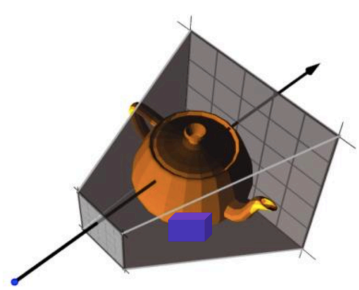
Topic 8
Visibility

- Intro visibility
- Object vs Image space algorithms
- Bounding volumes
- Culling
- Clipping
- Z-buffering
- Scan conversion

Visibility

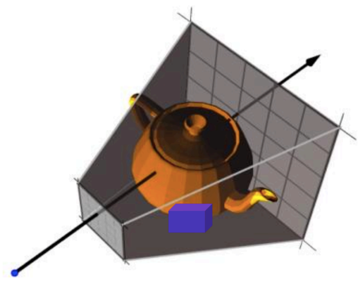
What is not visible?



Visibility

What is not visible?

- Primitives outside of the field of view
- Back-facing primitives
- Primitives occluded by other objects closer to the camera

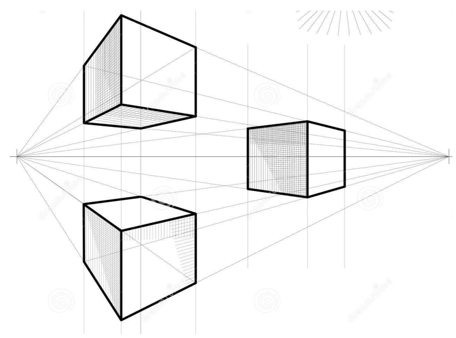


Why compute visibility?

- General principle: don't spend cycles drawing what you don't have to!
- Some things are not visible. Can we get rid of these?
- Efficiency: If it won't contribute to the final image, avoid unnecessary computations.
- Realism: Objects occlusions naturally happen in scenes

Why compute visibility

- Example cube in perspective:
- At most three faces will be visible
- Three sides don't even need to be drawn



(Google images)

