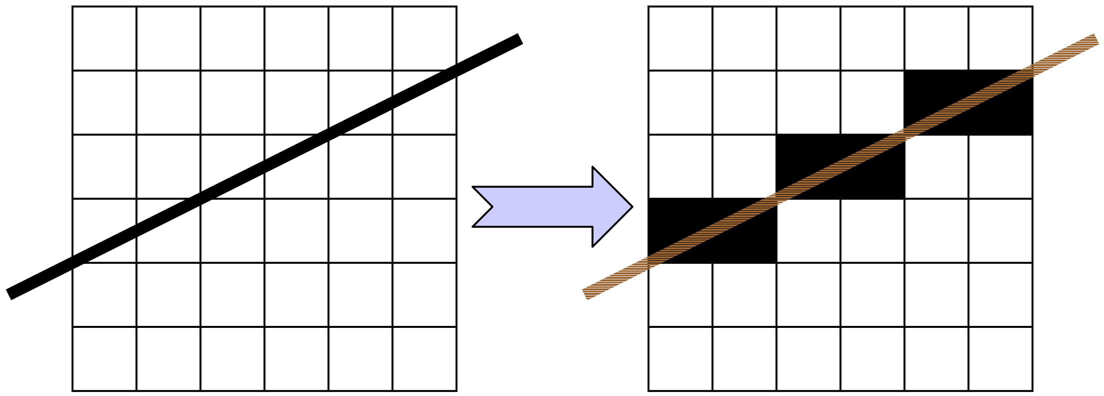
Topic #1: Rasterization (Scan Conversion)

We will generally model objects with geometric primitives

- points, lines, and polygons

For display, we need to convert them to pixels

- for points it's obvious
- but we'll need some algorithms for lines and polygons



General Comments on Rasterization

Moving from continuous geometry to discrete pixels is inexact

- we're attempting to approximate the primitive with pixels
- thus a certain amount of error is being introduced

Goal #1: Accuracy

- construct good approximations (i.e., low error)
- this can be hard because there may be many tricky cases

Goal #2: Efficiency

- this process is going to happen a lot
  - imagine we need to draw 10 million polygons/second
- one near-universal strategy: implement this stuff in hardware

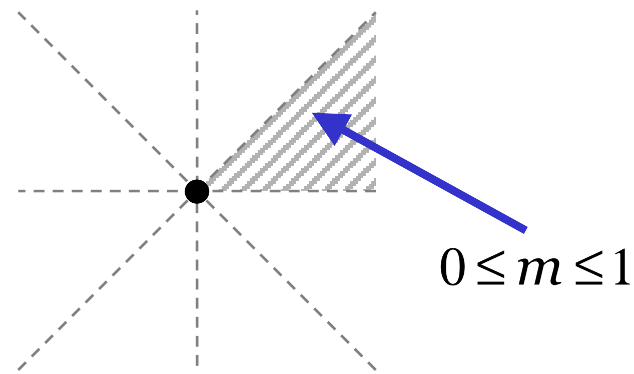
Line Rasterization

We have a 2-D line segment inside the viewport

- it's been projected & clipped

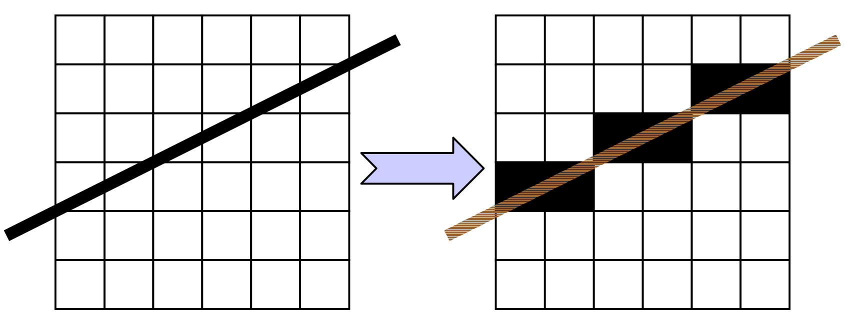
To simplify discussion, assume slope is between 0 and 1

