

Hw notes

- Loop Boundary - ignore edge vertex near boundary rule. use the same edge vertex rule.
- Deadline changed to **Friday night(7/20) 11:59p**
- Selection - extra credit.
- Face removal is not! - make sure you support some kind of face removal (random, in order with each click, read index from cmd line etc).
- Make sure to use the second data structure we talked about - based on half-edges
- Exercise

Exercise

- Tetrahedron

v 1. 1. 1.

v 2. 1. 1.

v 1. 2. 1.

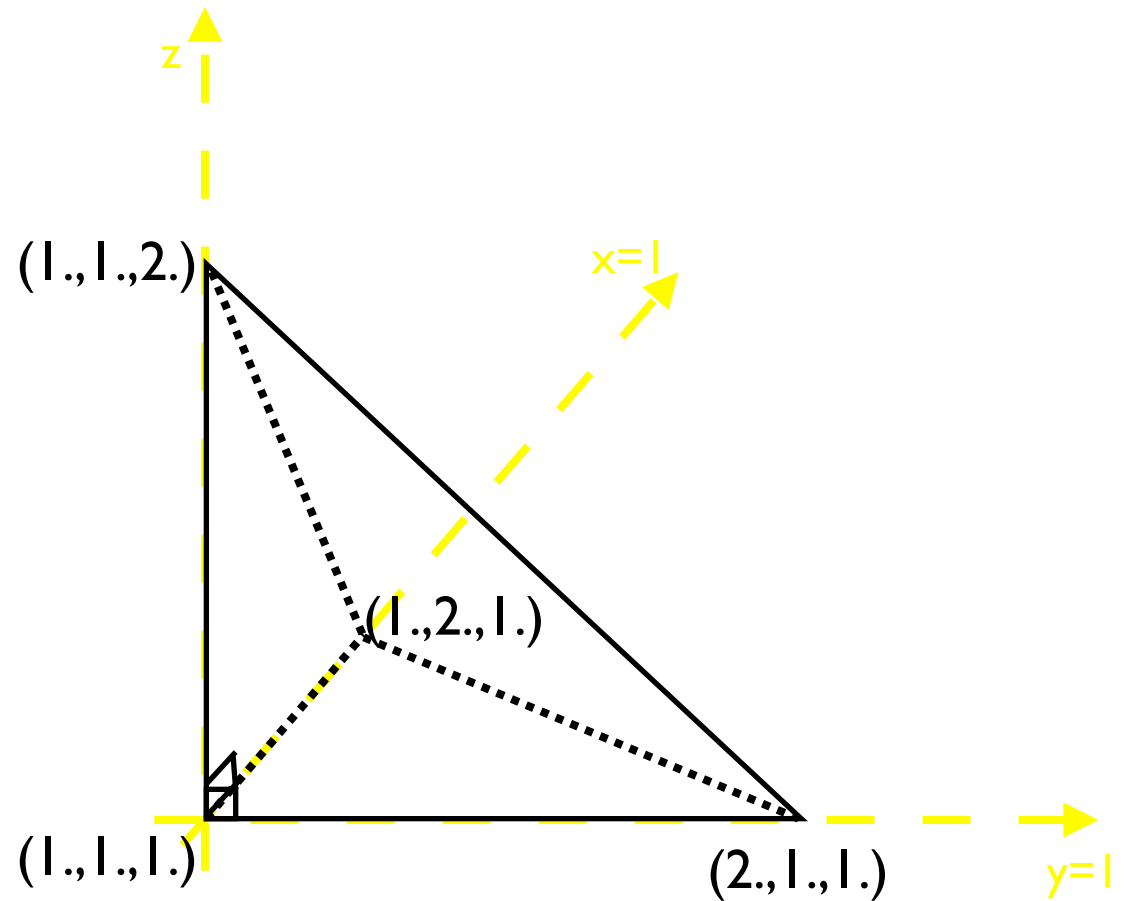
v 1. 1. 2.

f 1 3 2

f 1 4 3

f 1 2 4

f 2 3 4



Half Edge he

vertex * v

face * f

he * n

he * s

Vertex v

pos (x,y,z)

he * he

Face f

indices (v1, v2,..., vn)

he * he

Constructing a Half Edge Based Mesh Data Structure

Given: List of vertices, list of faces (vertices in ccw order)

Step 1: Create arrays of vertices, faces, and half-edges & initialize all pointers to zero.

Half-edges : assign one to every consecutive vertex pair in each face list

f: $[v_1 \ v_2 \ v_3 \ \dots \ v_n]$ he: $[v_1v_2], [v_2v_3], \dots, [v_{n-1}v_n]$

Constructing a Half Edge Based Mesh Data Structure

Step 2: Fill edge map

```
for each face f[i] with n vertices [v1 v2 ... vn]  
  f[i].he = [v1,v2]  
  for each vertex vj of face  
    if vj.he != 0 vj.he = [vj vj+1]  
  endfor  
  for each halfedge hej =[vkvm]of face f[i]  
    hej.face = f[i]  
    hej.next = next halfedge hej+1 of face  
    hej.vertex = vm  
    edgemap(vk,vm) = hej  
  endfor  
endfor
```

Constructing a Half Edge Based Mesh Data Structure

Step 3:

Go over all entries of edge map.

