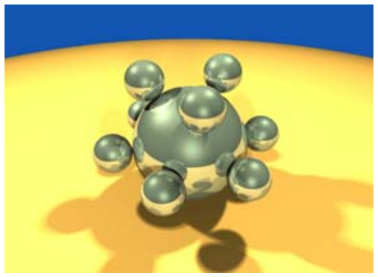
Computer Graphics using OpenGL, 3rd Edition

F. S. Hill, Jr. and S. Kelley

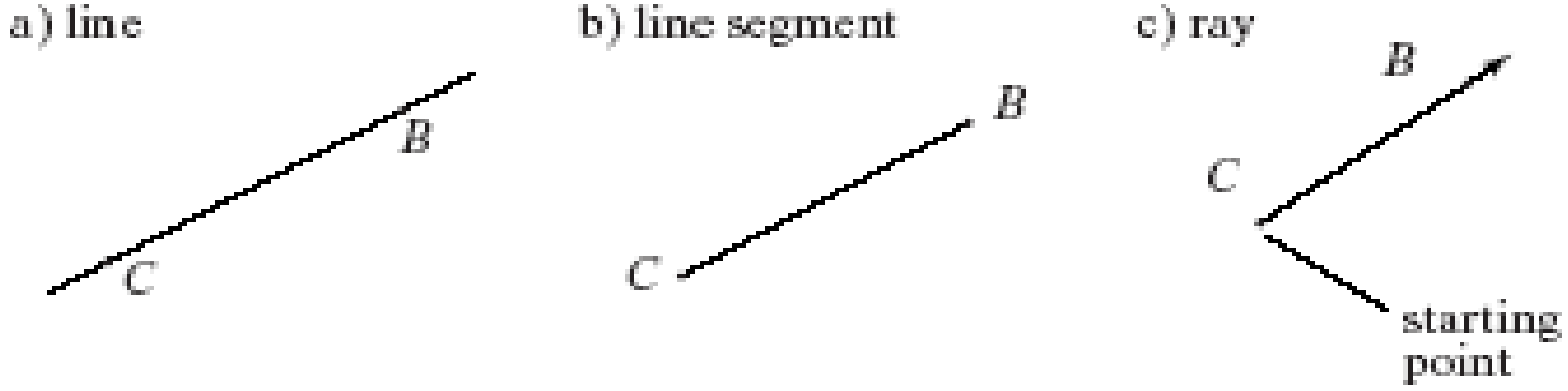


Chapter 4.4-8
Vector Tools for Graphics

S. M. Lea
University of North Carolina at Greensboro

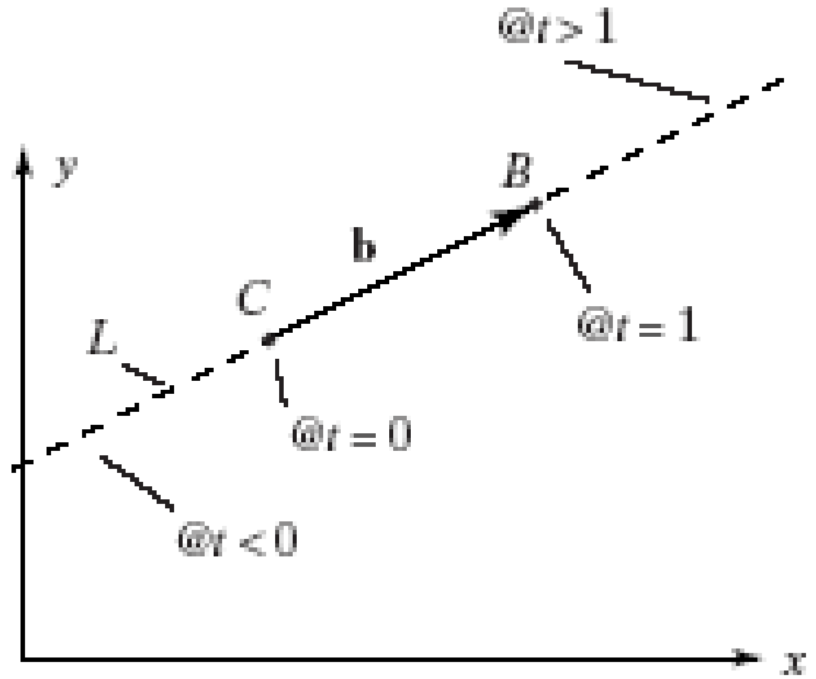
Representing Lines

- A line passes through 2 points and is infinitely long.
- A line segment has 2 endpoints.
- A ray has a single endpoint.



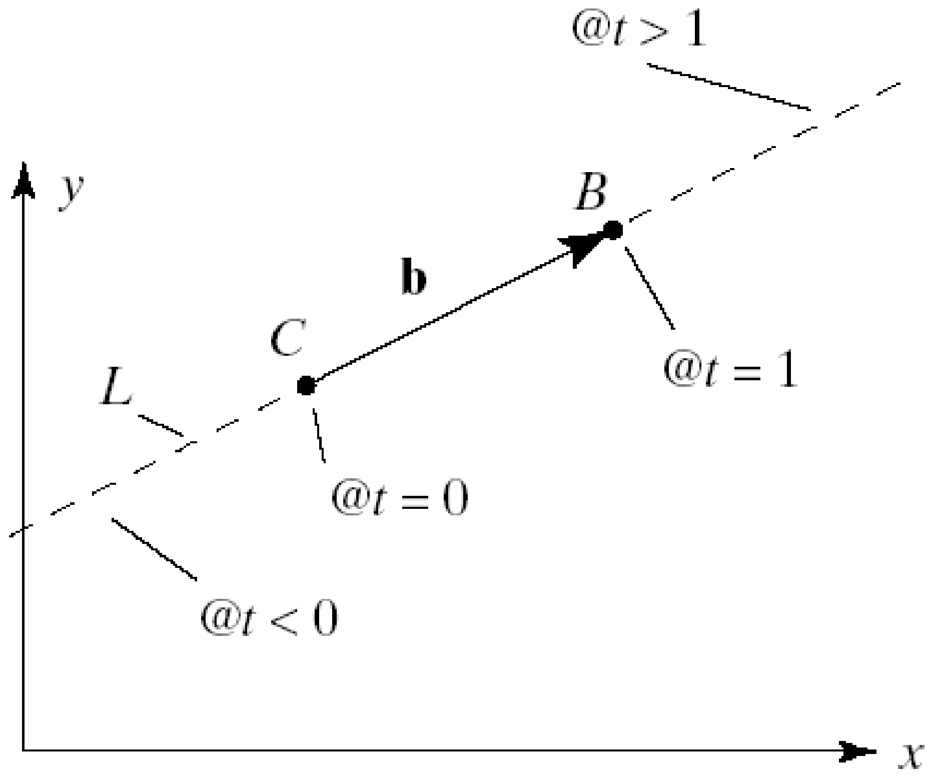
Representing Lines (2)

- There are 2 useful line representations:
- Parametric form: we have 2 points, B and C, on the line. P(x, y) is on the line when P = C + **b**t, where **b** = B – C.
  - 0 ≤ t ≤ 1: line segment; -∞≤ t ≤ ∞: line; -∞≤ t ≤ 0 or 0 ≤ t ≤ ∞: ray.



Representing Lines (3)

- As *t* varies so does the position of *L*(*t*) along the line. (Let *t* be time.)
- If *t* = 0, *L*(0) = *C* so at *t* = 0 we are at point *C.*
- At *t* = 1, *L*(1) = *C* + (*B* - *C*) = *B*.
- If *t* > 1 this point lies somewhere on the opposite side of *B* from *C*, and when *t* < 0 it lies on the opposite side of *C* from *B*.