- other cases are symmetric



Our goal, fill in pixels "on" line

- actually, most *nearly* on
- as measured at pixel centers



First Cut: Very Simple Line Algorithm

Compute equation of line

$$y = mx + b \quad \text{where } m = \frac{\Delta y}{\Delta x}$$

Now, start at the leftmost point and walk to the right

- in other words, increment x by 1 at each step
- for each x, compute y with equation
  - need to round y to integral coordinate
  - for instance, can use `rint(y)` or `floor(y + 0.5)`
- fill in pixel (x, y)

This is a correct algorithm, but it is inefficient

- requires floating point multiply/add/round for each pixel column

Fortunately, we can easily do better …

A More Efficient Incremental Algorithm

What does the slope of a line mean?

- it's the change in y for a unit change in x
- this is exactly what we need to know!

$$y(x+1) = m(x+1) + b = (mx+b) + m = y(x) + m$$

Again, let's start at leftmost point and walk to the right

- increment x by 1 at each step
- increment y by m at each step
- fill in pixel (x, *round*(y))

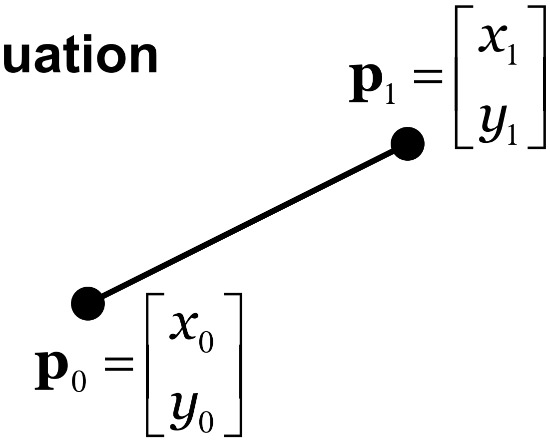
This has a fancy name: Digital Differential Analyzer (DDA)

Obviously better than our first try, but still rather inefficient

- we're still doing floating point add/round per pixel column

Bresenham's Algorithm (Midpoint Algorithm)

We'll switch to the implicit form of the line equation

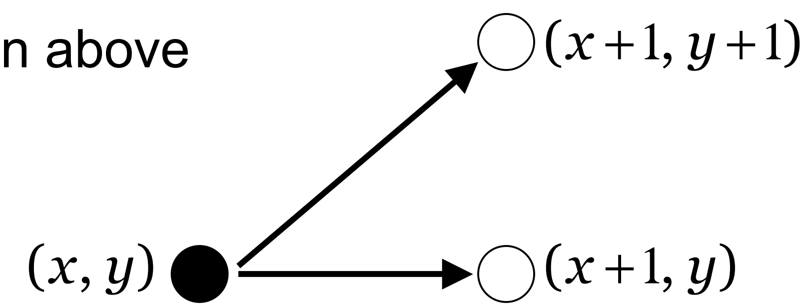


$$F(\mathbf{p}) = 2\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0 \qquad \text{where } \mathbf{n} = \begin{bmatrix} \Delta y \\ -\Delta x \end{bmatrix}$$

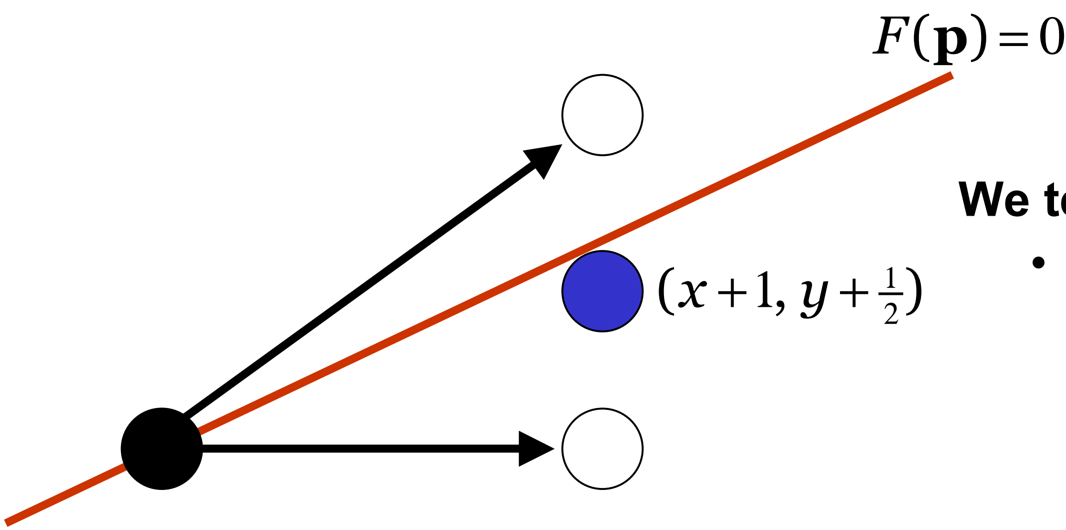
$$\Delta x = x_1 - x_0 \quad \text{and} \quad \Delta y = y_1 - y_0$$