Why compute visibility

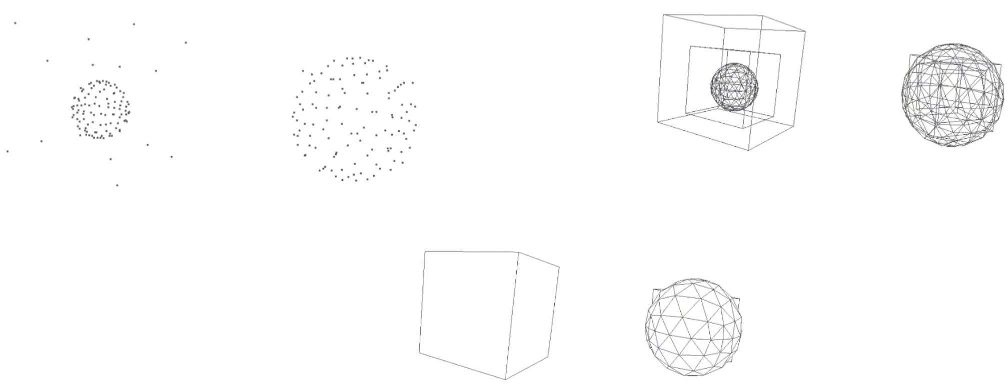


Image source: http://www.cs.sfu.ca/~torsten/

Why compute visibility

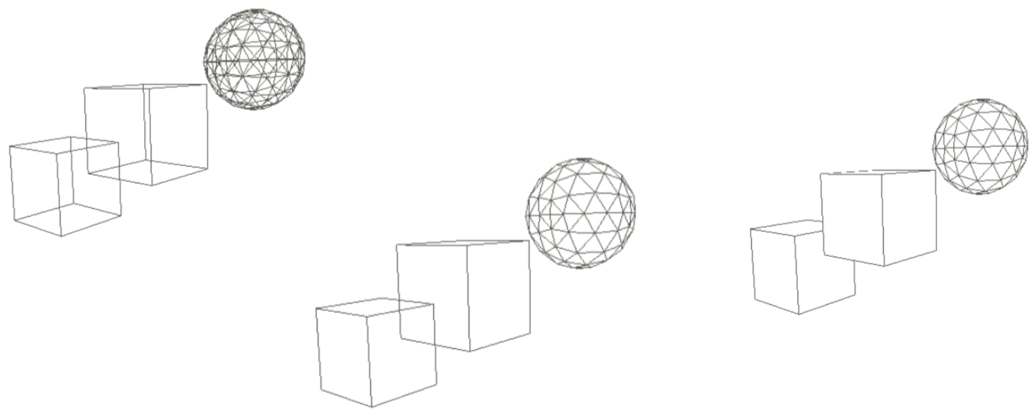


Image source: http://www.cs.sfu.ca/~torsten/

Types of algorithms

- Object space
  - Occurs at the polygon level in object space
  - Do the work on the objects themselves *before* they are converted to pixels
  - Done at the mathematical/analytical level independent of resolution

- Images space
  - Occurs at the pixel level in image space
  - Work done when objects are being converted to pixels
  - Resolution of the display matters here
  - Determine the colour of pixel based on what is visible

Object Space

```
for each object in scene {
    determine which parts of objects are
    visible (parts unobstructed by itself or
    other objects)
    rasterize only those parts
}
```

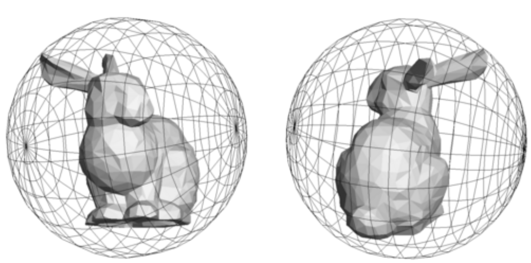
Image Space

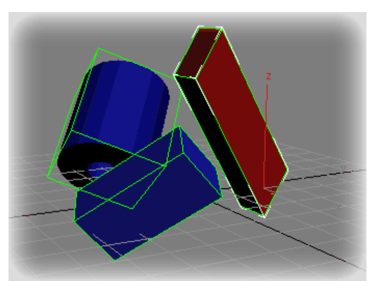
```
for each pixel in image {
    determine polygon closest to the viewer at
    that pixel location
    colour the pixel with the appropriate colour
}
```

Efficiency

- Bounding volumes (boxes, spheres)
- Back-face culling
- Coherence (exploit local similarity)

Bounding Volumes





Source: Google images

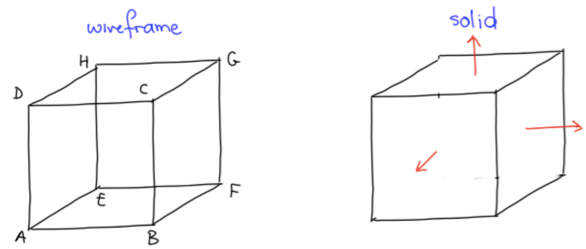
Culling

- Frustum culling
  - Culling of triangles outside of the view frustum
- Occlusion culling
  - Culling of triangles within the frustum that are occluded by others
- Backface culling
  - Culling of triangles facing away from the camera

Backface Culling

- Goal: Remove surfaces that point *away* from the camera (i.e. the “backfacing” polygons). Don’t draw these!
- For (most) solid objects, back faces should not be visible → hidden
- Can be done in window coordinates (winding order) or in world coordinates using face normals.

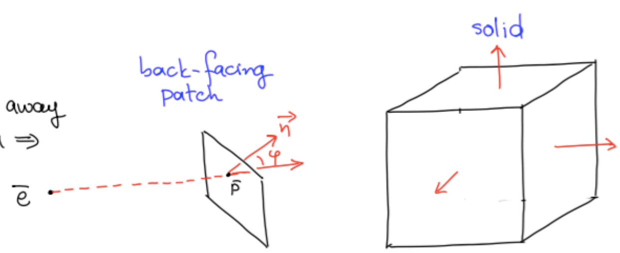
Back-face Culling



Backfacing faces on the wireframe model:
ADHE, EHGF, AEFB will not be drawn.