Representing Lines (4)

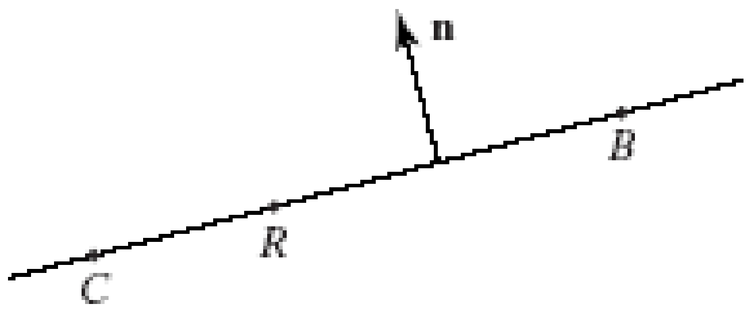
- $L(t)$ lies fraction t of the way between C and B when t lies between 0 and 1.
- When $t = 1/2$ the point $L(0.5)$ is the **midpoint** between C and B, and when $t = 0.3$ the point $L(0.3)$ is 30% of the way from C to B: $|L(t) - C| = |\mathbf{b}|\,|t|$ and $|B - C| = |\mathbf{b}|$, so the value of $|t|$ is the ratio of the distances $|L(t) - C|$ to $|B - C|$.

Representing Lines (5)

- The speed with which the point $L(t)$ moves along line L is given by distance $|\mathbf{b}|\,t$ divided by time t, so it is moving at constant speed $|\mathbf{b}|$.
- There is a significant difference between a parametric form for a curve ($p(t)$) and a motion path for the same curve.
  - Given $p(t)$ swept out as t increases gives no information as to how fast the point moves along that path. (The picture for $p(t)$ is the same as that for $p(t^2)$ or $p(t^3)$.)

Representing Lines (6)

- Point-normal (implicit) form: (this works in 2D only; the 3D version requires 2 equations.)
  - fx + gy = 1 gives (f, g)·(x, y) = 1.



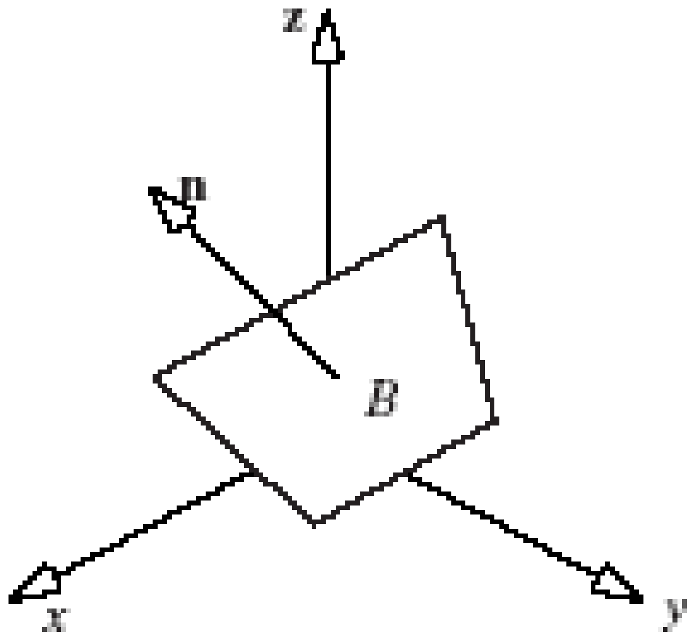
  - Given B and C on the line, **b** = B – C gives $\mathbf{b}^{\perp}$ = **n**, which is perpendicular to R – C. (R is any point (x, y) on the line.)
  - The equation is **n** · (R – C) = 0.

Changing Representations

- From $f\,x + g\,y = 1$ to point-normal form: Writing $(f, g) \cdot (x, y) = 1$, $\mathbf{n}$ is (f, g) (or any multiple thereof).
- From point normal form $\mathbf{n} \cdot (P - C) = 0$ to parametric form: $L(t) = C + \mathbf{n}^{\perp}t$

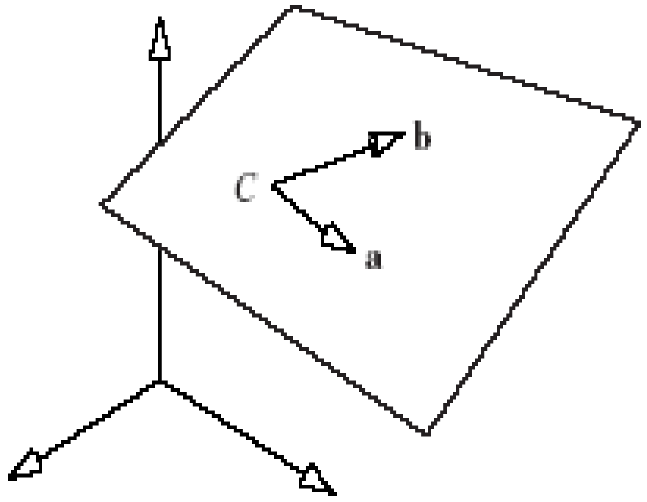
Representing Planes: Point-Normal Form

- Point-normal form: $\mathbf{n} \cdot (P - B) = 0$; where B is a given point on the plane, and $P = (x, y, z, 1)^T$.



Planes: Parametric Form

- A plane can be infinite in 2 directions, semi-infinite, or finite.
  - Parametric form: requires 3 non-collinear points on the plane, A, B, and C.
  - $P(s, t) = C + s\mathbf{a} + t\mathbf{b}$, where $\mathbf{a} = A - C$ and $\mathbf{b} = B - C$.
  - $-\infty \le s \le \infty$ and $-\infty \le t \le \infty$: infinite plane.
  - $0 \le s \le 1$ and $0 \le t \le 1$: a finite plane, or patch.



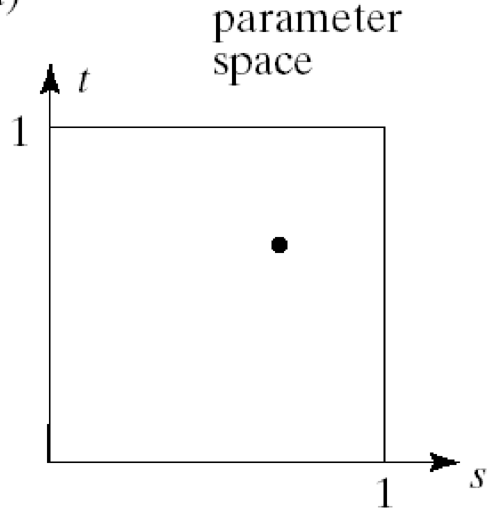
Planes: Parametric Form (2)

- We can rewrite *P(s,t)* = C + s**a** + t**b**, where **a** = A – C and **b** = B – C, as an affine combination of points: $P(s, t) = s A + t B + (1 - s - t)C$.