If both $\text{edgemap}(i, j)$ and $\text{edgemap}(j, i)$ exist

fill sym fields for both s.t. they point to each other

Lecture 10

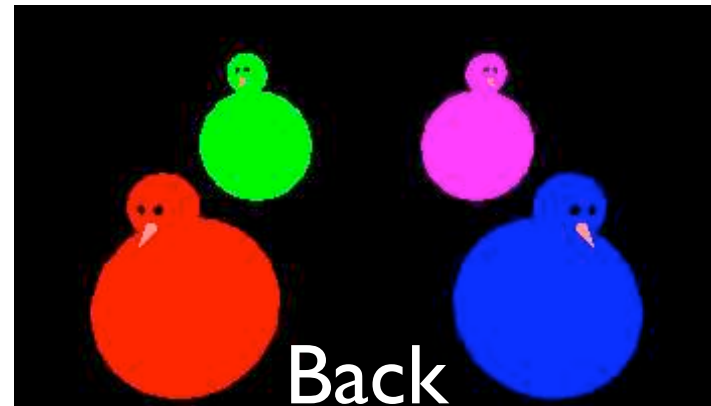
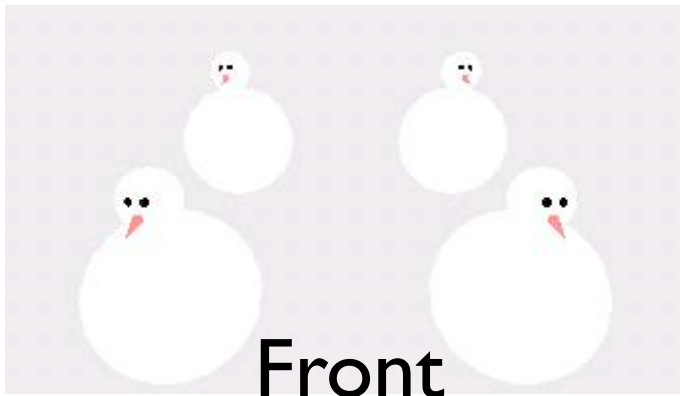
Picking
Ray Casting / Tracing

Picking

- Using OpenGL Selection
- Using Back Buffer

Using Back Buffer

- Need to be in double-buffer mode
- When the user picks an object scene gets drawn in back buffer - object identifiers in colors



<http://www.lighthouse3d.com/opengl/picking/index.php3?color1>

- Read the pixel under mouse - R,G,B values gives object number

- In `draw()`: if in picking mode
 1. disable lighting/textures/dithering
 2. pick a color for each object ID.
 3. draw object

- In `mouseClick()`
 1. select back buffer
 2. draw scene in picking mode
 3. read pixels under mouse
 4. revert value to integer

- In `draw()`: if in picking mode

1. disable lighting/textures/dithering

```
glDisable(GL_LIGHTING);
```

2. pick a color for each object:

e.g.

```
unsigned int address=(unsigned int) face;
```

```
GLubyte *bytes = (GLubyte *) &address;
```

```
glColor4ub(bytes[0],bytes[1],bytes[2],bytes[3]);
```

3. draw object

- In `mouseClick()`

1. select back buffer `glDrawBuffer(GL_BACK)`

2. draw scene in picking mode

3. read pixels under mouse

```
glReadPixels(x,h-y,1,1, GL_RGBA, GL_UNSIGNED_BYTE, &value)
```

4. revert value to integer

```
uint* ptr = (uint *)& value;  
int faceno = ptr[0];
```

Properties

- Automatically picks object in front
- Very easy to implement
- User never sees false colors - in back buffer (one should never swap buffers)
- Need to set colors using ubytes(avoid rounding)
- If static image, can keep a copy of once-drawn backbuffer instead of re-drawing it with each click.
- attention to uniqueness of color.

Using OpenGL Selection

3 modes of opengl:

1. Render

2. Select

3. Feedback



Drawing information
returned to application
instead of frame buffer



Screen stays put

We'll use selection
mode for picking

Using OpenGL Selection

- 1) Enter selection mode
- 2) Designate a small “pickable region” around the mouse
- 3) "Draw" the scene also assigning "names" to objects
- 3) Leave selection mode and enter render mode - info will be returned
- 4) Process the returned information.

In mouseclick() call : checkHits(x,y)

```
glSelectBuffer( BUFSIZE, selectBuf );  
glRenderMode( GL_SELECT );  
glInitNames();
```

```
glMatrixMode( GL_PROJECTION );  
glPushMatrix();  
    setupCameraView();  
    glMatrixMode( GL_PROJECTION );  
    glGetDoublev( GL_PROJECTION_MATRIX, matrix );  
    glLoadIdentity();
```

```
    glGetIntegerv( GL_VIEWPORT, viewport );  
    gluPickMatrix( (double)x, (double)( viewport[3]-y),  
                  X_PICK_SIZE, Y_PICK_SIZE, viewport);  
    glMultMatrixd( matrix );
```

```
    drawScene();  
    glMatrixMode( GL_PROJECTION )  
glPopMatrix();  
hits = glRenderMode( GL_RENDER );  
processHits( hits, selectBuf );
```

In drawScene()

```
glMatrixMode( GL_MODELVIEW );
```

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
glColor3f( 1.0, 1.0, 0.0 );
```

```
glPushName( OBJ_NAME );  
drawObj();  
glPopName();
```

```
glFlush();
```

In ProcessHits()

