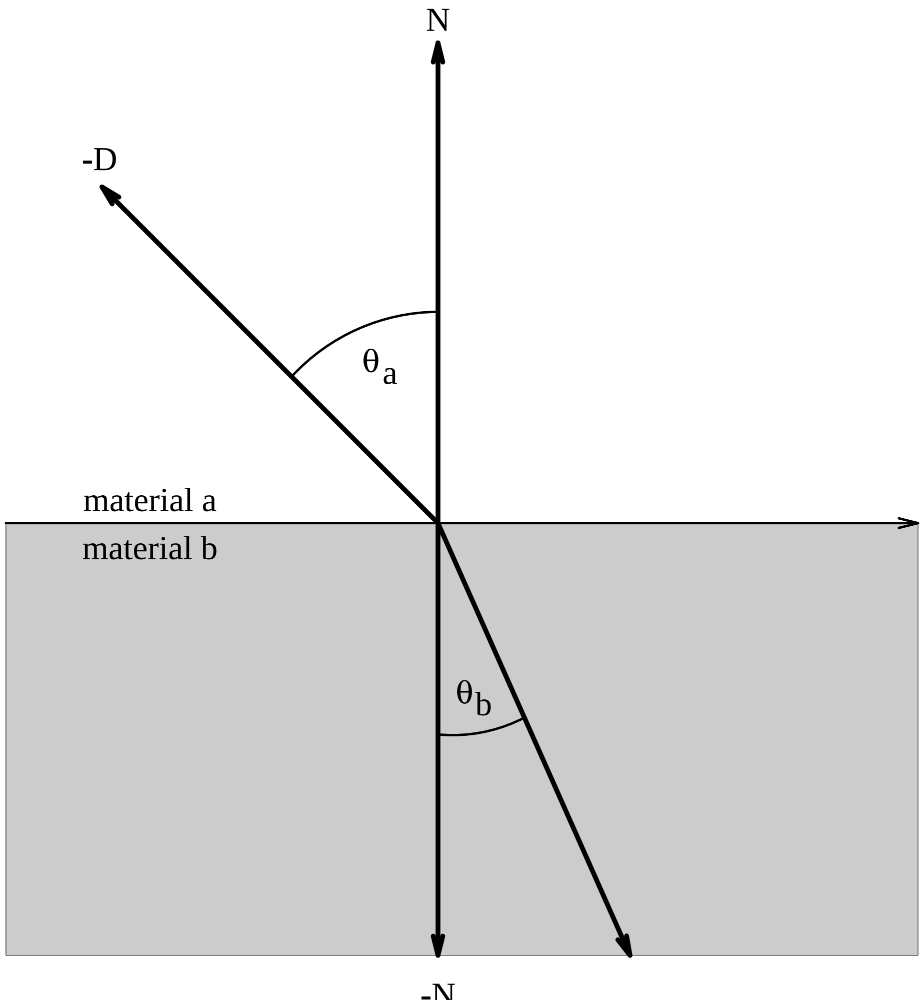
$$\begin{aligned} C &= (-D \cdot N)N \\ R &= -D + 2(C - (-D)) \\ &= -D + 2(C + D) \\ &= -D + 2((-D \cdot N)N + D) \\ &= D - 2(D \cdot N)N \end{aligned}$$



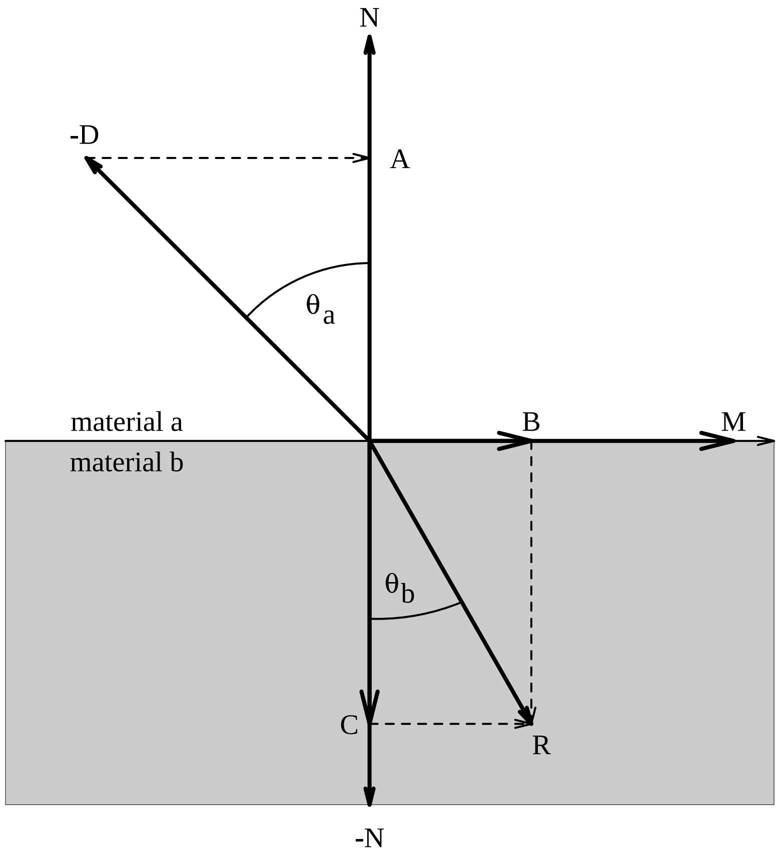
Transparency and Refraction

Ray tracing permits us to model light passing through an object. Each object has an **index of refraction** n. Snell's law tells how the direction of a light ray changes:

$$\frac{\sin\theta_a}{\sin\theta_b} = \frac{n_b}{n_a}$$



Computing Refraction Ray



$$\cos\theta_a = (-D \cdot N)$$

$$A = N(-D \cdot N) \qquad \cos^2\theta_a$$

$$M\sin\theta_a = A - (-D)$$

$$M\sin\theta_a = A + D$$

Snell's Law

$$n_a \sin\theta_a = n_b \sin\theta_b$$

$$\sin^2\theta_a = 1 - \cos^2\theta_a$$

$$M\sin\theta_b = (A + D)(n_a/n_b)$$

$$\sin^2\theta_b = (n_b/n_a)^2 \sin^2\theta_a$$

$$B = (A + D)(n_a/n_b)$$

$$\cos^2\theta_b = 1 - \sin^2\theta_b$$

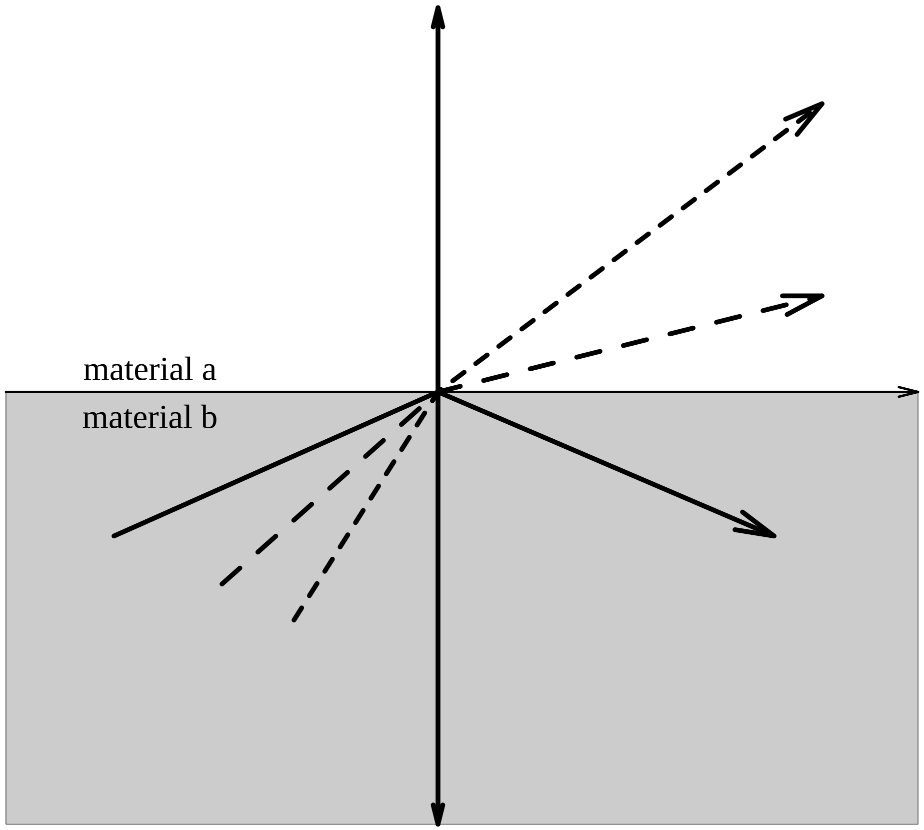
$$C = \cos\theta_b(-N)$$

$$R = B + C$$

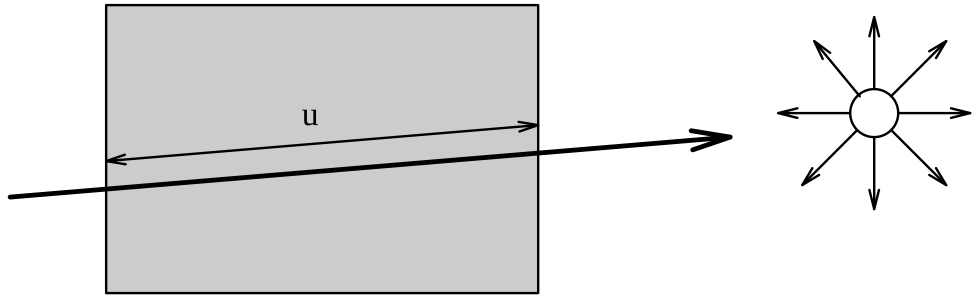
Internal Reflection

$$R = N\left(\frac{n_a}{n_b}(-D\cdot N) - \sqrt{1-\left(\frac{n_a}{n_b}\right)^2(1-(-D\cdot N)^2)}\right) + \frac{n_a}{n_b}D$$

When a light ray hits a boundary between a dense region to a less dense region, the square root may not exist. When this happens, the light ray is **reflected internally**. We compute reflection instead of refraction.



Transparency



- **Interpolative Transparency**. Suppose light source has colors $L = (L_R, L_G, L_B)$ and object has colors $O = (O_R, O_G, O_B)$ with transparencies $T = (T_R, T_G, T_B)$. Then, the light emerging (backwards) along the ray is

$$((1 - T_R)O_R + T_R L_R, (1 - T_G)O_G + T_G L_G, (1 - T_B)O_B + T_B L_B)$$

- **Screen Door Transparency**. Create a mesh in which some pixels are rendered according to the more distant object (or light source) and other pixels are rendered using the nearer object.

- **Absorption**. Object's colors specify which wavelengths it absorbs. Depends on distance u travelled inside the object.

$$(uT_R L_R, uT_G L_G, uT_B L_B)$$