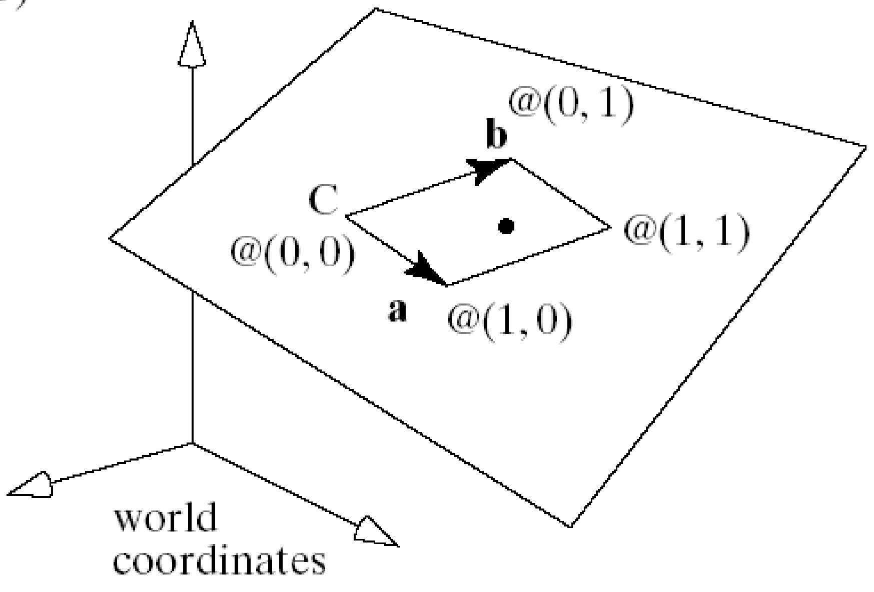
Planes: Parametric Form (3)

- The figure shows the available range of *s* and *t* as a square in **parameter space**, and the patch that results from this restriction in object space.

a)



b)



Patches

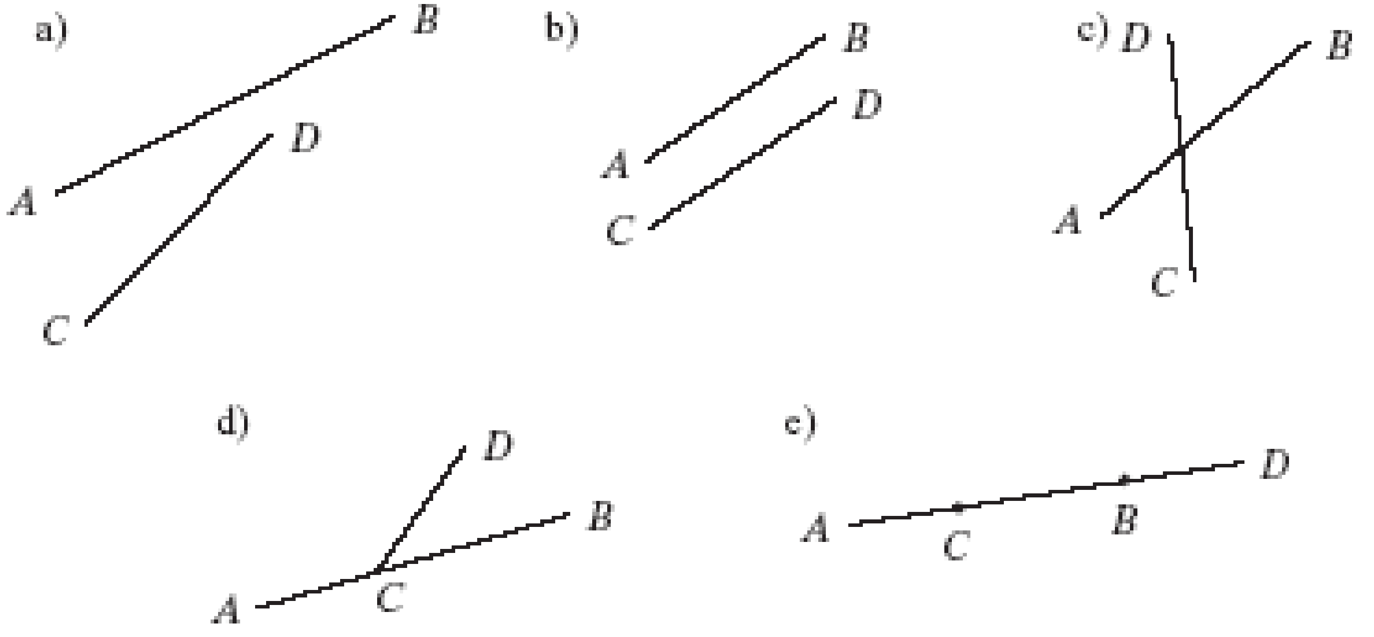
- Mapping textures onto faces involves finding a mapping from a portion of parameter space onto object space, as we shall see later.
- Each point (s, t) in parameter space corresponds to one 3D point in the patch $P(s, t) = C + \mathbf{a}s + \mathbf{b}t$.
- The patch is a parallelogram whose corners correspond to the four corners of parameter space and are situated at $P(0, 0) = C$; $P(1, 0) = C + \mathbf{a}$; $P(0, 1) = C + \mathbf{b}$; $P(1, 1) = C + \mathbf{a} + \mathbf{b}$.

Patches (2)

- The vectors **a** and **b** determine both the size and the orientation of the patch.
- If **a** and **b** are perpendicular, the grid will become rectangular.
- If in addition **a** and **b** have the same length, the grid will become square.
- Changing *C* just translates the patch without changing its shape or orientation.

Finding the Intersection of 2 Line Segments

- They can miss each other (a and b), overlap in one point (c and d), or even overlap over some region (e). They may or may not be parallel.



Intersection of 2 Line Segments (2)

- Every line segment has a **parent line**, the infinite line of which it is part. Unless two parent lines are parallel, they will intersect at some point in 2D. We locate this point.
- Using parametric representations for each of the line segments in question, call *AB* the segment from *A* to *B.* Then $AB(t) = A + \mathbf{b}\, t$, where for convenience we define $\mathbf{b} = B - A$.
- As *t* varies from 0 to 1 each point on the finite line segment is crossed exactly once.

Intersection of 2 Line Segments (3)

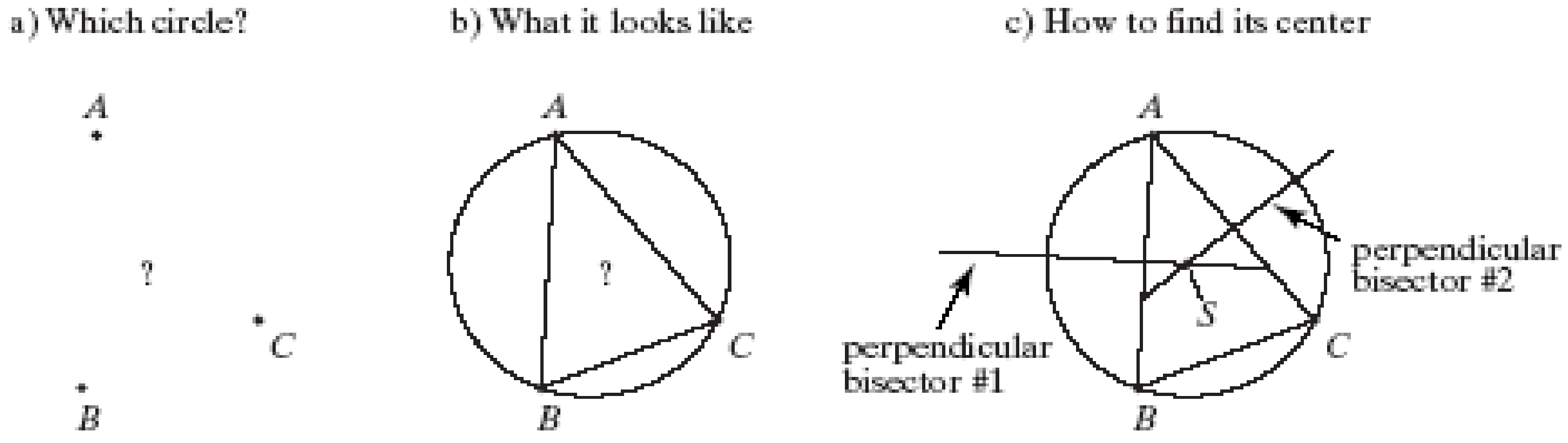
- AB(t) = A + **b**t, CD(u) = C + **d**u, where **b** = B – A and **d** = C – D.
- At the intersection, A + **b**t = C + **d**u, or **b**t = **c** + **d**u, with **c** = C – A.
- Taking dot product with $\mathbf{d}^{\perp}$ gives $\mathbf{b}\cdot\mathbf{d}^{\perp}\, t = \mathbf{c}\cdot\mathbf{d}^{\perp}$.
- Taking dot product with $\mathbf{b}^{\perp}$ gives $-\,\mathbf{c}\cdot\mathbf{b}^{\perp} = \mathbf{d}\cdot\mathbf{b}^{\perp}\, u$.