For the next pixel, we either increment *x* or both *x*, *y*

- we want to pick the one closest to the line
- can do this with our line equation above



Selecting the Next Pixel



We test the midpoint

- evaluate F at midpoint

  >0 means it's below line

  ≤ 0 means it's on or above line

This tells us which pixel is closer

- and hence which one to pick

  >0 ➔increment x and y

  ≤0 ➔increment x only

The Key Insight

We can incrementalize this test of F

$$F(\mathbf{p}+\mathbf{d}) = 2\mathbf{n}\cdot(\mathbf{p}+\mathbf{d}-\mathbf{p}_0) = F(\mathbf{p}) + 2\mathbf{n}\cdot\mathbf{d}$$

where $\mathbf{d} = \begin{bmatrix}1\\0\end{bmatrix}$ or $\begin{bmatrix}1\\1\end{bmatrix}$

- note that the dot product can be precomputed
- incremental update of F requires a single integer addition!

So, we initially compute F at the beginning

- at each step, we use F to pick how to increment (x,y)
  - hence it is called the decision variable
- and it also tells us how to increment F

If $F > 0$

$$(x,y) \rightarrow (x+1, y+1)$$
$$F \rightarrow F + 2\Delta y - 2\Delta x$$

If $F \leq 0$

$$(x,y) \rightarrow (x+1, y)$$
$$F \rightarrow F + 2\Delta y$$

Bresenham's Line Algorithm in C

```
void line(int x0, int y0, int x1, int y1)
{
  int x = x0,  y = y0;
  int dx = x1-x0,  dy = y1-y0;
  int F = 2*dy-dx
  int incX = 2*dy,  incXY = 2*(dy-dx);

  for(x=x0; x<=x1; x++)
  {
    write_pixel(x, y);
    if( F<=0 )  { F += incX; }
    else        { F += incXY; y++; }
  }
}
```

Bresenham's (Midpoint) Algorithm for Circles

Can use the same methodology for drawing circles

- write the implicit equation of the circle

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

- derive decision variable scheme
- exploit 8-way symmetry — only need to compute 1 octant

And it even generalizes to other conic sections

- ellipses, parabolas, hyperbolas

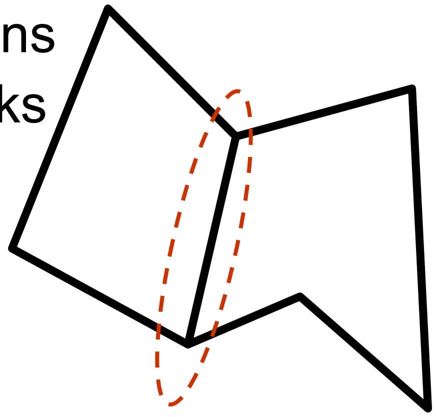
See textbook for algorithm details

Polygon Rasterization

We want to fill every pixel covered by the polygon

And we need to be really careful!

