8. From Geometry to Pixels

Outline

- Rendering Overview
- Clipping
- Polygon Rendering
- Rasterization
- Display Issues

Rendering Overview

Objectives

- Examine what happens between the vertex shader and the fragment shader
- Introduce basic implementation strategies
- Clipping
- Rendering
  - lines
  - polygons
- Give a sample algorithm for each

Overview

- At end of the geometric pipeline, vertices have been assembled into primitives
- Must clip out primitives that are outside the view frustum
  - Algorithms based on representing primitives by lists of vertices
- Must find which pixels can be affected by each primitive
  - Fragment generation
  - Rasterization or scan conversion

Required Tasks

- Clipping
- Rasterization or scan conversion
- Transformations
- Some tasks deferred until fragment processing
  - Hidden surface removal
  - Antialiasing

Rasterization Meta Algorithms

- Any rendering method process every object and must assign a color to every pixel
- Think of rendering algorithms as two loops
  - over objects
  - over pixels
- The order of these loops defines two strategies
  - image oriented
  - object oriented

Object Space Approach

- **For every object**, determine which pixels it covers and shade these pixels
  - Pipeline approach
  - Must keep track of depths for HSR
  - Cannot handle most global lighting calculations
  - Need entire framebuffer available at all times

Image Space Approach

- **For every pixel**, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel
  - Ray tracing paradigm
  - Need all objects available
- Patch Renderers
  - Divide framebuffer into small patches
  - Determine which objects affect each patch
  - Used in limited power devices such as cell phones

Algorithm Experimentation

- Create a framebuffer object and use render-to-texture to create a virtual framebuffer into which you can write individual pixels

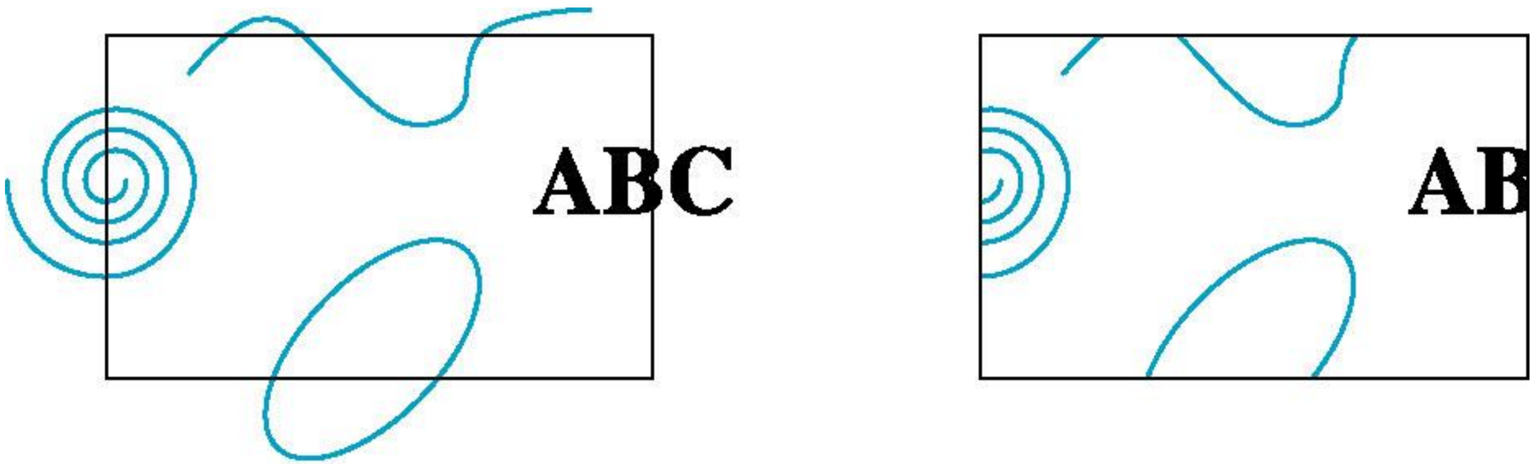
Clipping

Objectives

- Clipping lines
- First of implementation algorithms
- Clipping polygons
- Focus on pipeline plus a few classic algorithms