Intersection of 2 Line Segments (4)

- Case1: $\mathbf{b}\cdot\mathbf{d}^{\perp} = 0$ means $\mathbf{d}\cdot\mathbf{b}^{\perp} = 0$ and the lines are either the same line or parallel lines. There is no intersection.
- Case 2: $\mathbf{b}\cdot\mathbf{d}^{\perp} \neq 0$ gives $t = \mathbf{c}\cdot\mathbf{d}^{\perp} / \mathbf{b}\cdot\mathbf{d}^{\perp}$ and $u = -\,\mathbf{c}\cdot\mathbf{b}^{\perp} / \mathbf{d}\cdot\mathbf{b}^{\perp}$.
- In this case, the line segments intersect if and only if $0 \le t \le 1$ and $0 \le u \le 1$, at $P = A + \mathbf{b}(\mathbf{c}\cdot\mathbf{d}^{\perp} / \mathbf{b}\cdot\mathbf{d}^{\perp})$.

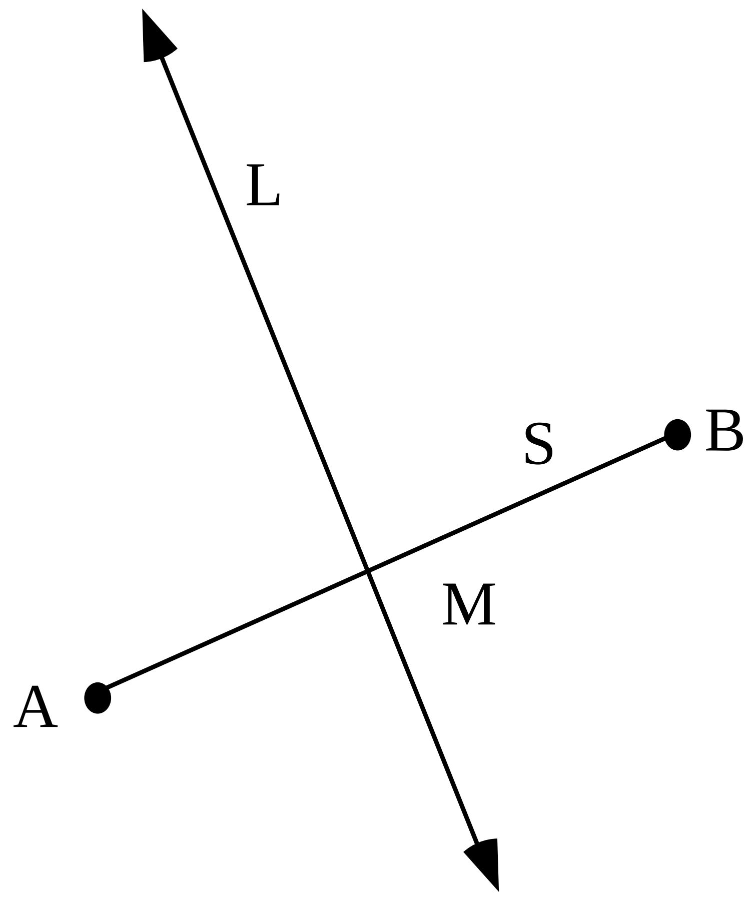
Finding A Circle through 3 Points

- We want to find the center and the radius of the circle.
  - The 3 points make a triangle, and the center S is where the perpendicular bisectors of two of the sides of the triangle meet.
  - The radius is r = |A – S|.


a) Which circle?
b) What it looks like
c) How to find its center
A
B
C
?
S
perpendicular bisector #1
perpendicular bisector #2

Circle through 3 Points (2)

- The perpendicular bisector passes through the midpoint M = ½ (A + B) of the line AB, in the direction (B – A)$^{\perp}$.
- Let **a** = B – A, b = C – B, and **c** = A - C.

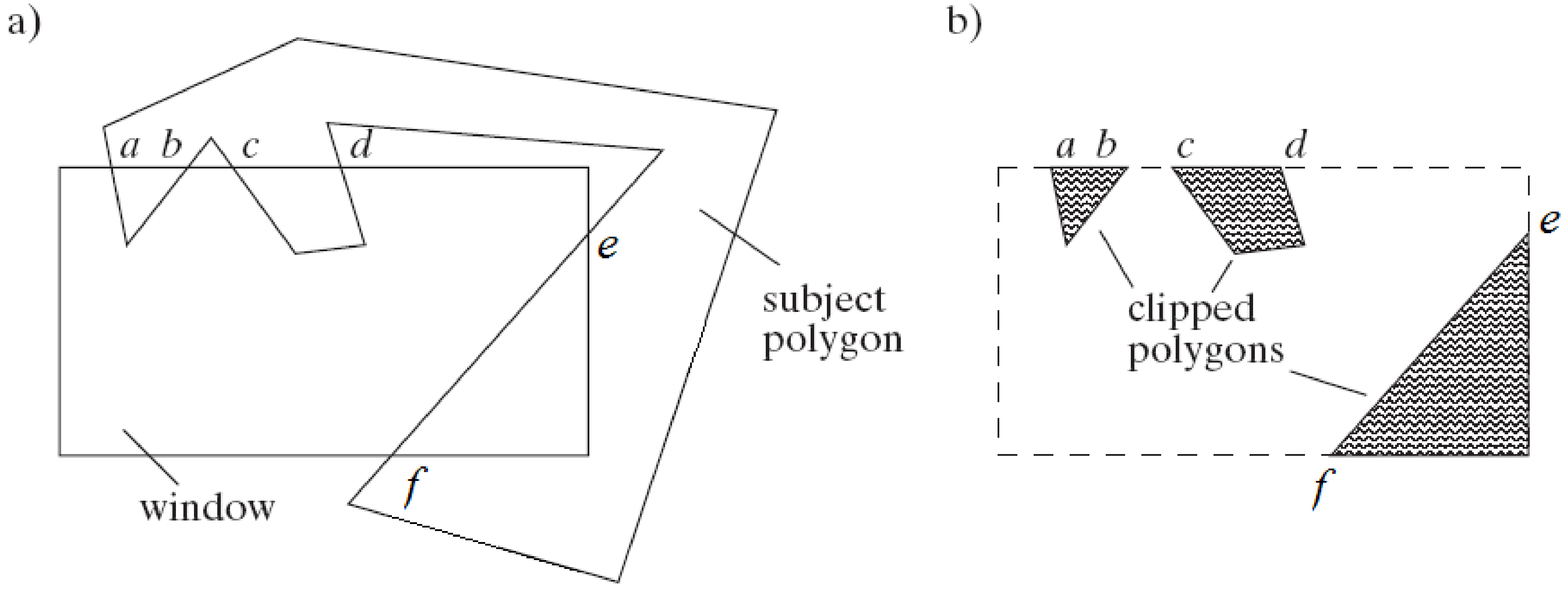


Circle through 3 Points (3)