- suppose we have two adjacent polygons
- we don't want any overlap or any cracks
- visit every covered pixel exactly once



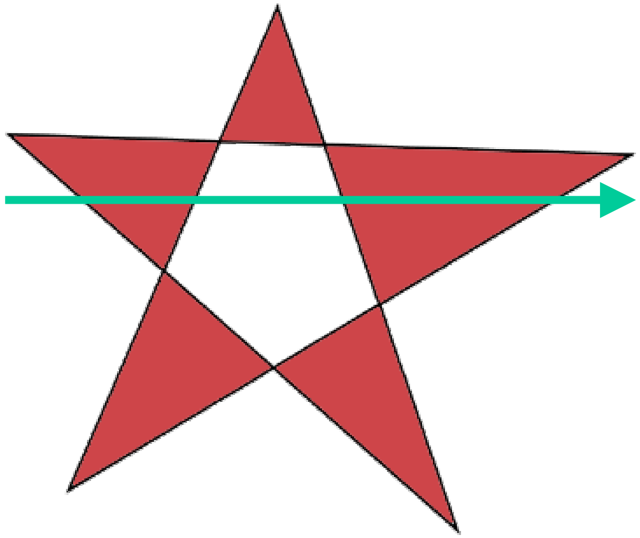
What's the Inside of a Polygon?

This is not obvious when the polygon intersects itself

- over time, people came up with some arbitrary definitions

Definition #1: Odd–even rule

- pass horizontal line through shape; points with odd # crossings are in
- this is the one generally used for polygon rasterization



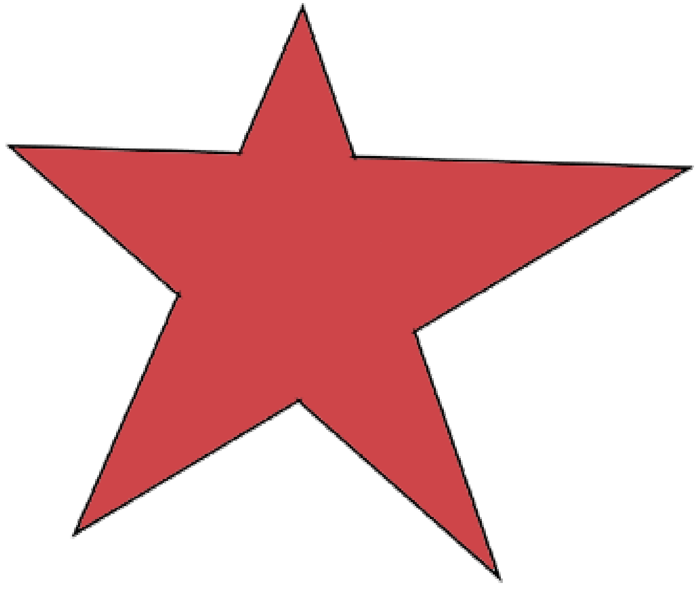
What's the Inside of a Polygon?