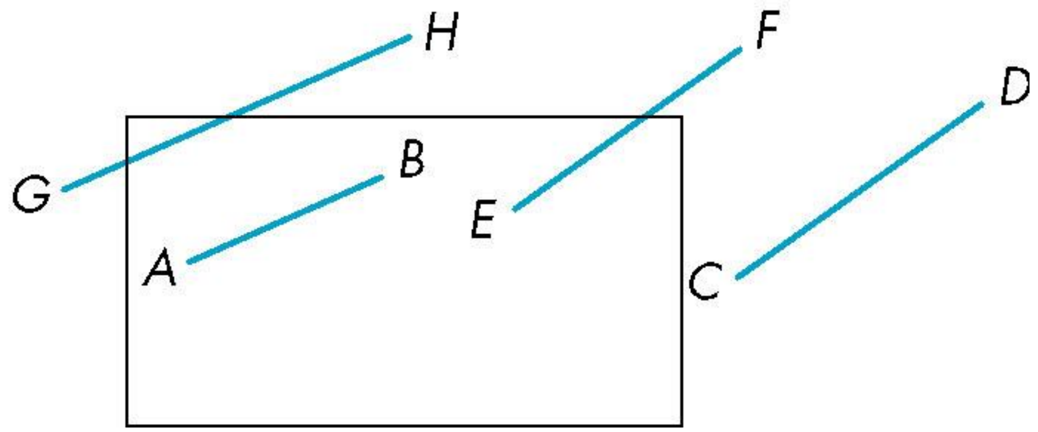
Clipping

- 2D against clipping window
- 3D against clipping volume
- Easy for line segments polygons
- Hard for curves and text
  - Convert to lines and polygons first



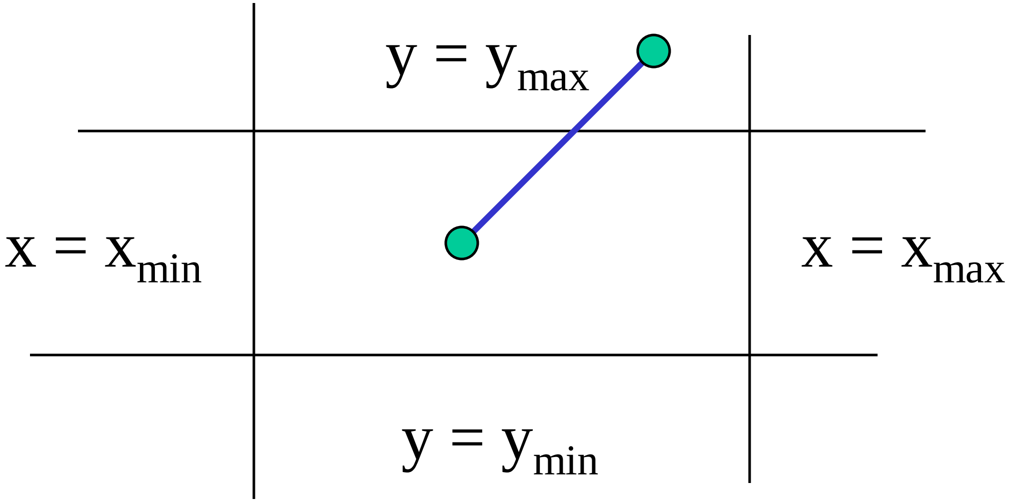
Clipping 2D Line Segments

- Brute force approach: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection



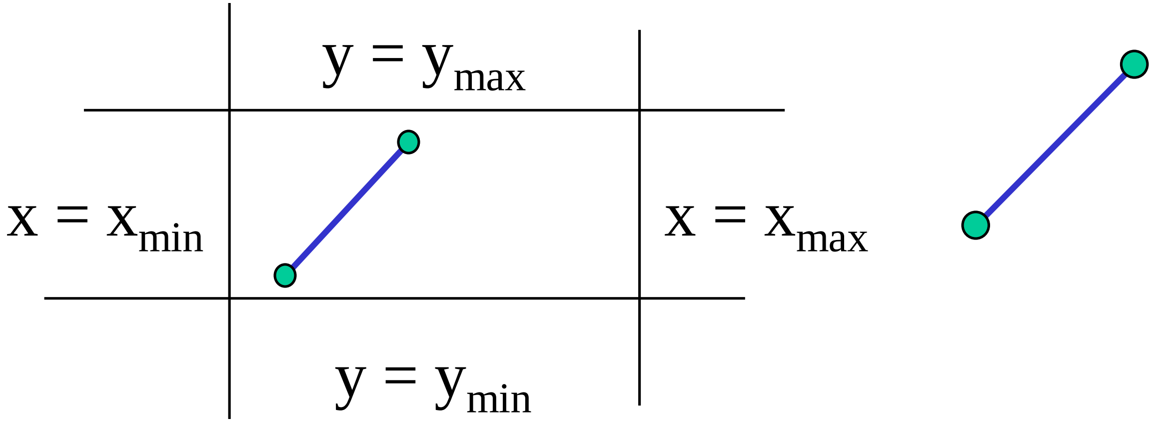
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