- Midpoint of *AB: A* + **a** / 2; direction perpendicular to *AB*: $\mathbf{a}^{\perp}$.
- The perpendicular bisector of AB is *A* + **a** / 2 + $\mathbf{a}^{\perp}$ *t,* and for AC, *A* - **c** / 2 + $\mathbf{c}^{\perp}$ *u*, using parameter *u.*
- Point *S* lies where these meet, at the solution of $\mathbf{a}^{\perp} t = \mathbf{b} / 2 + \mathbf{c}^{\perp} u$ (where we have used **a** + **b** + **c** = **0**).
- Radius R = |S – A|, or $radius = \frac{|\mathbf{a}|}{2}\sqrt{\left(\frac{\mathbf{b}\cdot\mathbf{c}}{\mathbf{a}^{\perp}\cdot\mathbf{c}}\right)^2 + 1}$

Clipping a Polygon

- “To clip a polygon against a window” means to find which part of the polygon is inside the window and thus will be drawn.

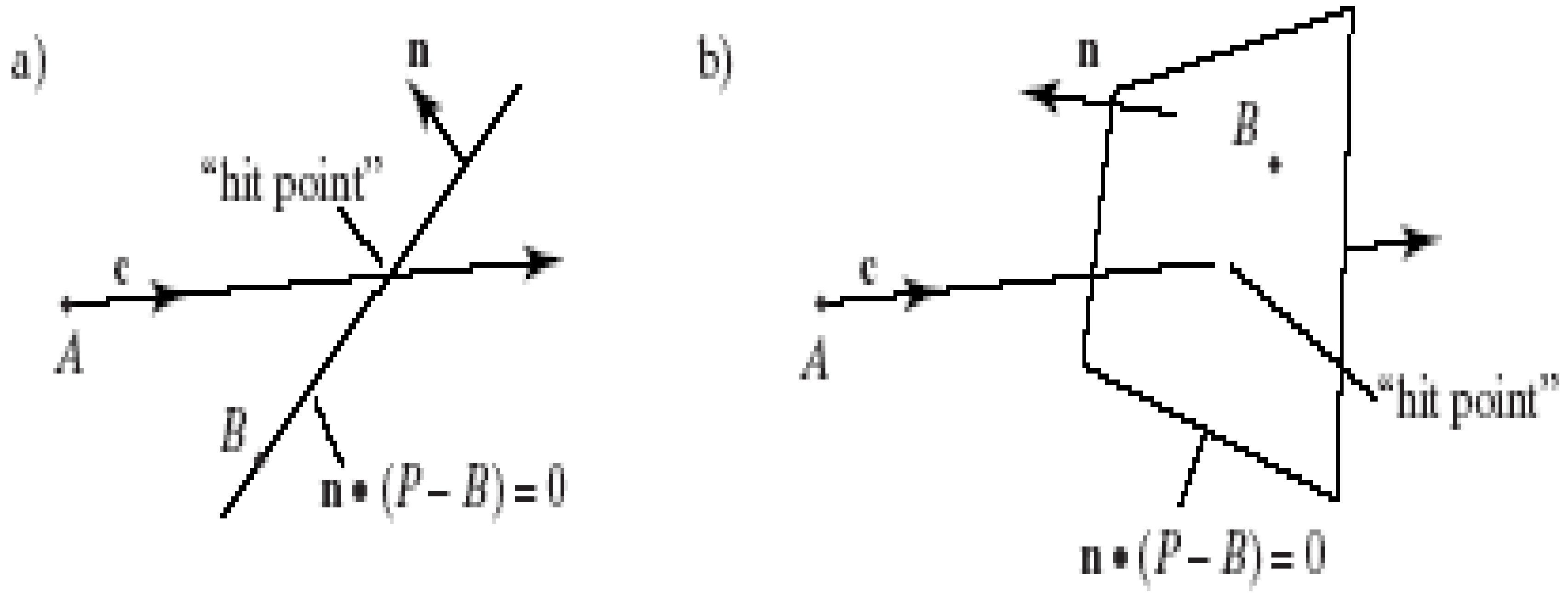


Clipping a Polygon (2)

- In Chapter 3, we looked at Cohen-Sutherland clipping of lines in a rectangular window, which involves the intersection of 2 lines.
- Here we will focus the Cyrus-Beck clipping algorithm, which clips polygons against lines and planes.

Intersections of Lines and Planes

- Intersections of a line and a line or plane are used in ray-tracing and 3D clipping: we want to find the "hit point".


a)
n
"hit point"
c
A
B
$\mathbf{n} \bullet (P - B) = 0$
b)
n
B
c
A
"hit point"
$\mathbf{n} \bullet (P - B) = 0$

Intersections of Lines and Planes (2)

- Suppose the ray hits at $t = t_{hit}$, the **hit time**.
- At this value of t the ray and line or plane must have the same coordinates, so $A + \mathbf{c}\, t_{hit}$ must satisfy the equation of the point normal form for the line or plane, $\mathbf{n}\cdot(P - B) = 0$.
- When the ray intersects (hits) the line or plane, $A + \mathbf{c}t_{hit} = P$, giving $\mathbf{n}\cdot(A + \mathbf{c}t_{hit} - B) = 0$.

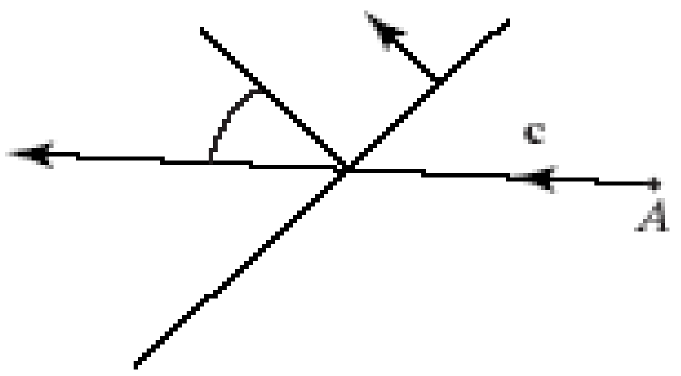
Intersections of Lines and Planes (3)

- Expanding and solving for t_{hit} gives $t_{hit} = \mathbf{n} \cdot (B - A) / \mathbf{n} \cdot \mathbf{c}$, if $\mathbf{n} \cdot \mathbf{c} \neq 0$.
  – If $\mathbf{n} \cdot \mathbf{c} = 0$, the line is parallel to the plane and there is no intersection.
- To find the hit point P_{hit}, substitute t_{hit} into the representation of the ray: $P_{hit} = A + \mathbf{c} t_{hit}$ $= = A + \mathbf{c}(\mathbf{n} \cdot (B - A) / \mathbf{n} \cdot \mathbf{c})$.