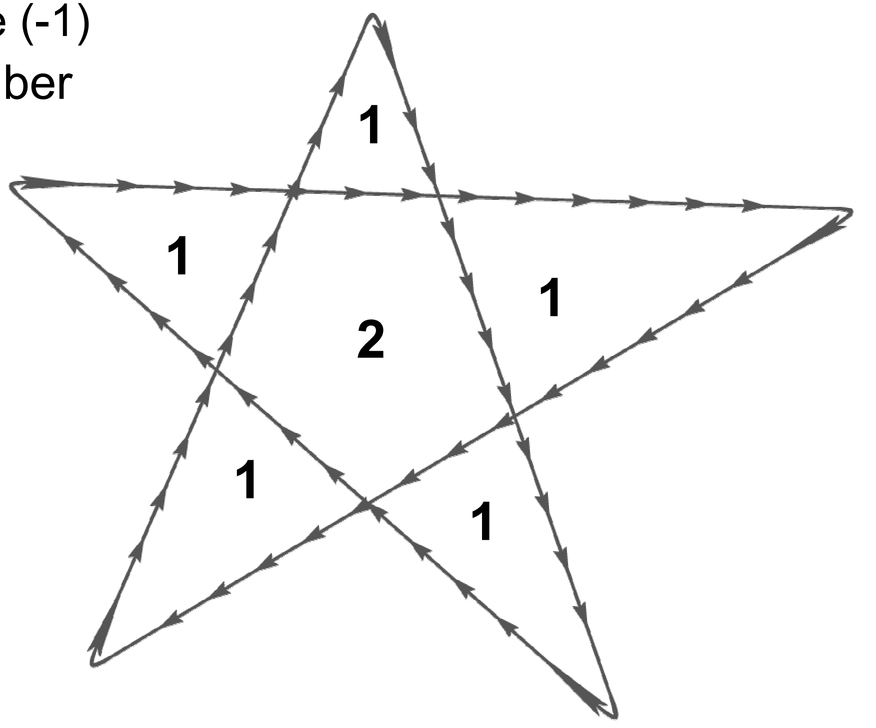
Definition #1: Odd–even rule

- pass horizontal line through shape; points with odd # crossings are in

Definition #2: Winding rule

- walk around entire polygon; add up # of times you encircle a point
  - clockwise (+1) or counter-clockwise (-1)
- fill points with non-zero winding number

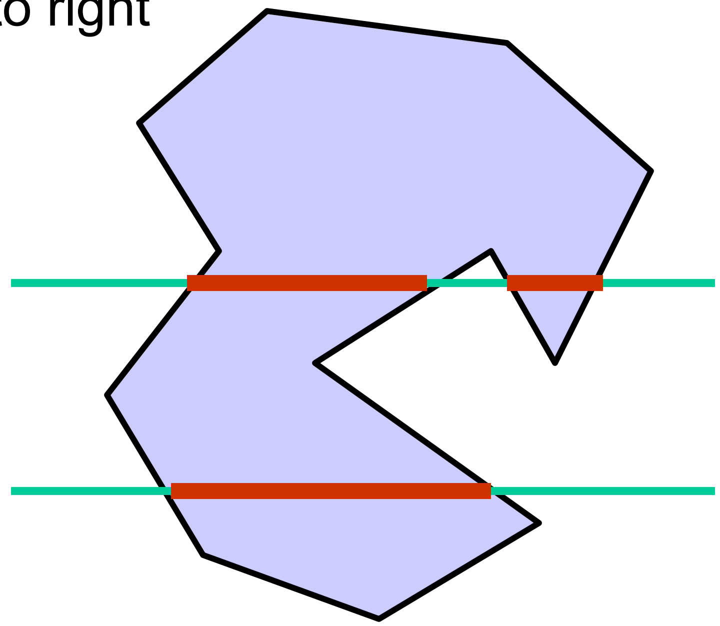




Scan Converting Polygons

Loop over all scanlines covered by polygon

- find points of intersection, from left to right
- fill all the interior spans
  - these are the odd spans
  - as per the odd–even rule



Some special cases to watch out for

- horizontal edges
- grazing vertices

Efficiently Tracking Scanline Intersections

We could do something simple, but inefficient

- directly compute intersection of every scanline with every edge

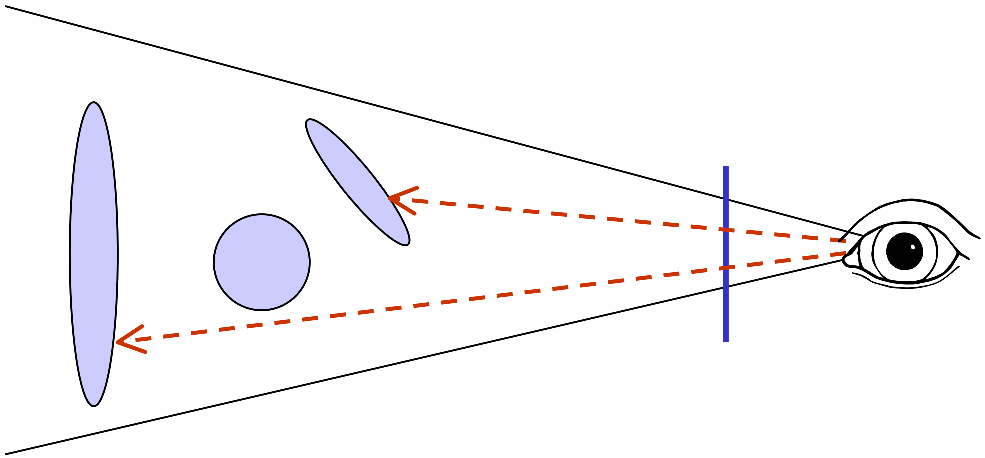
But we can do better by exploiting coherence of scanlines