Back-face Culling

if $\varphi < 90^\circ \Rightarrow$
patch faces away
from camera $\Rightarrow$
CULL



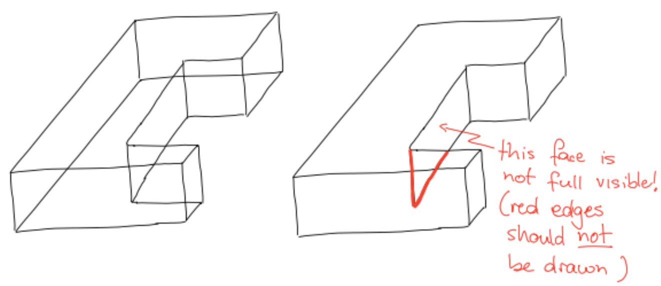
Culling criterion:

$(\bar{p}-\bar{e})\cdot\vec{n} > 0 \Rightarrow$ CULL

$(\bar{p}-\bar{e})\cdot\vec{n} \leq 0 \Rightarrow$ DO NOT CULL (may be visible)

Back-face Culling

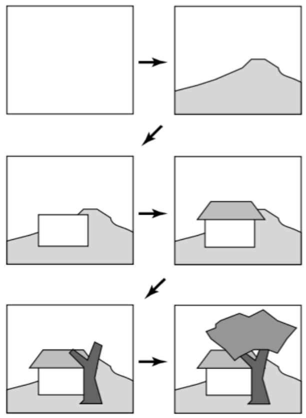
- Some limitations:



Visibility Algorithms

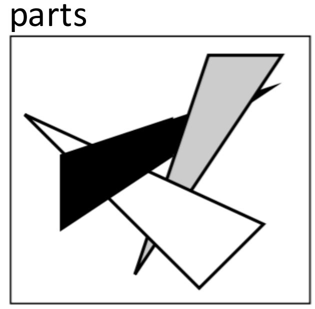
- Object-space algorithms:
  - Painter's Algorithm
  - Binary Space Partitioning algorithm (BSP) (next week)
- Image-space algorithms:
  - Z-buffer
  - Scanline

Painter's algorithm



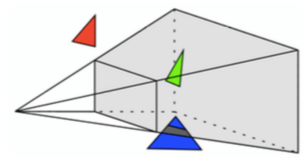
Painter's algorithm

- Limitations: Cycles
- There is *no* sort order here to allow correct visibility handling
- Workaround: break polygons into smaller parts



Clipping

In general, we cannot expect all triangles to lie within the view frustum. Triangles that lie partly outside the view frustum must be clipped.



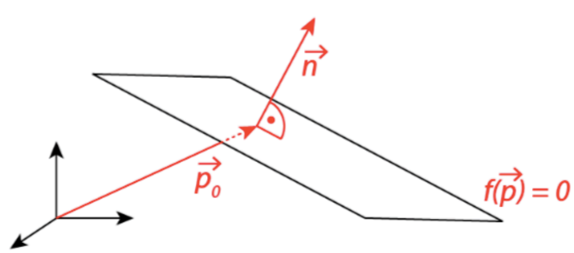
We must ...

1. verify if a triangle intersects with a hyperplane
2. create new triangle(s)

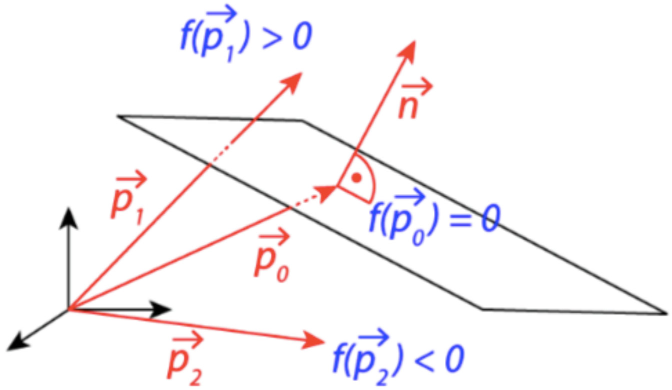
Clipping

Remember the implicit equation of a plane:

$$f(\vec{p}) = \vec{n}(\vec{p} - \vec{p_0}) = 0 \quad \text{or} \quad f(\vec{p}) = \vec{n} \cdot \vec{p} + d = 0$$