```
nameStackDepth = *ptr++;
zmin = *ptr++;
zmax = *ptr++;
nearestZ = zmin;
for( j = 0; j < nameStackDepth; j++ ) {
    currname = *ptr++;
}
nearestCurrname = currname;
//if more than 1 hit
for( i = 1; i < hits; i++ ) {
    nameStackDepth = *ptr++;
    zmin = *ptr++;
    zmax = *ptr++;
    for( j = 0; j < nameStackDepth; j++ ) {
        currname = *ptr++;
    }
    if (zmin < nearestZ) {
        nearestZ = zmin;
        nearestCurrname = currname;
    }
}
```

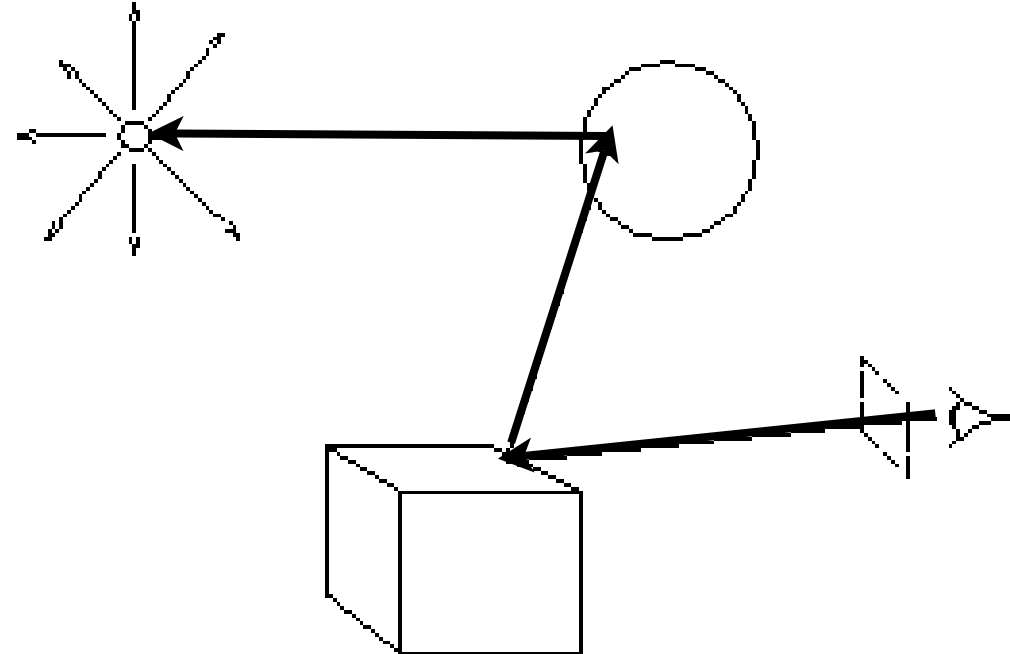
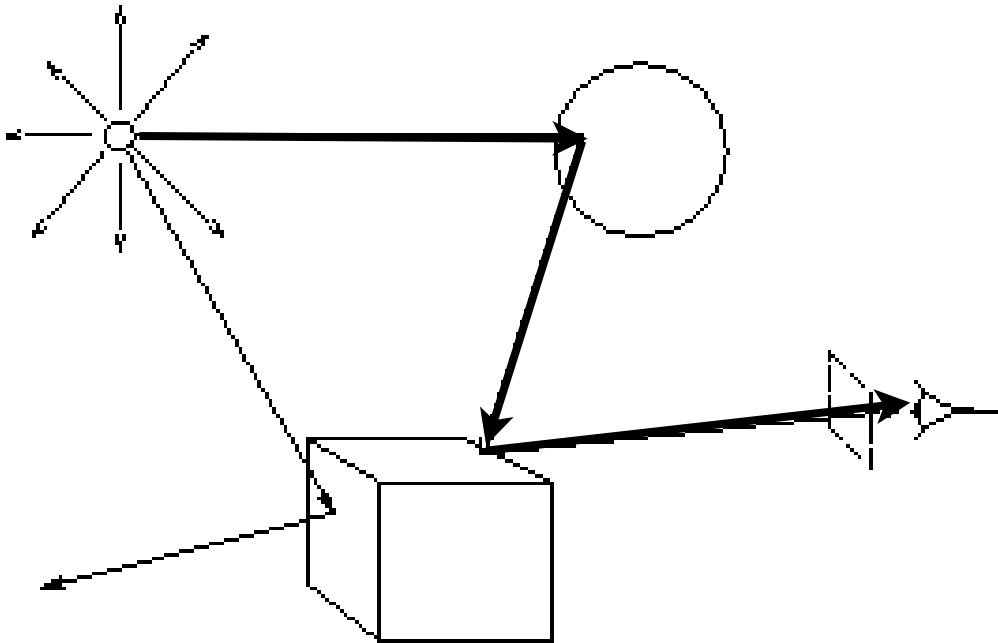
Properties

- Can be more complicated and/or cumbersome
- If one needs all objects underneath mouse location, the only way to go.
- One should do some extra comparisons to find front-most object.

Ray Tracing

Ray Tracing

- An approach for photo-realistic rendering of scenes
- Based on following light rays through the scene (backwards!)
- Iterate over pixels instead of primitives



Ray Tracing

- An approach for photo-realistic rendering of scenes
- Based on following light rays through the scene (backwards!)
- Iterate over pixels instead of primitives
- May be slow - if large nr of objects in scene (pixel blocks)
- Makes some effects easier: shadows reflections, transparency, procedural textures, objects, CSG
- Selection without need for extra buffers
- hidden surface removal comes for free

Ray Casting

Basic Algorithm

For each pixel {

 Shoot a ray from camera to pixel

 for all objects in scene

 Compute intersection with ray