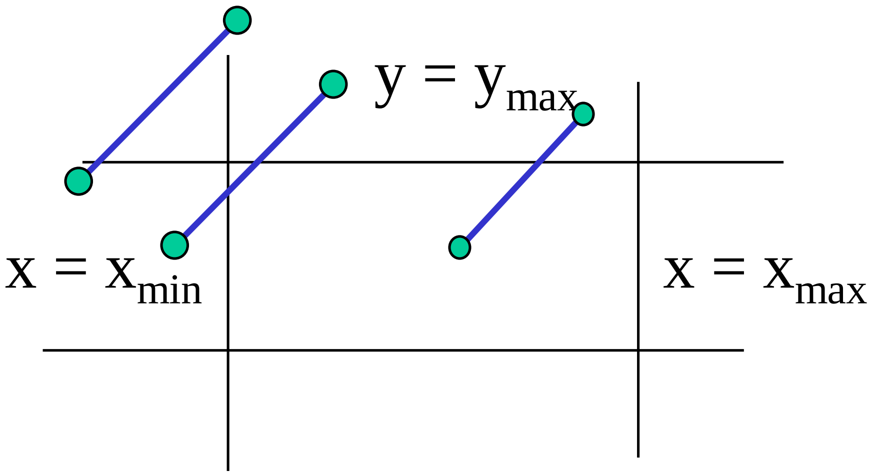
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection

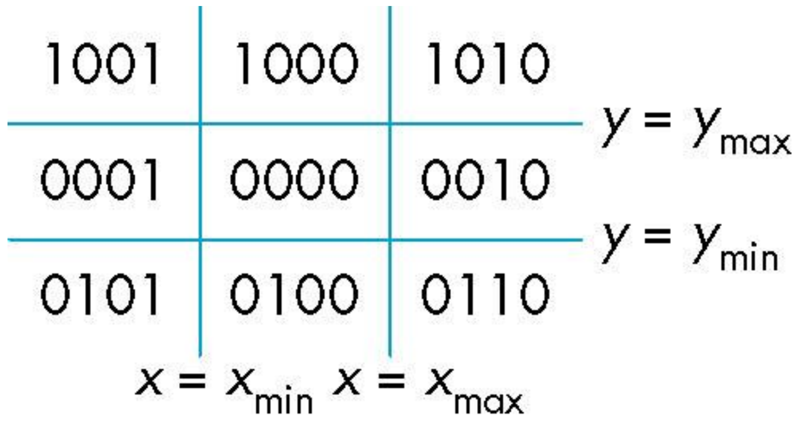


Defining Outcodes

- For each endpoint, define an outcode

$$b_0 b_1 b_2 b_3$$

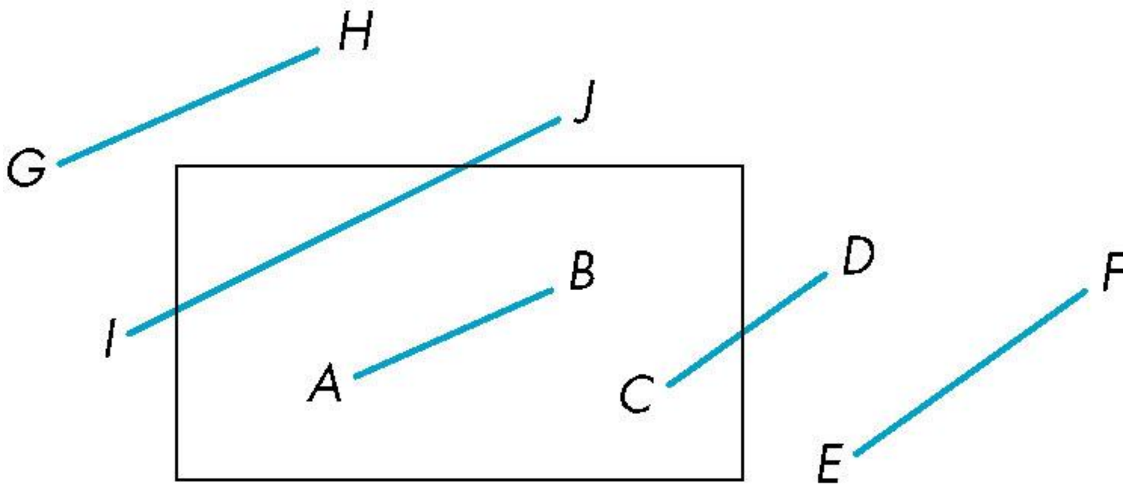
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