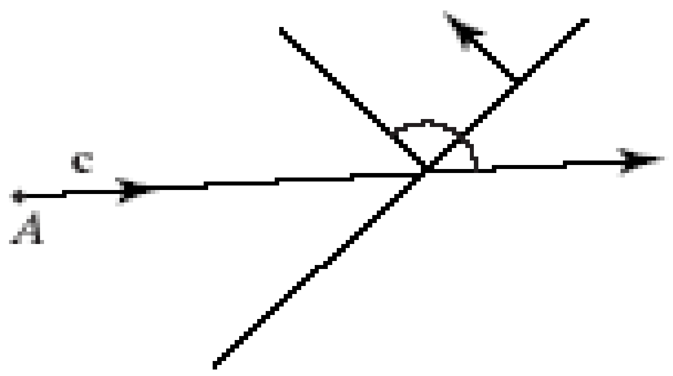
Direction of Ray

- If **n·c** = 0, the ray is parallel to the line.
- If **n·c** > 0, **c** and **n** make an angle of less than 90^o with each other.
- If **n·c** < 0, **c** and **n** make an angle of more than 90^o with each other.

a) ray is aimed "along with" **n**

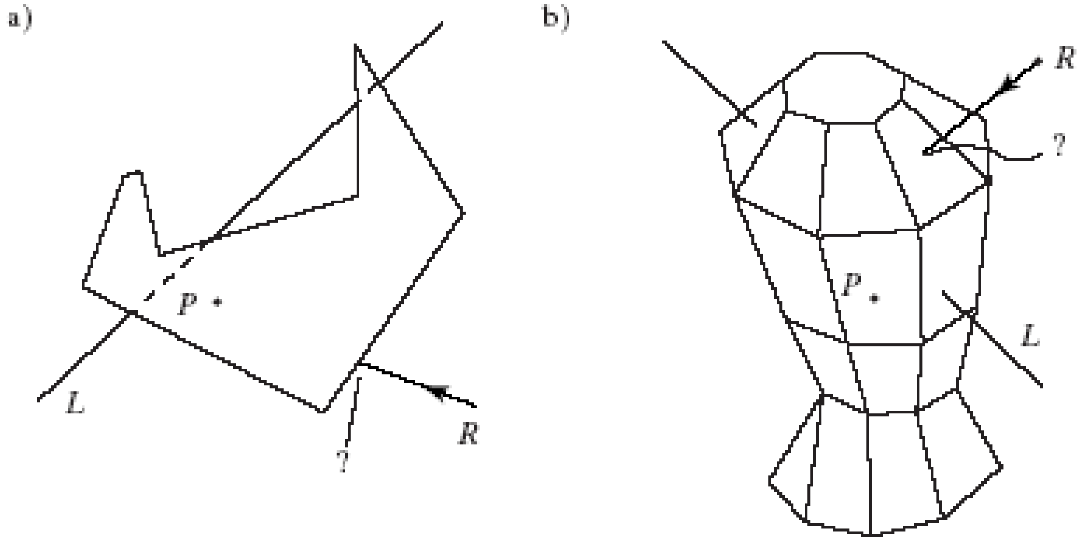


b) ray is aimed "counter to" **n**



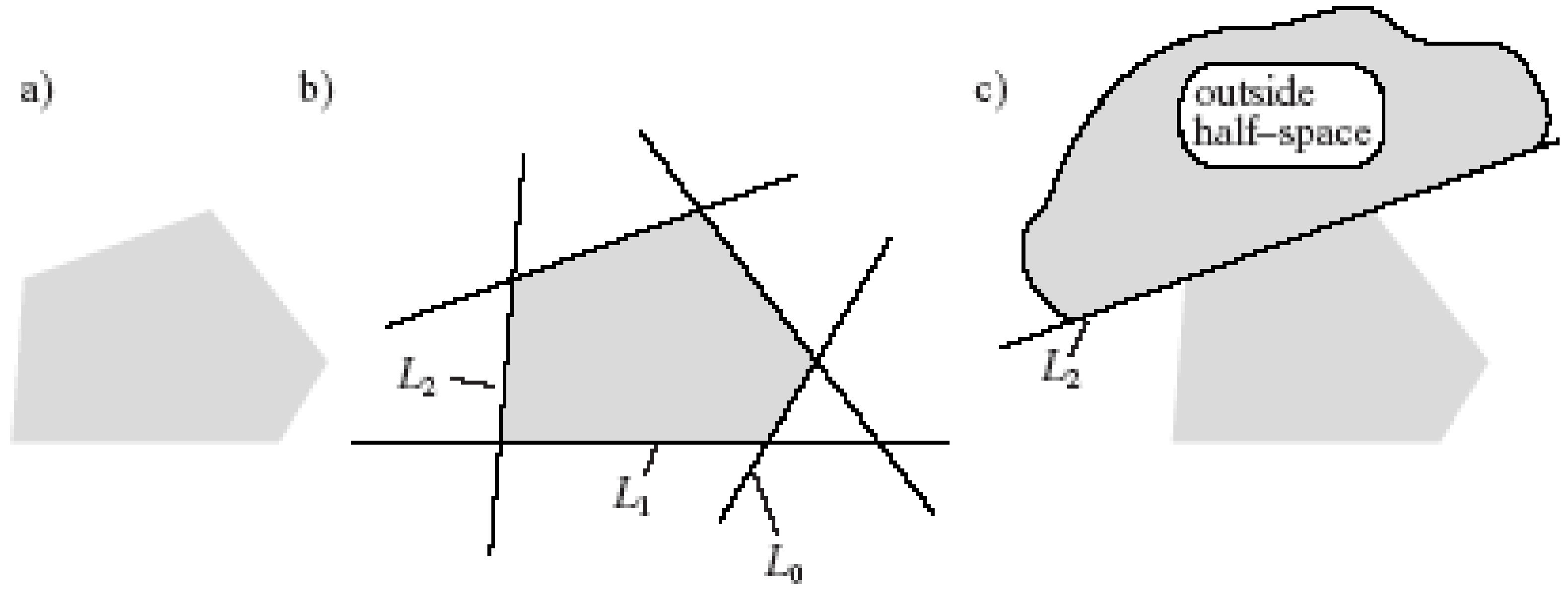
Polygon Intersection Problems

- Is a given point inside or outside the polygon?
- Where does a given ray first intersect the polygon?
- Which part of a given line L lies inside the object, and which outside?



Polygon Intersection Problems (2)

- Convex Polygons and Polyhedra: a simple case
- They are described by a set of bounding lines/planes, and the entire polygon/polyhedron is on one side of the line/plane.



Polygon Intersection Problems (3)

- The line/plane divides space into two halves: the outside space, which shares no points with the polyhedron, and the inside half space, where the polyhedron lies.
- The polyhedron is the intersection of all the inside half spaces.