 Find object with closest intersection

 Display color using object + light property

}

Ray Casting

Basic Algorithm

For each pixel {

Shoot a ray from camera to pixel

for all objects in scene

 Compute intersection with ray

Find object with closest intersection

Display color using object + light property

}

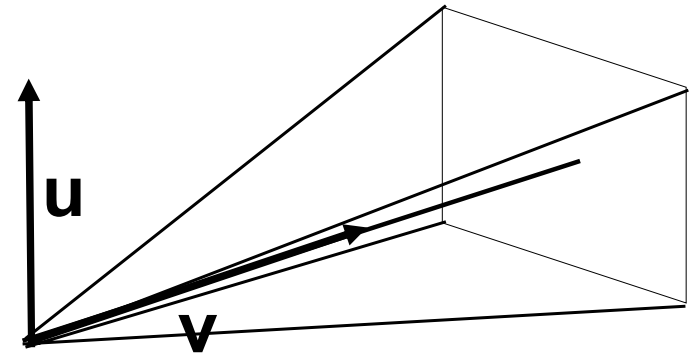
Rays

- ray = origin Point + propagation direction
- Parametric line: $p(t) = q + tv$
- write ray from camera c to a point $p(t)$ in scene as a line:
$$p(t) = c + t(\text{view_dir})$$
- Information:
 - for $t_i > 0$, if $t_1 < t_2 \Rightarrow p(t_1)$ closer to eye than $p(t_2)$
 - if $t < 0$, $p(t)$ behind the eye -- not visible

Pixel rays

Goal: Find direction of the ray to the center of the pixel (i,j). Let camera parameters be

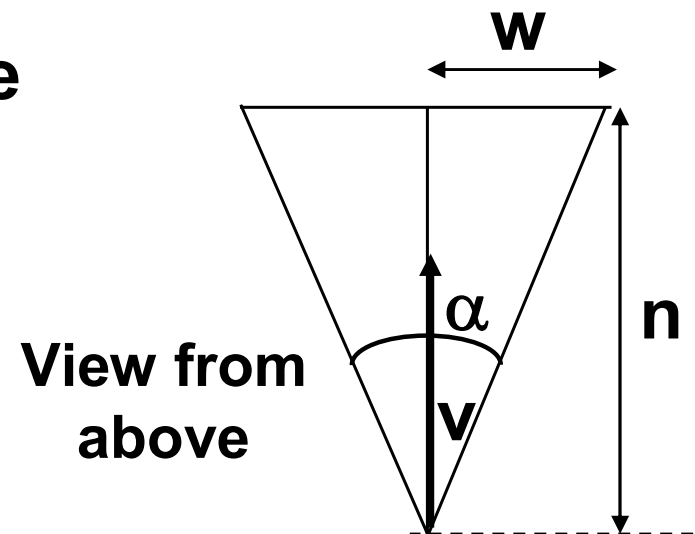
c position
 α horizontal field of view
v viewing direction
u up direction
s aspect ration



Then the image half-width in the “virtual world” units is

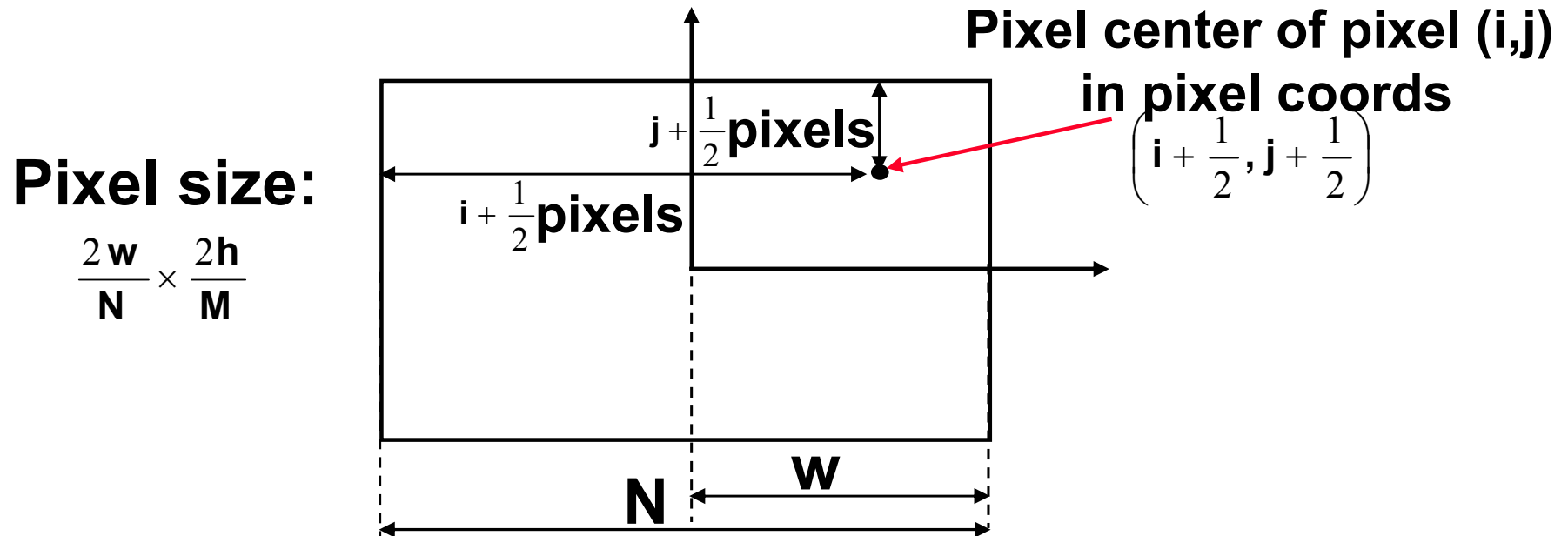
$$w = n \operatorname{tg} \frac{\alpha}{2}$$

The half-height is $h = sn \operatorname{tg} \frac{\alpha}{2}$



Pixel rays

From coordinates in pixel units to virtual world coordinates in image plane:



Displacements of the pixel from the image center in virtual space units:

$$h - \left(j + \frac{1}{2}\right) \frac{2h}{M}, \quad \left(i + \frac{1}{2}\right) \frac{2w}{N} - w$$

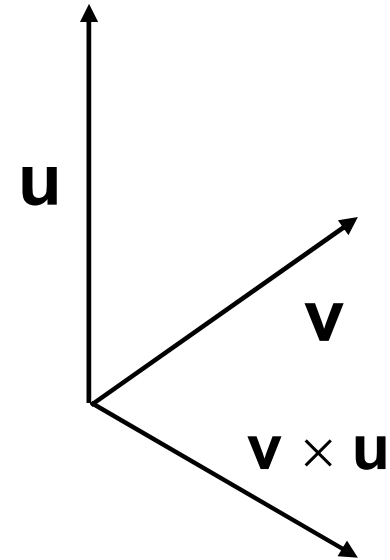
Pixel rays