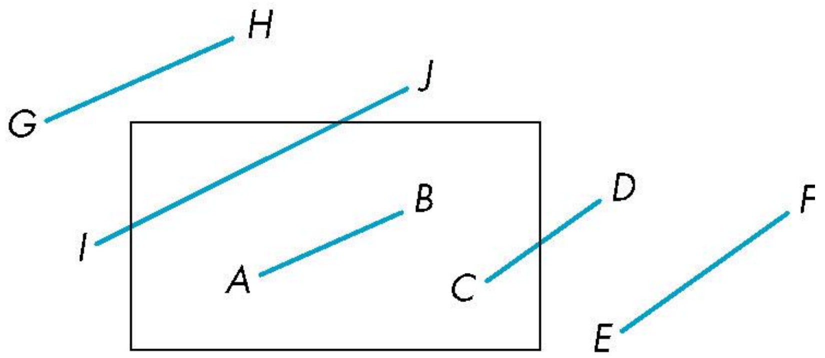
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - Accept line segment



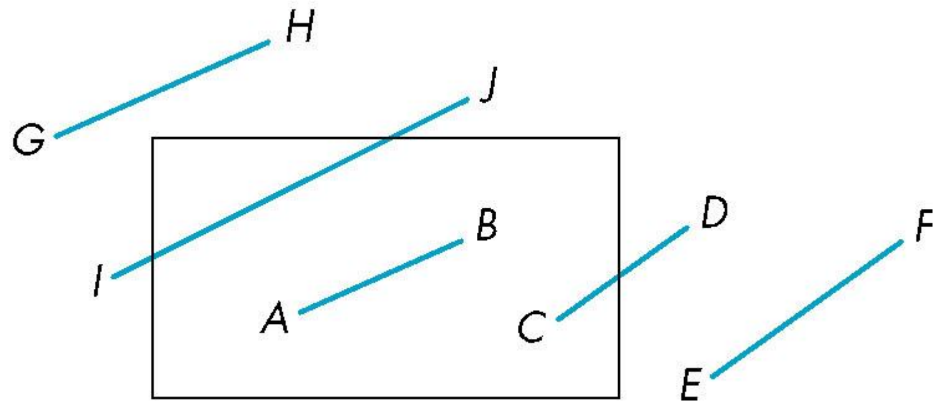
Using Outcodes

- CD: outcode (C) = 0, outcode(D) $\neq$ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interesections



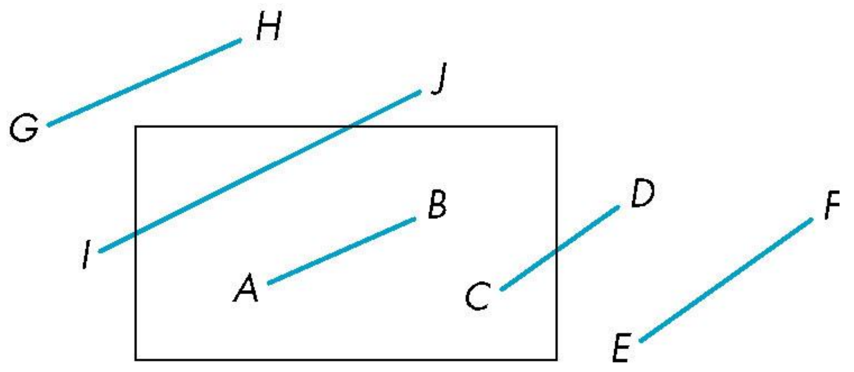
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) $\neq 0$
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - reject



Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm

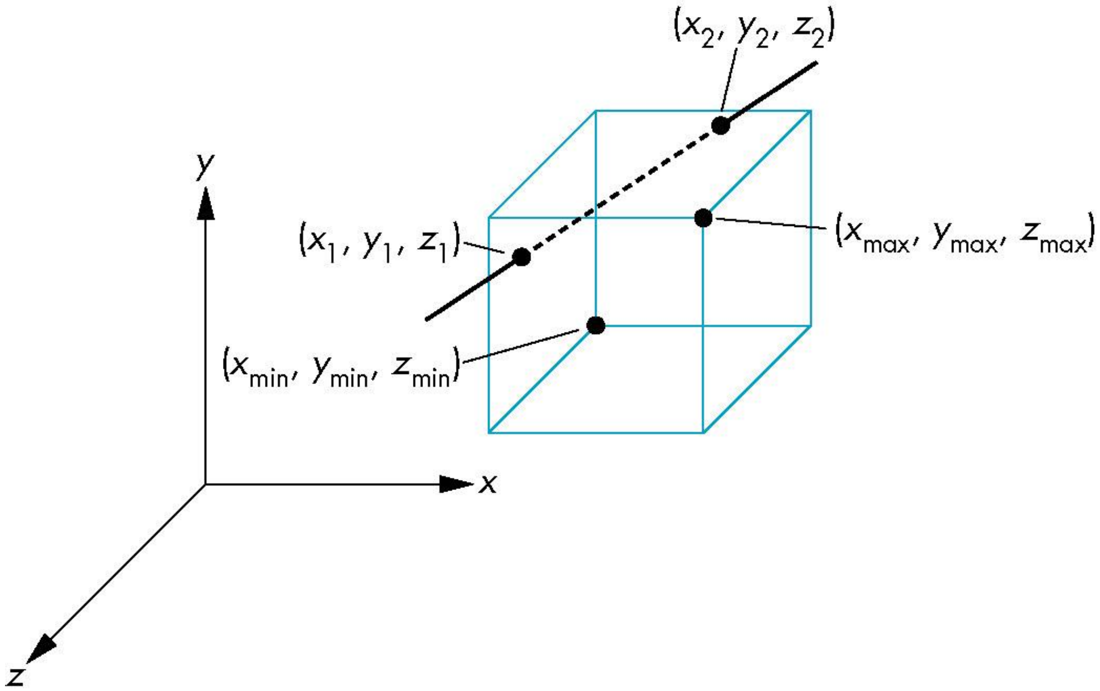


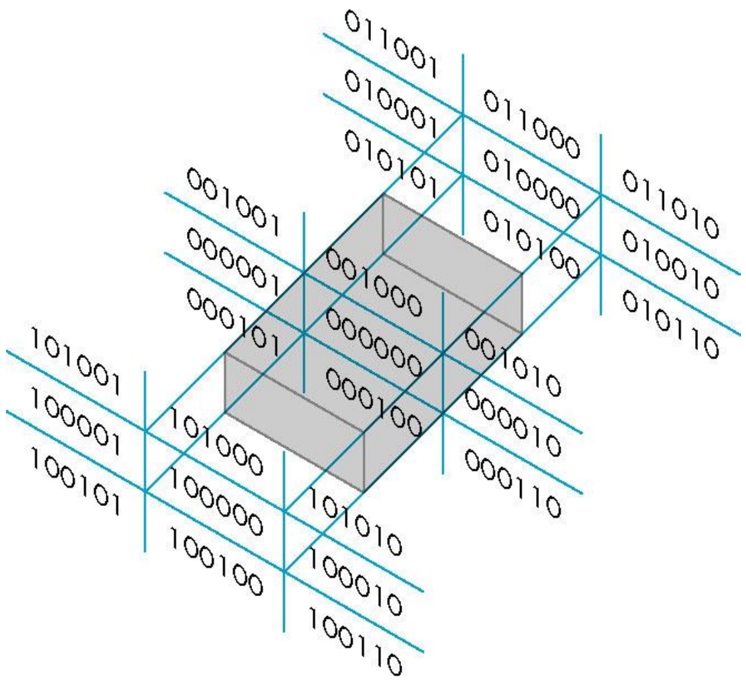
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

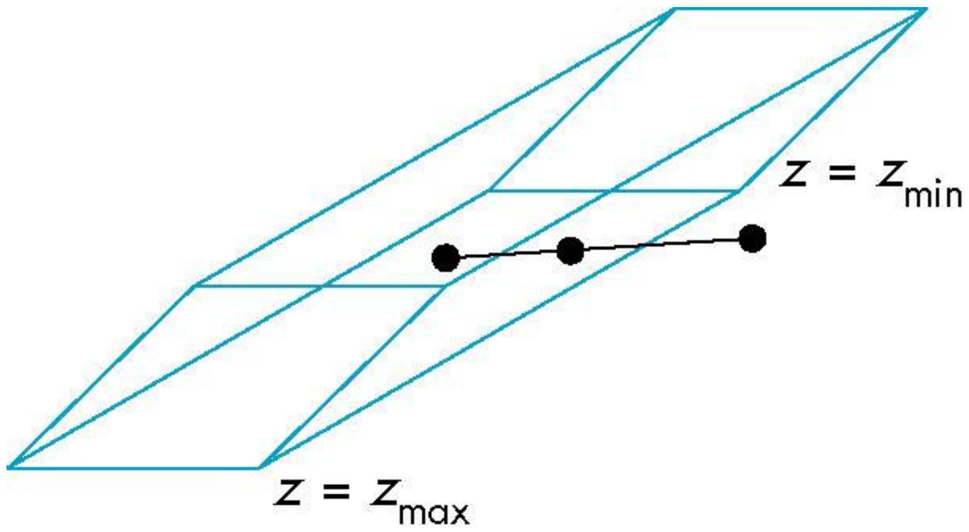
- Use 6-bit outcodes
- When needed, clip line segment against planes