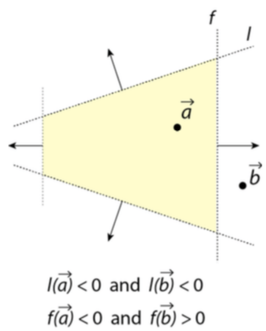
Clipping



Clipping

Hence, if the normals of the clipping planes point outward:

- $\vec{p}$ is "inside the plane" if $f(\vec{p}) < 0$
- $\vec{p}$ is "outside the plane" if $f(\vec{p}) > 0$



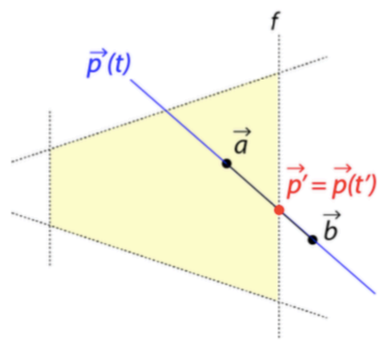
Clipping

If two points $\vec{a}$ and $\vec{b}$ are on different sides of a hyperplane, we first determine the parametric equation of the line through the points:

$$\vec{p}(t) = \vec{a} + t(\vec{b} - \vec{a})$$

Substituting this into the hyperplane equation yields

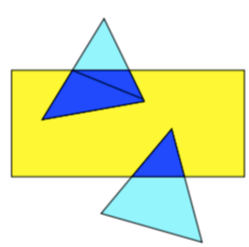
$$t' = \frac{\vec{n} \cdot \vec{a} + d}{\vec{n} \cdot (\vec{a} - \vec{b})}$$



Clipping

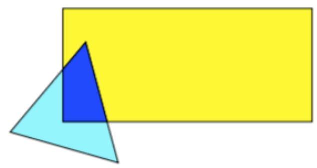
Given the intersection points, we can clip the triangle against the hyperplane as follows:

- If two vertices are on the positive side, we get one new triangle.
- If one vertex is on the positive side of the hyperplane, we get two new triangles.



Clipping

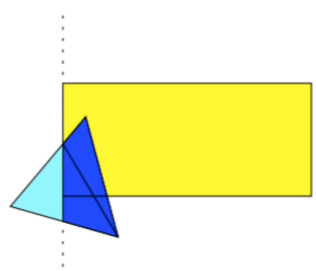
But what do we do if a triangle is intersected by two clipping planes?



Clipping

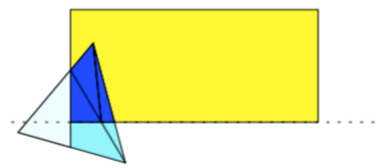
We have to deal with such situations one clipping plane at a time.

We first clip the initial triangle against one of the clipping planes. . .



Clipping

. . . and clip the remaining triangle(s) against the other clipping plane.

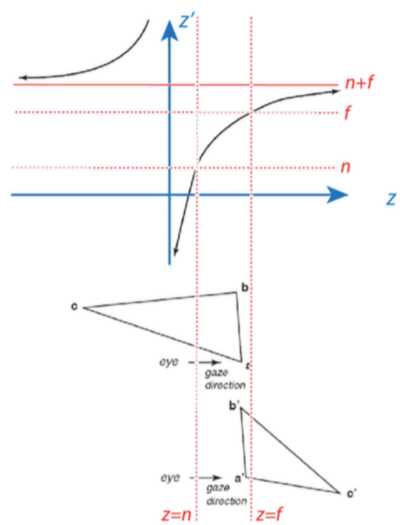


When to do Clipping

$$z' = n + f - \frac{fn}{z}$$

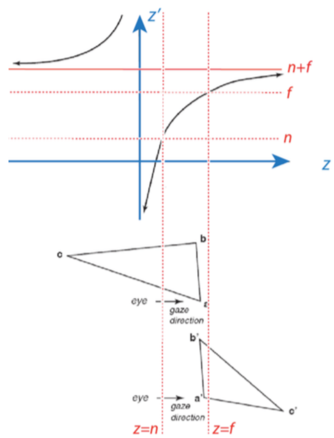
Because of the discontinuity, objects behind the eye can move in front of it.