Virtual world coordinates of pixel (i,j):
image center + displacements.

Image center: $\mathbf{c} + \mathbf{v}n$

pixel (i, j) = $\mathbf{c} + \mathbf{v}n +$

$$\left(\mathbf{h} - \left(\mathbf{j} + \frac{1}{2} \right) \frac{2\mathbf{h}}{\mathbf{M}} \right) \mathbf{u} + \left(\left(\mathbf{i} + \frac{1}{2} \right) \frac{2\mathbf{w}}{\mathbf{N}} - \mathbf{w} \right) \mathbf{v} \times \mathbf{u}$$



Ray Casting

Basic Algorithm

For each pixel {

Shoot a ray from camera to pixel

for all objects in scene

Compute intersection with ray

Find object with closest intersection

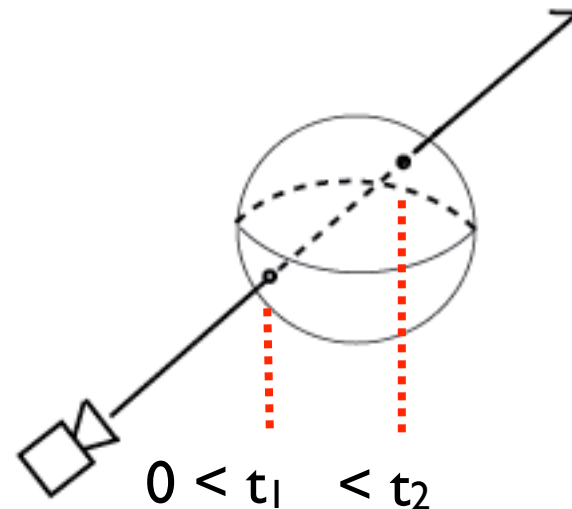
Display color using object + light property

}

Intersecting a Ray with an Object

General Approach:

- Use implicit equation for object: $F(q) = 0$
- Use parametric line equation for ray: $q = p + vt$
- Solve equation $F(p+vt) = 0$ for possible values of t .
- Take minimal non-negative t as intersection pt with that object.



Some primitives

Finite primitives:

- polygons
- spheres, cylinders, cones
- parts of general quadrics

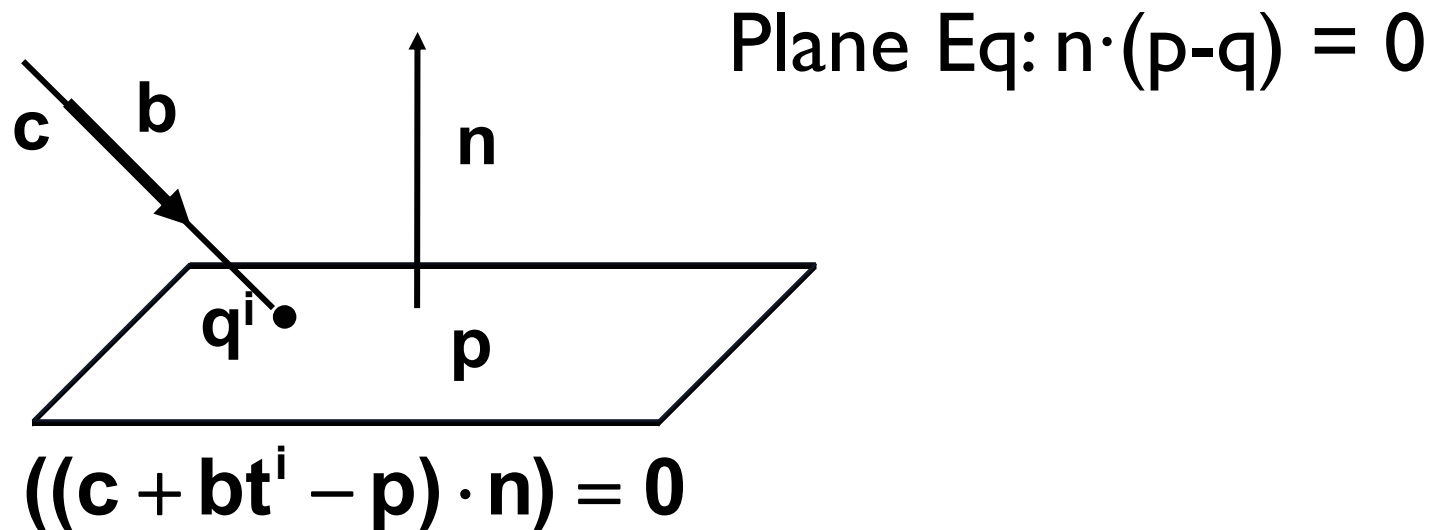
Infinite primitives:

- planes
- infinite cylinders and cones
- general quadrics

A finite primitive is often an intersection of an infinite with an area of space

Intersecting a line and a plane

Same old trick: use the parametric equation for the line, implicit for the plane. In the case of a pixel ray, $b = p(i,j) - c$



$$t^i = -\frac{((c - p) \cdot n)}{(b \cdot n)}$$