- Create an Edge Table with all edges sorted by *ymin*
- Maintain Active Edge Table to hold list of edges intersecting current scanline sorted left to right

If we process the polygon from *ymin* to *ymax*

- add edge to AET at its *ymin* value
- remove edge at its *ymax* value
- when the AET is empty, we're done
- can use something like Bresenham's line algorithm to efficiently track x-coordinate of intersections

Topic #2: Visible Surface Determination



Rasterization will convert are primitives to pixels in the image

- but we need to make sure we don't draw occluded objects

For each pixel, what is the nearest object in the scene?

- this is the only thing we need to draw at this pixel
  - provided the object isn't transparent
- we need to determine the visible surface

Painter's Algorithm

Developed thousands of years ago

- probably by cave dwellers

Draws every object in depth order

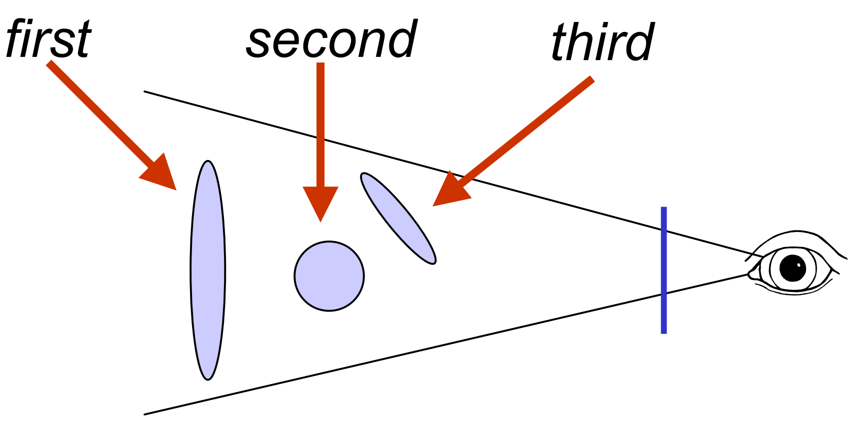
- from back to front
- near objects overwrite far objects

What could be simpler?

Painter's Algorithm:

```
sort objects back to front

loop over objects
  rasterize current object
  write pixels
```

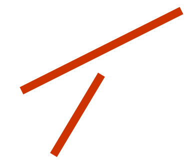


But the Catch is in the Depth Sorting

What do we sort by?

- minimum z value — no

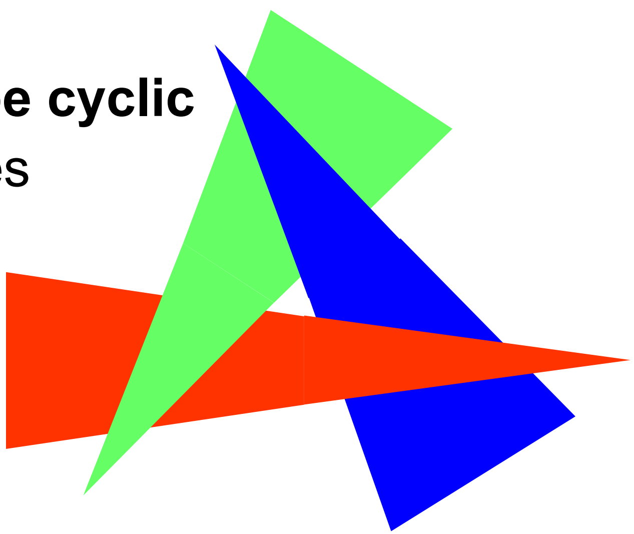
maximum z value — no



- in fact, there’s no single z value we can sort by

Worse yet, depth ordering of objects can be cyclic

- may need to split polygons to break cycles



Looking at Painter's Algorithm

It has some nice strengths

- the principle is very simple
- handles transparent objects nicely
  – just composite new pixels with what's already there