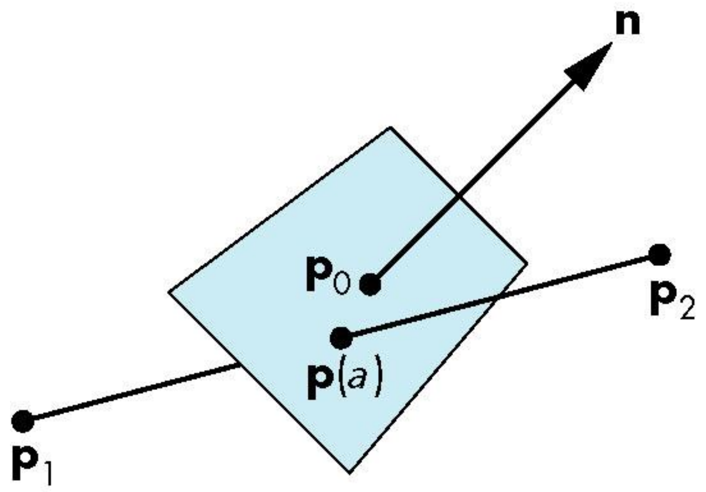


Clipping and Normalization

- General clipping in 3D requires intersection of line segments against arbitrary plane
- Example: oblique view



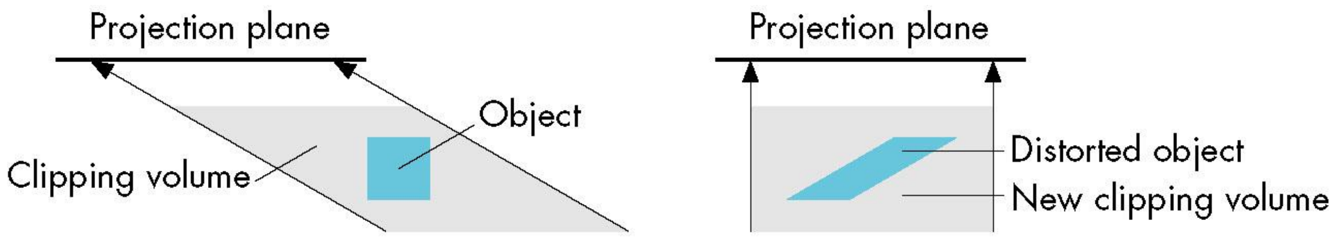
Plane-Line Intersections



$$a = \frac{n \bullet (p_o - p_1)}{n \bullet (p_2 - p_1)}$$

Normalized Form

top view



before normalization after normalization

Normalization is part of viewing (pre clipping) but after normalization, we clip against sides of right parallelepiped

Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$?