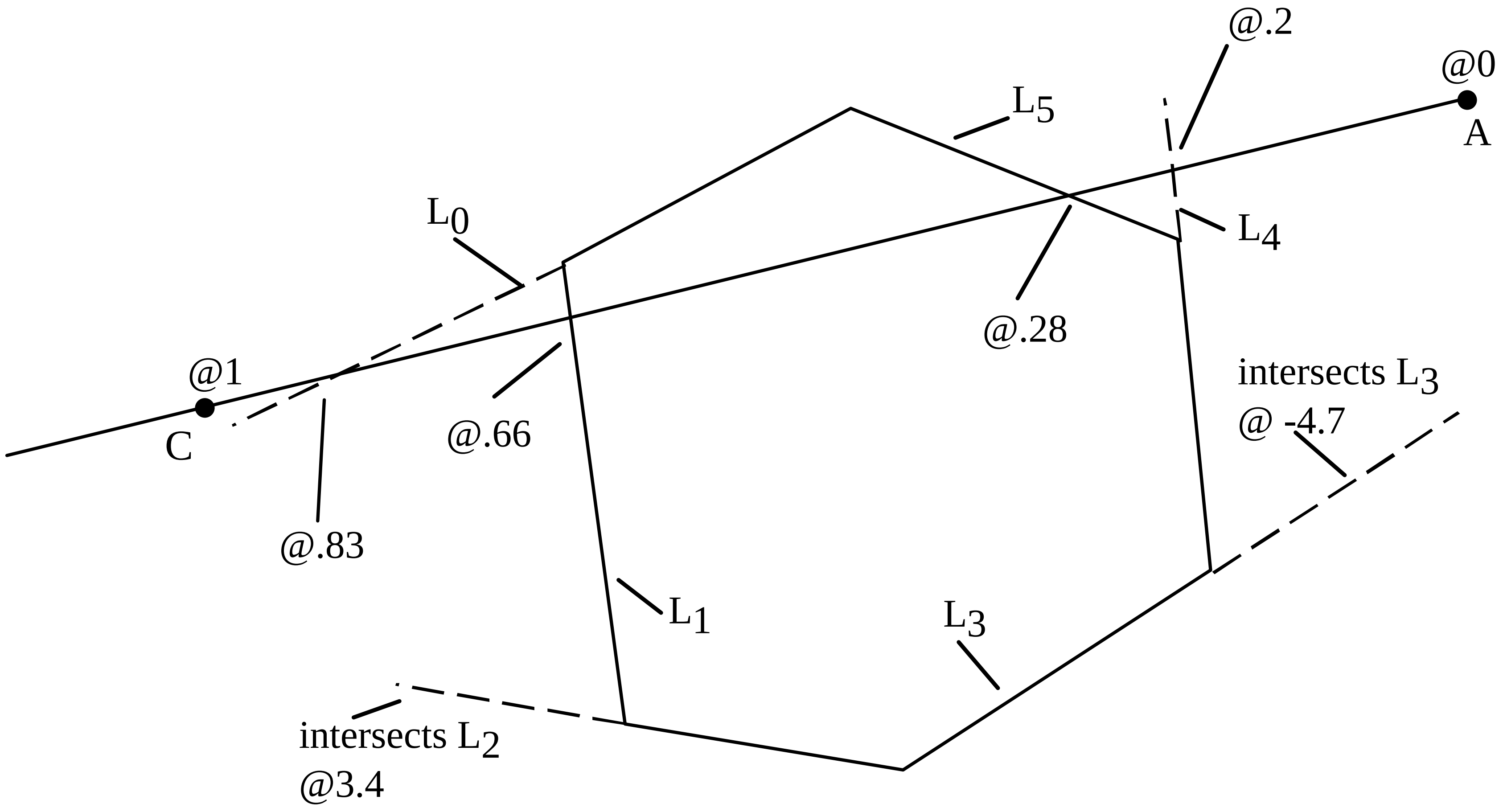
Ray Intersection Problem

- Where does the ray A + $\mathbf{c}t$ hit convex polygon P?
- For each bounding line, we find the (point-normal form of the) equation and the outside normal to this line.
  - If we traverse vertices counterclockwise, the outside normal will always be the clockwise normal, $(v_y, -v_x)$.
- Because of convexity, each line hits the polygon twice: going in, and coming out.

Ray Intersection Problem (2)

- For a line A + **c**t, find its intersections with each of the boundary lines: $t_{hit} = \mathbf{n}\cdot(B - A)/\mathbf{n}\cdot\mathbf{c}$, if $\mathbf{n}\cdot\mathbf{c} \neq 0$.
- If $\mathbf{n}\cdot\mathbf{c} > 0$, the line is entering the polygon. Set $t_{in} = \max(t_{hit}, t_{in})$.
- If $\mathbf{n}\cdot\mathbf{c} < 0$, the line is leaving the polygon. Set $t_{out} = \min(t_{hit}, t_{out})$.
- If $t_{in} > t_{out}$ the ray misses the polygon entirely. Otherwise, the ray is inside the polygon during $[t_{in}, t_{out}]$.

Example



Example (2)

- The sequence of updates to T_{in} and T_{out} as the various line intersections are tested.

Line tested	T_{in}	T_{out}
0	0	0.83
1	0	0.66
2	0	0.66
3	0	0.66
4	0.2	0.66
5	0.28	0.66

Inside-Outside Tests

- Is point P inside or outside the polygon?
  - Form a vector **u** = D – P, where D is any point outside the polygon, and **u** intersects no vertices of the polygon.
  - Create **n** normal to **u**. Let w = 0 be an int (the winding number.)
  - Let **E** be the vector along an edge crossed by **u**. Calculate d = **E·n**: if d > 0, add 1 to w; if d < 0, subtract 1 from w.
  - Repeat for each edge crossed by **u**. If at the end, w = 0, P is outside; else P is inside.

Inside-Outside Tests (2)

- Equivalent method: create **u** as before.
  - Calculate t_{in}, t_{out}, and t_P (time **u** reaches P).
  - If $t_{in} \le t_P \le t_{out}$, P is inside; else it is outside.

Cyrus-Beck Clipping

- Cyrus-Beck clipping clips a line segment against any convex polygon P:
- int CyrusBeckClip(Line& seg, LineList& L) uses parameters seg (the line segment to be clipped, which contains the first and second endpoints named seg.first and seg.second), and the list L of bounding lines of the polygon.
- It clips seg against each line in *L*, and places the clipped segment back in seg. (This is why seg must be passed by reference.)
- The routine returns 0 if no part of the segment lies in *P* or 1 if some part of the segment does lie in *P*.

Cyrus-Beck Pseudo-code (2D)

- Variables *numer* and *denom* hold the numerator and denominator for t_{hit}:
  - Numer = **n**·(B – A), denom = **n·c**

```
int CyrusBeckClip(LineSegment& seg, LineList& L)
{  double numer, denom;
   double tIn = 0.0, tOut = 1.0;
   Vector2 c, tmp;
    <form vector: c = seg.second - seg.first>
```

Cyrus-Beck Pseudo-code (2)

```
for(int i = 0; i < L.num; i++) // chop at each
  bounding line
  {  <form vector tmp = L.line[i].pt - first>
     numer = dot(L.line[i].norm, tmp);
     denom = dot(L.line[i].norm, c);
     if (!chopCI(numer, denom, tIn, tOut)) return 0;
       // early out
  }
 // adjust the endpoints of the segment; do second
  one 1st.
```

Cyrus-Beck Pseudo-code (3)

```
    if (tOut < 1.0 ) // second endpoint was altered
    {     seg.second.x  = seg.first.x + c.x * tOut;
          seg.second.y  = seg.first.y + c.y * tOut;
    }
    if (tIn > 0.0)  // first endpoint was altered
    {     seg.first.x  = seg.first.x + c.x * tIn;
          seg.first.y  = seg.first.y + c.y * tIn;
    }
    return 1; // some segment survives
}
```

Cyrus-Beck Pseudo-code (4)

- The routine chopCI() uses numer and denom to calculate the hit time at which the ray hits a bounding line, determines whether the ray is entering or exiting the polygon, and chops off the piece of the candidate interval CI that is thereby found to be outside the polygon.

Cyrus-Beck Pseudo-code (5)

```
int chopCI(double& tIn, double& tOut, double
   numer, double denom)
{  double tHit;
   if (denom < 0)              // ray is entering
   {    tHit = numer / denom;
        if (tHit > tOut) return 0;  // early out
        else if (tHit > tIn) tIn = tHit; // take larger t
   }
```

Cyrus-Beck Pseudo-code (6)

```
else if (denom > 0)          // ray is exiting
   {    tHit = numer / denom;
        if(tHit < tIn) return 0;      // early out
        if(tHit < tOut) tOut = tHit; // take smaller t
   }
   else              // denom is 0: ray is parallel
   if (numer <= 0) return 0;   // missed the line
   return 1;     // CI is still non-empty
}
```

3D Cyrus-Beck Clipping

- The Cyrus Beck clipping algorithm works in three dimensions in exactly the same way.
- In 3D the edges of the window become planes defining a convex region in three dimensions, and the line segment is a line in 3D space.
- ChopCI() needs no changes at all (since it uses only the values of dot products).
- The data types in CyrusBeckClip() must of course be extended to 3D types, and when the endpoints of the line are adjusted the z-component must be adjusted as well.