Because of the change of signs at $f(z) = 0$, objects in front of the eye can move behind it.



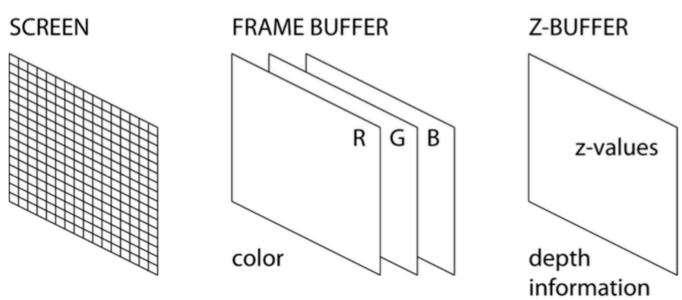
When to do Clipping

- Easiest to clip *before* the homogenization step and clip in homogeneous coordinates
- This means we actually clip in four dimensions against three dimensional clipping hyperplanes.
- After homogenization, the result gives us the coordinates in 3D space.



Z-buffering

Apart from the frame buffer, which contains the pixels of the image, also maintain a Z-buffer of the same width and height, to store depth information for the projected triangles.



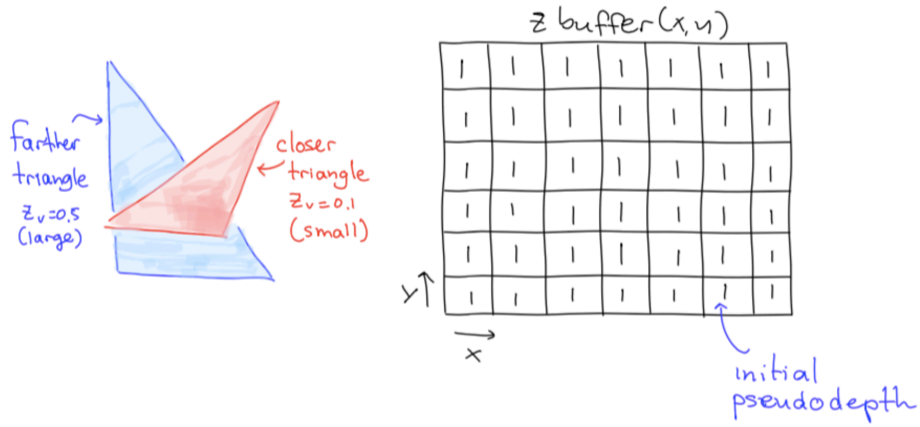
Z-buffering

- An image-space algorithm.
- Maintains depth for each pixel (really the pseudo-depth)
- Initially set to “very far away”
- Checks the depth of a colour before colouring the pixel.
- If colour is “closer” then, colour pixel with it and update the z-buffer.
- Else, keep everything as is.

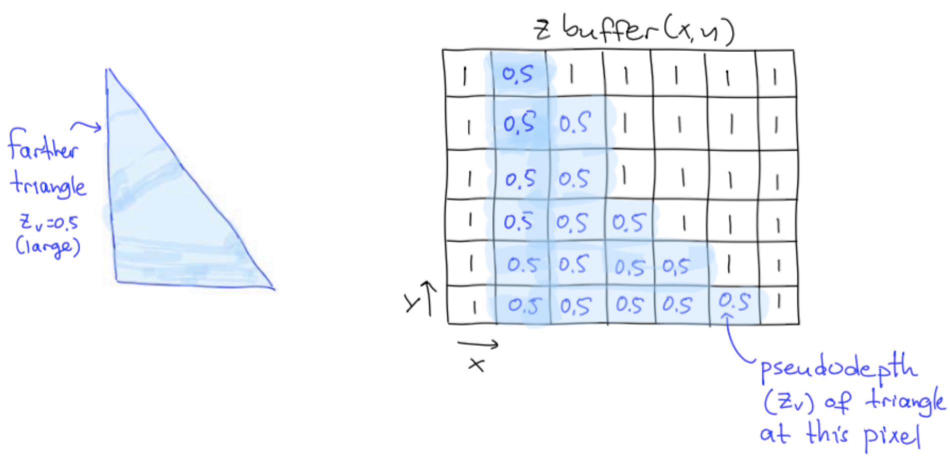
Z-buffering

```
for each polygon
    for each pixel p in the polygon's projection
     {
     pz = pseudo-depth at (x, y);
     if (pz > zBuffer[x, y]) // closer to the camera
         {
         zBuffer[x, y] = pz;
         framebuffer[x, y] = colour of pixel p
         }
     }
```