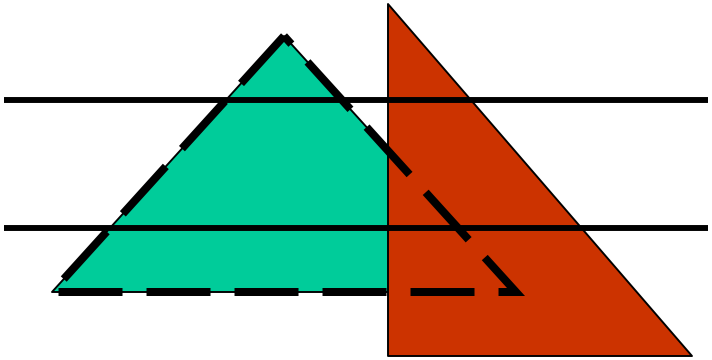
But it also has some noticeable weaknesses

- general sorting is a little expensive — worse than $O(n)$
- need to do splitting for depth cycles, interpenetration, …
- and what if the objects aren't planar polygons?

Scanline Visibility

Looks a lot like polygon rasterization

- maintains active object table
- looks at one scanline at a time — no need to store entire image
  - nice if memory is scarce



Scanline Algorithm:

```
sort objects by ymin

loop over scanlines
  update active object list
  sort active objects by x
  loop over x values
    find closest active object
    write pixel
```

The Z-Buffer Algorithm

Create new frame buffer channel

- a depth component
- to go with our RGBα channels

Records depth of pixel contents

- overwrite pixel it's farther away

This used to look pretty wasteful

- say 24 bits * number of pixels
- doubles size of framebuffer
- but memory is cheap now

Now most common method

- especially for hardware design

Z-Buffer Algorithm:

```
allocate z-buffer
initialize values to infinity

loop over all objects
  rasterize current object
  for each covered pixel (x,y)
    if z(x,y) < zbuffer(x,y)
      zbuffer(x,y) = z(x,y)
      write pixel
```

OpenGL — glEnable(GL_DEPTH_TEST)

Looking at the Z-Buffer Algorithm

It has some attractive strengths

- it's very simple, and easy to implement in hardware
- can easily accommodate any primitive you can rasterize
  - not just planar polygons

But it does have a few problems

- it doesn't handle transparency well
- needs intelligent selection of *znear* & *zfar* clipping planes
  - z-buffers typically use integer depth values
  - fixed bit precision mapped to range *znear..zfar*

Making Z-Buffers Efficient

When we rasterize a polygon, we need z value at each pixel

- we could just compute it at every pixel
- but this is pretty expensive

Can use the same incrementalization trick as in rasterization

- the projected polygon satisfies some plane equation

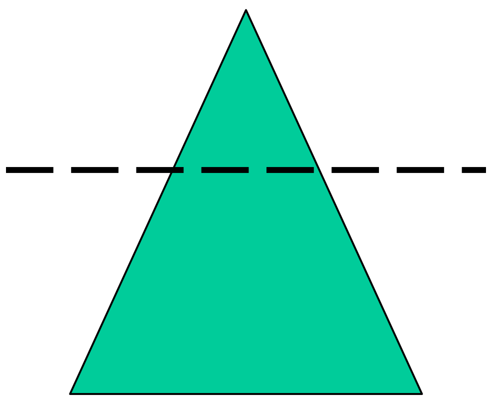
$$ax + by + cz + d = 0$$

- we could compute the depth as

$$z = \frac{-d - ax - by}{c}$$

- but taking account of coherence

$$\Delta z = -\frac{a}{c}\Delta x \quad \text{for fixed values of } y$$



Ray Casting

This is a very general algorithm

- works with any primitive we can write intersection tests for
- but it's hard to make it run fast

We'll come back to this idea later

- can use it for much more than visibility testing
- shadows, refractive objects, reflections, motion blur, …

Ray Casting:

```
loop over every pixel (x,y)
  shoot ray from eye through (x,y)
  intersect with all surfaces
  find first intersection point
  write pixel
```