More Advanced Clipping

Algorithm	**OpenGL Yes/No?**	**Description**
Cohen Sutherland	Yes	Line segments against a rectangle or cube (2D: square)
Cyrus-Beck	Tries; poor results	Line against a convex polygon
Sutherland–Hodgman	No	Any polygon (convex or non-convex) against any convex polygon
Weiler–Atherton	No	Any polygon against any polygon

Sutherland-Hodgman Clipping

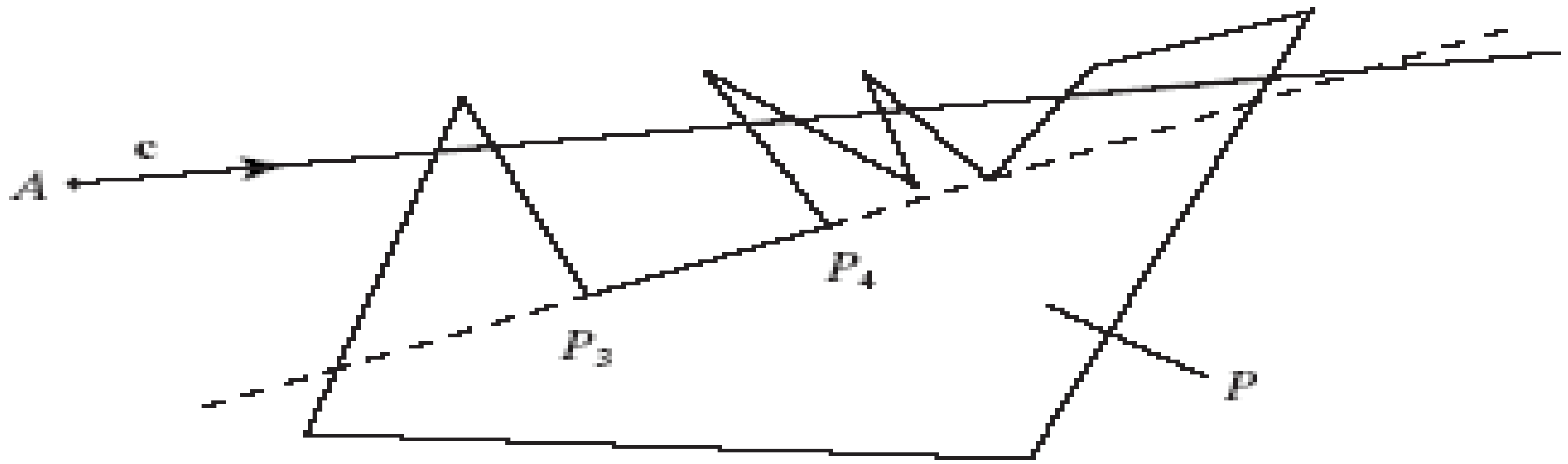
- The **Sutherland–Hodgman** clipper is similar to the Cyrus–Beck method, clipping against a convex polygon.
- It clips an entire polygon (which need not be convex) against the convex polygon. Its output is a *polygon* (or a set of polygons).
- It can be important to retain the polygon structure during clipping, since the clipped polygons may need to be filled with a pattern or color.
- This is not possible if the edges of the polygon are clipped individually.

Weiler-Atherton Clipping

- The **Weiler–Atherton** clipping algorithm clips any polygon, *P*, against *any* other polygon, *W*, convex or not.
- It can output the part of *P* that lies inside *W* (**interior clipping**) or the part of *P* that lies outside W (**exterior clipping**).
- In addition, both *P* and *W* can have holes in them.
- This algorithm is much more complex than the others and a thorough discussion is beyond the scope of this course.

Clipping for Arbitrary Polygons

- Much harder than for convex polygons.
  - Line may intersect polygon any even number of times, in in/out pairs.
  - If a line passes through a vertex, the vertex is counted as a pair of points (preferred) or as no intersection.

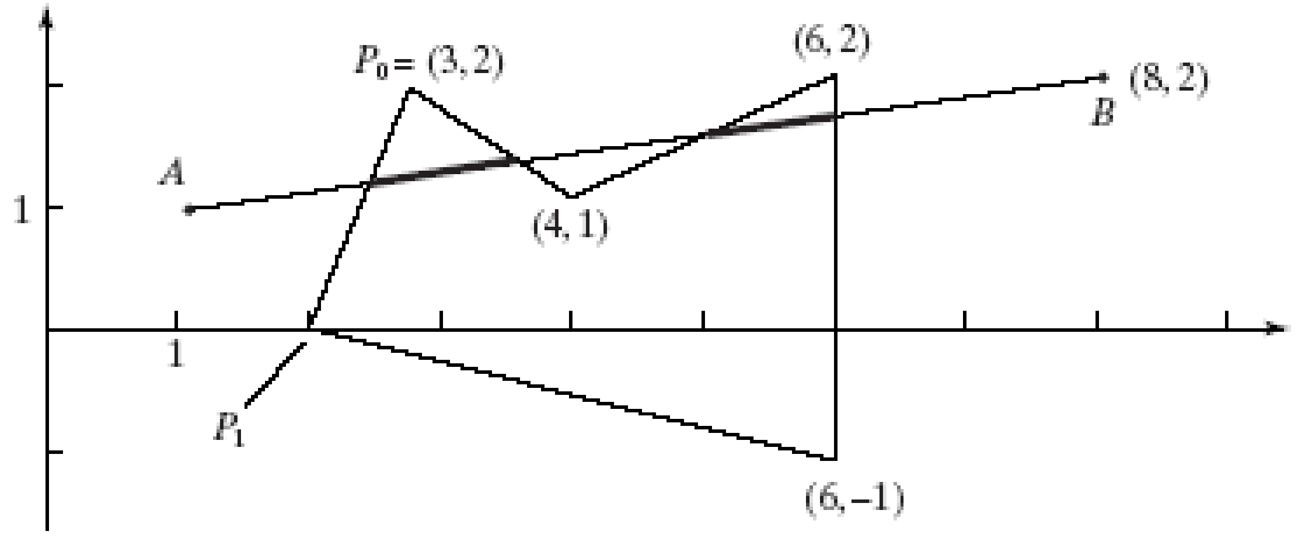


Clipping for Arbitrary Polygons (2)

- We create a list of hit times for the given line, A + **c**t, with the polygon edges: the edge from P_{i+1} to P_i is given by P_i + $\mathbf{e}_i$u, where $\mathbf{e}_i = P_{i+1} - P_i$, and $0 \le i \le N$.
- Because the polygon is closed, $P_N = P_0$.
- $t_{hit} = \mathbf{e}_i^{\perp} \cdot \mathbf{b}_i / \mathbf{e}_i^{\perp} \cdot \mathbf{c}$, where $\mathbf{b}_i = P_i - A$.
- $u = \mathbf{c}^{\perp} \cdot \mathbf{b}_i / \mathbf{e}_i^{\perp} \cdot \mathbf{c}$
- True hits occur only for $0 \le u \le 1$.

Clipping for Arbitrary Polygons (3)

- We sort the list of hits by increasing t_{hit} times and take successive pairs as representing an in/out time. (In other words, the line is inside between the first pair of times, the second pair of times, etc.)



$P_0 = (3, 2)$
(6, 2)
(8, 2)
B
A
1
(4, 1)
1
P_1
(6, –1)