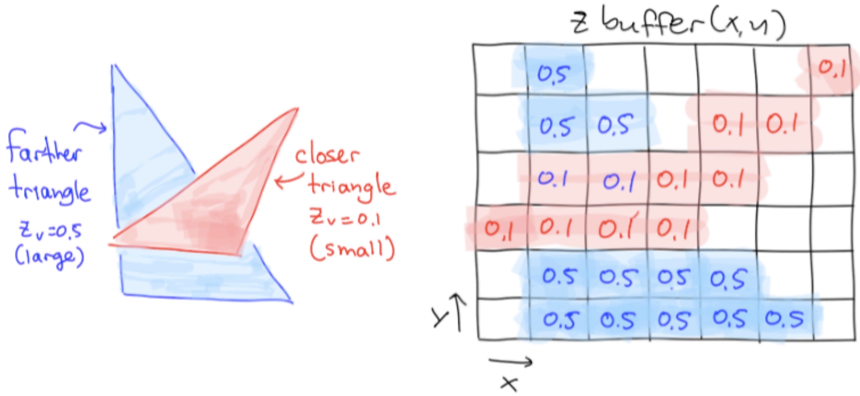
Z-buffering



Z-buffering

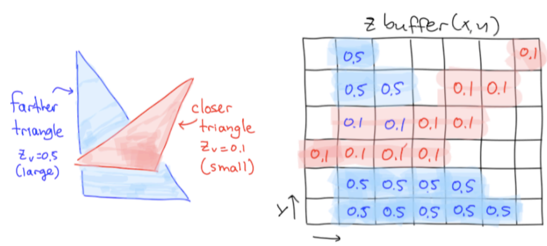


Z-buffering



Z-buffering

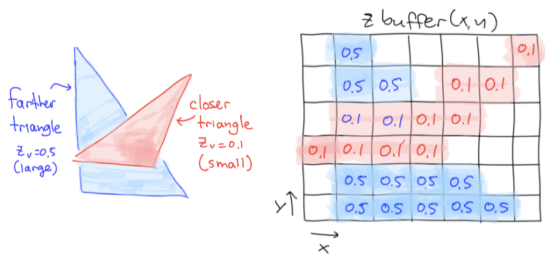
Q: What would the result be if we drew the closest triangle first?



Z-buffering

Q: What would the result be if we drew the closest triangle first?

A: The result would be the *same*



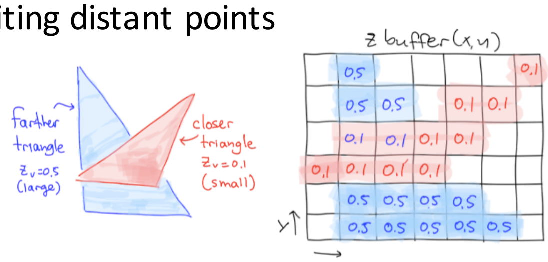
Z-buffering

Advantages:

- Simple and accurate
- Independent of the order polygons are drawn

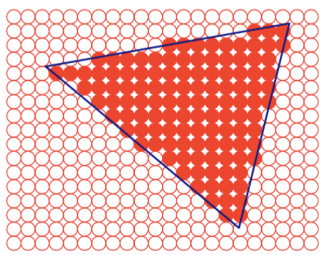
Disadvantages:

- Biggest issue with z-buffering is finite precision (z-fighting)
- Wasted computation when overwriting distant points



Rasterization or scan conversion

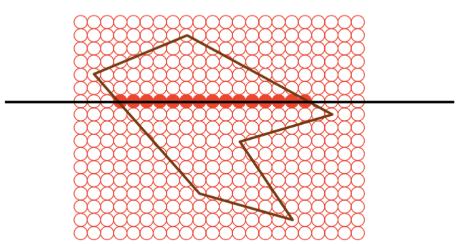
Rasterization takes shapes like triangles and determines which pixels to fill.