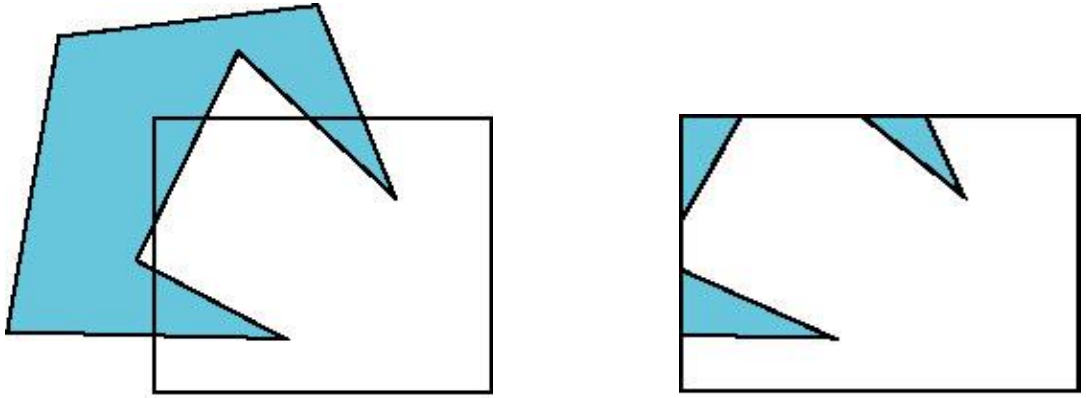
Polygon Rendering

Objectives

- Introduce clipping algorithms for polygons
- Survey hidden-surface algorithms

Polygon Clipping

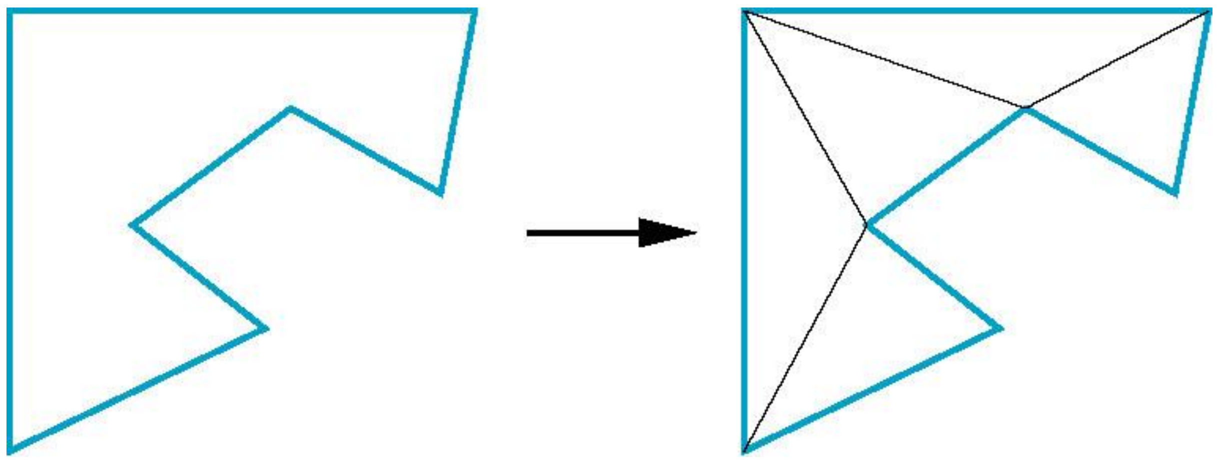
- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons



- However, clipping a convex polygon can yield at most one other polygon

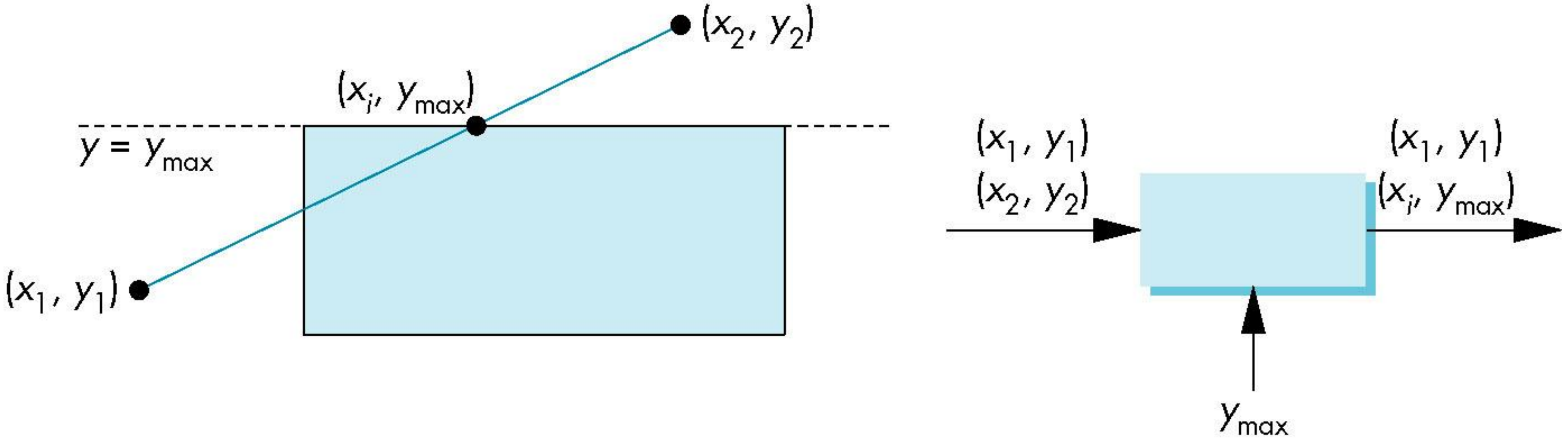
Tessellation and Convexity

- One strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier
- Tessellation through tesselllation shaders