Check for zero in the denominator; t^i should be positive for the intersection to be in front of the camera.

Ray-Triangle Intersection

- Use barycentric coordinates.
- Match a point on the ray, to a point in the triangle

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

3 equations, 3 unknowns

Solution easy

Book (pg. 207-208) has Cramer's rule solution written out, ready for implementation.

Ray-Triangle Intersection

Algorithm

compute γ ;

if ($\gamma < 0$ or $\gamma > 1$)

 return false;

compute β ;

if ($\beta < 0$ or $\beta > 1 - \gamma$)

 return false;

compute t ;

return t as candidate;

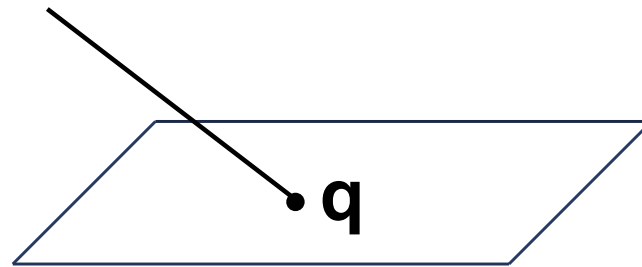
Polygon-ray intersections

Two steps:

- intersect with the plane of the polygon
- check if the intersection point is inside the polygon

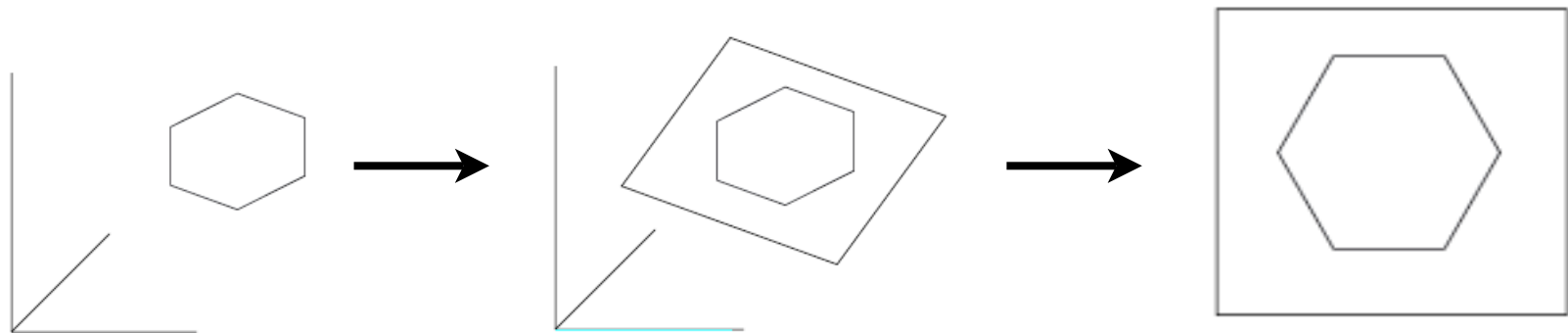
We know how to compute intersections with the plane

Let $q=[q_x, q_y, q_z]$
be the intersection
point



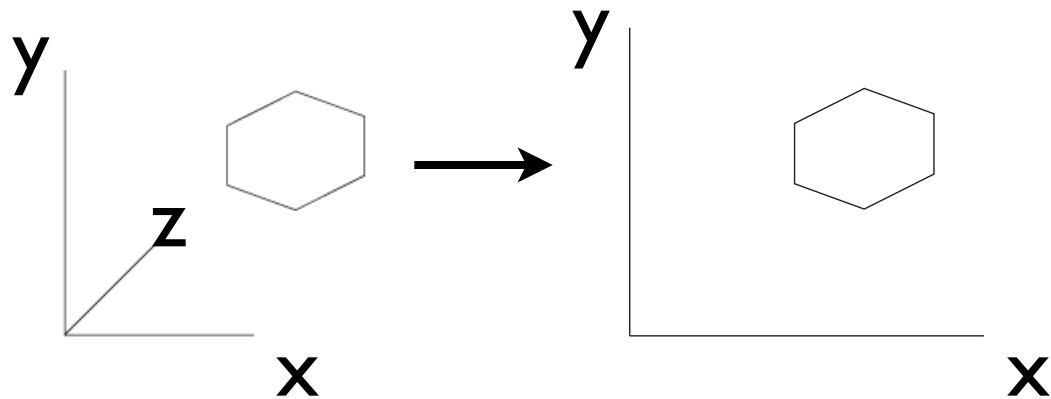
Polygon-Ray Intersections

- Move from 3D => 2D : More efficient!
- How?
 - Converting to coordinates on a plane difficult



Polygon-Ray Intersections

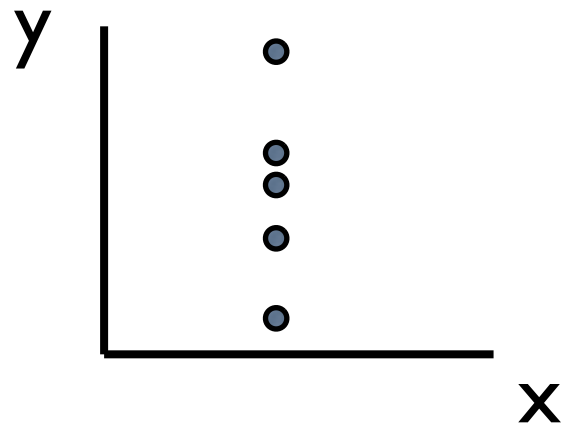
- Move from 3D \Rightarrow 2D : More efficient!
- How?
 - Converting to coordinates on a plane difficult
 - Project to a coord plane (XY, YZ, or XZ) (i.e. discard a coordinate)



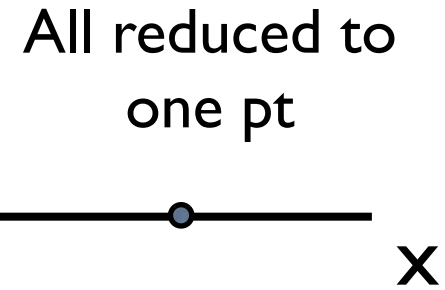
Polygon-Ray Intersections

- But cannot always discard the same coordinate.

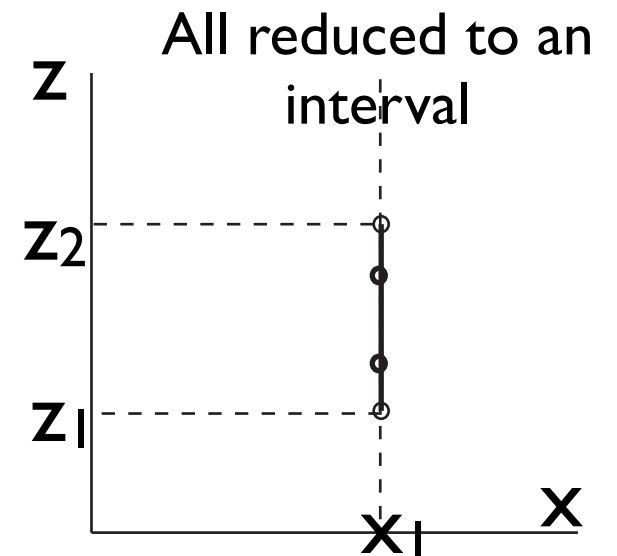