Filling polygons

First approach:

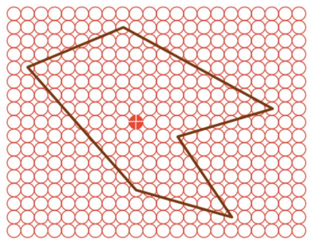
1. Polygon Scan-Conversion
   - Rasterize a polygon scan line by scan line, determining which pixels to fill on each line.



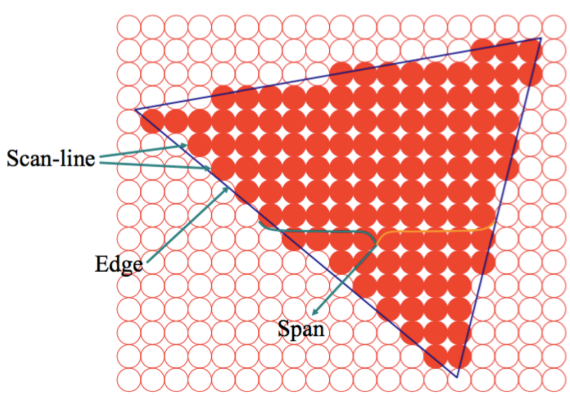
Filling polygons

Second Approach:

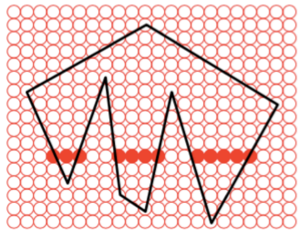
2. Polygon Fill
   - Select a pixel inside the polygon. Grow outward until the whole polygon is filled.



Coherence



Polygon scan conversion

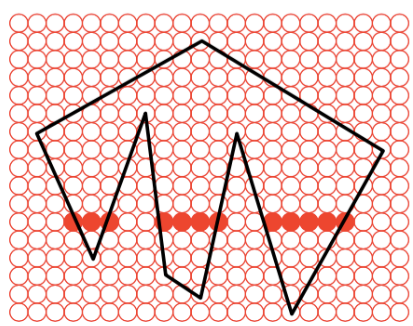


- Intersection Points
- Other points in the span

Polygon scan conversion

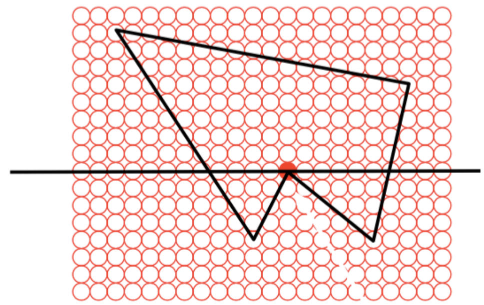
Process each scan line

1. Find the intersections of the scan line with all polygon edges.
2. Sort the intersections by x coordinate.
3. Fill in pixels between pairs of intersections using an odd-parity rule.
   - Set parity even initially.
   - Each intersection flips the parity.
   - Draw when parity is odd.



Special cases: vertices

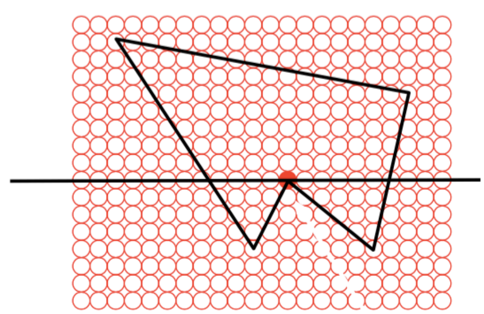
- Q: How do we count the intersecting vertex in the parity computation?



Special cases: vertices

- Q: How do we count the intersecting vertex in the parity computation?