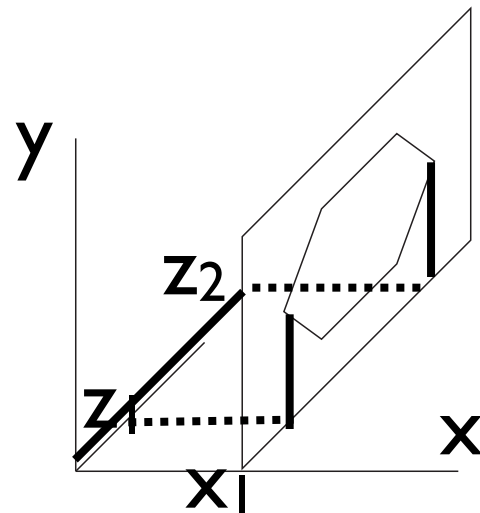
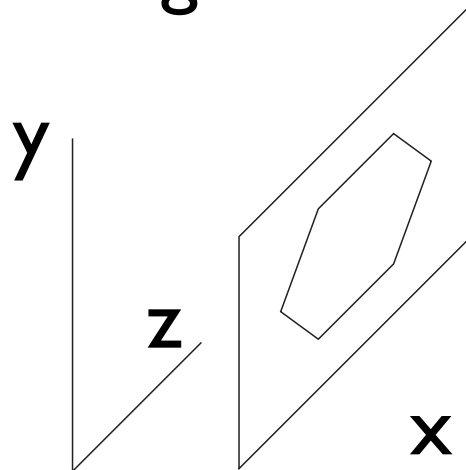
- e.g. 2D => 1D:



Discard y



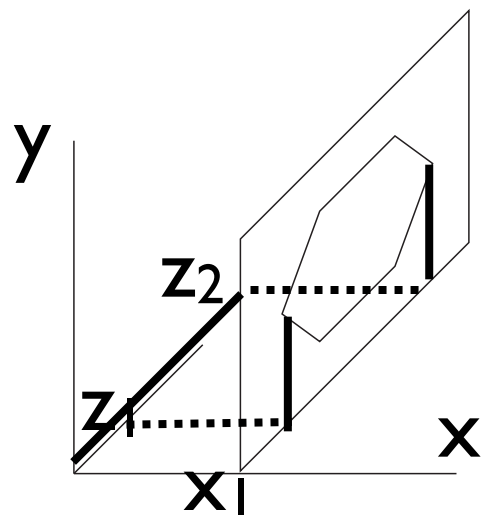
- e.g. 3D => 2D:



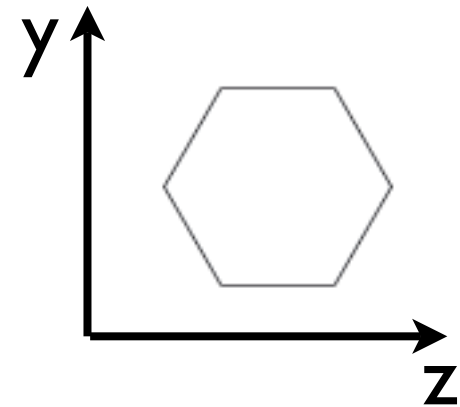
Polygon-Ray Intersections

Solution:

- Look at normal of polygon $n = [n_x, n_y, n_z]$
- Discard the coordinate with the largest component.
- E.g. if $n_x > n_y > n_z$, discard n_x .

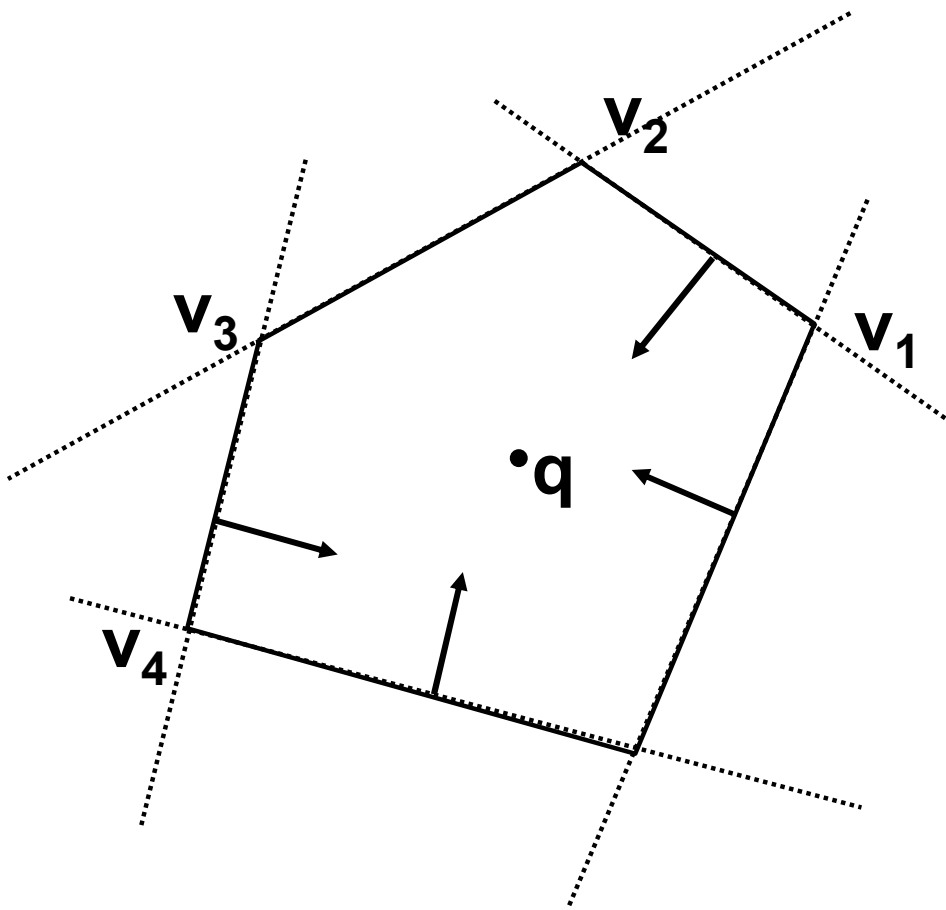


$n = [1, 0, 0]$
Discard X.



2D Polygon-ray intersections

Now we can assume that all vertices v_i and the intersection point q are 2D points.



Assume that polygons are convex. A convex polygon is the intersection of a set of half-planes, bounded by the lines along the polygon edges. To be inside the polygon the point has to be in each half-plane. Recall that the implicit line equation can be used to check on which side of the line a point is.

2D Polygon-ray intersections

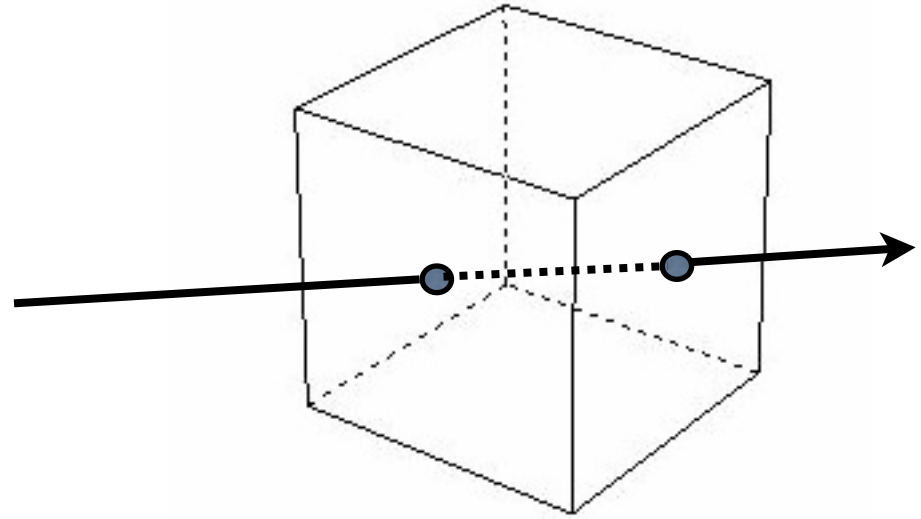
Equation of the line through the edge connecting vertices v_i and v_{i+1} :

$$\left(v_i^y - v_{i+1}^y\right)\left(x - v_i^x\right) + \left(v_{i+1}^x - v_i^x\right)\left(y - v_i^y\right) = 0$$

If the quantity on the right-hand side is positive, then the point (x,y) is to the left of the edge, assuming we are looking from v_i to v_{i+1} .

Algorithm: if for each edge the quantity above is nonnegative for $x = q^x$, $y = q^y$ then the point q is in the polygon. Otherwise, it is not.