A: Count it zero or two times.

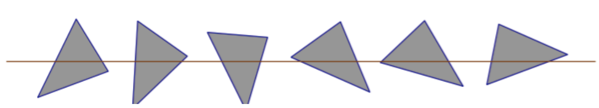


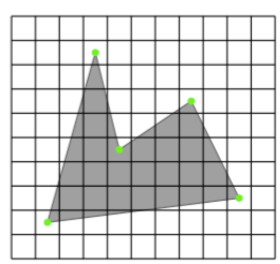
Computing intersections

- For each scan line, we need to know if it intersects the polygon edges.
- It is expensive to compute a complete line-line intersection computation for each scan line.
- After computing the intersection between a scan line and an edge, we can use that information in the next scan line.
- We can exploit scanline coherence.

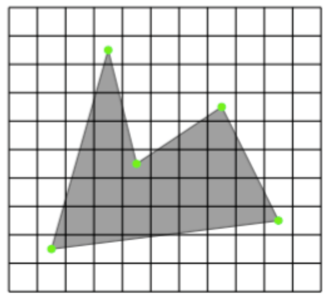
Rasterizing general polygons

- Let's look at a more general method for polygons.
- For triangles, it becomes even more efficient (only one span per scanline since it is always convex)

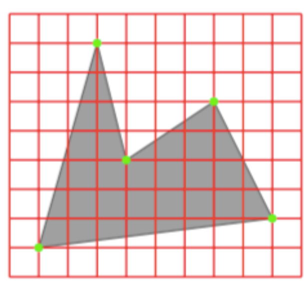




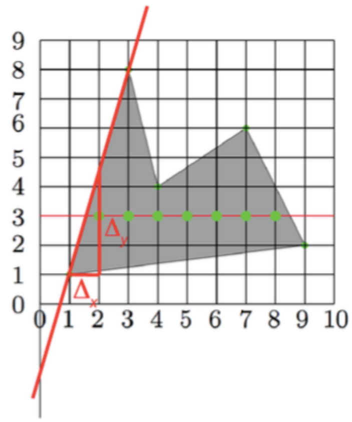
Rasterizing general polygons



Rasterizing general polygons



Rasterizing general polygons



Rasterizing general polygons

The left active edge runs from $(1,1)$ to $(3,8)$.

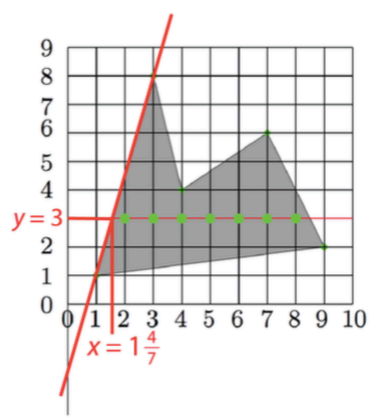
The slope intercept equation, i.e.

$$y = \frac{y_j - y_i}{x_j - x_i}x + d \text{ with } j > i$$

for the line through these points is

$$y = 3.5x - 2.5$$

The x-coordinate of the intersection of this edge with scanline 3 is $1\frac{4}{7}$, so we know that we have to set pixels starting from $x = 2$.

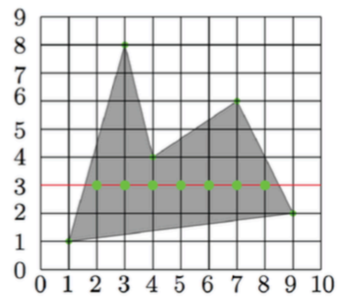


Rasterizing general polygons

Computing intersections of scanlines and edges requires a division. These are expensive, and we'd like to avoid them.

We use the fact that the intersection of an edge with scanline i is related to the intersection with scanline $i-1$.

This is called vertical coherence.



Rasterizing general polygons

The slope of our left active edge is

$$\frac{y_j - y_i}{x_j - x_i} = \frac{\Delta_y}{\Delta_x} = \frac{7}{2} \text{ (with } j < i\text{)}$$

Hence, the increase in x-coordinate from one scanline to the next is

$$\Delta_x = \frac{2}{7}$$