In the formulas x and y should be replaced by x and z if y coord. was dropped, or by y and z if x was dropped.

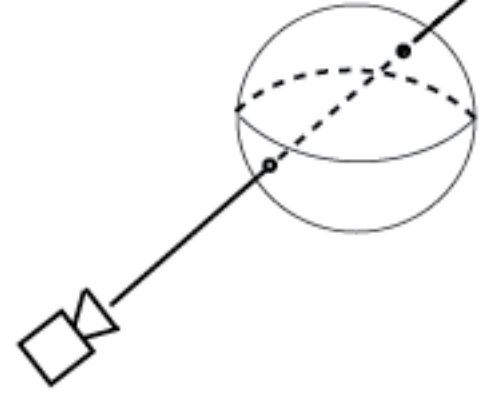


3D Box-Ray

- Can treat as 6 polygons.
- Keep track of all candidate intersection points
- Pick the one with smallest non-neg t .

Note that at this point, you can ray cast any mesh!

Intersecting a ray with a sphere



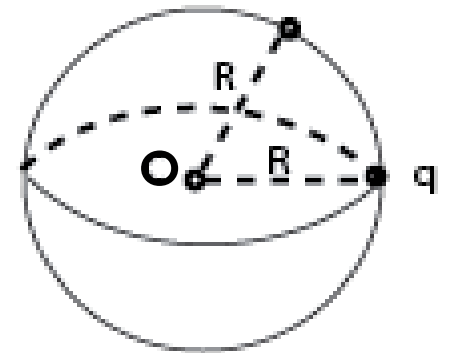
Sphere equation: $(q-o)^2 - r^2 = 0$

For a ray $q = c + bt^i$ we get $((c-o) + bt)^2 - r^2 = 0$

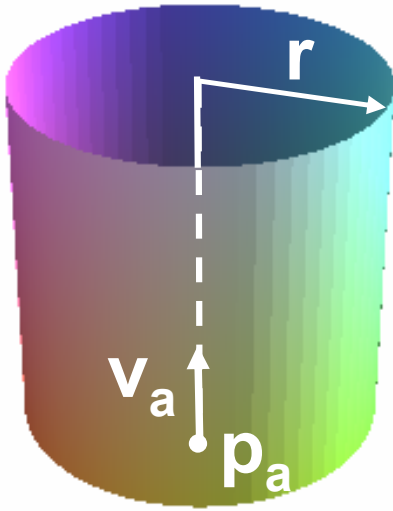
$$b^2t^2 + t(2b(c-o)) + ((c-o)^2 - r^2)$$

This quadratic equation in t may have no solutions (no intersection) or two (possibly coinciding) solutions (entry and exit points).

The correct point to return is the one that is closest to ray origin.



Infinite cylinder-ray intersections



Infinite cylinder along y axis of radius r has equation $x^2 + z^2 - r^2 = 0$.

The equation for a more general cylinder of radius r oriented along a line $p_a + v_a t$:

$$(q - p_a - (v_a \cdot (q - p_a)) v_a)^2 - r^2 = 0$$

where $q = (x, y, z)$ is a point on the cylinder.

Infinite cylinder-ray intersections

To find intersection points with a ray $c+bt$
substitute $q = c+bt$ and solve:

$$(c - p_a + bt - (v_a \cdot (c - p_a + bt) v_a))^2 - r^2 = 0$$

reduces to $At^2 + Bt + C = 0$

with

$$A = (b - (b \cdot v_a)v_a)^2$$

$$B = 2((b - (b \cdot v_a)v_a) \cdot (cp - (cp \cdot v_a)v_a))$$

$$C = (cp - (cp \cdot v_a)v_a)^2 - r^2$$

where $cp = c - p_a$