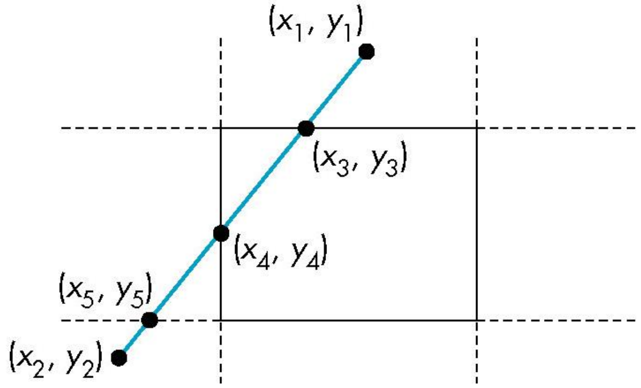
Clipping as a Black Box

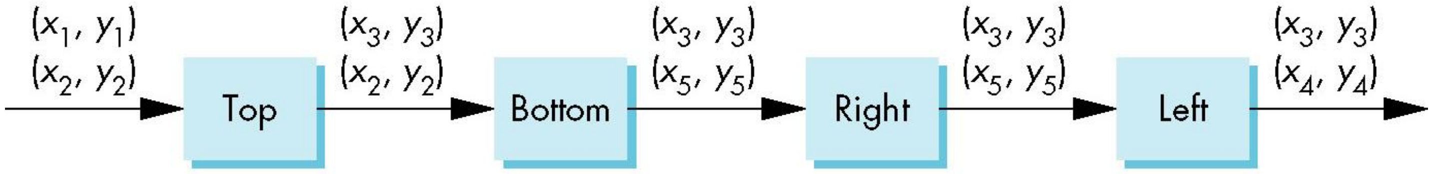
- Can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment



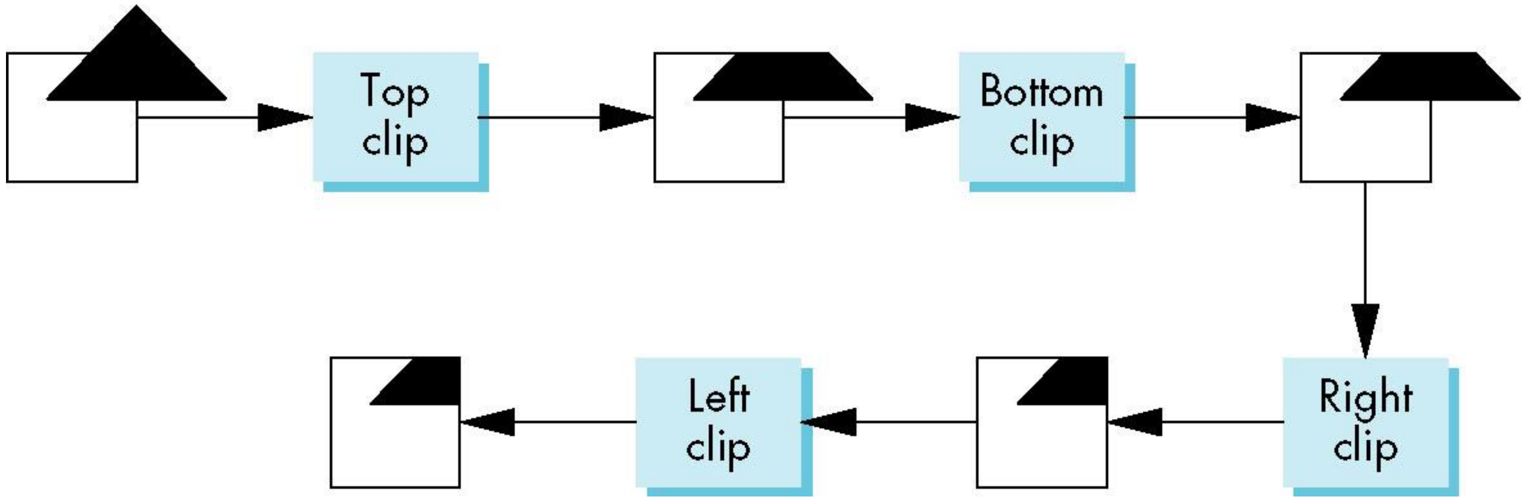
Pipeline Clipping of Line Segments

- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline





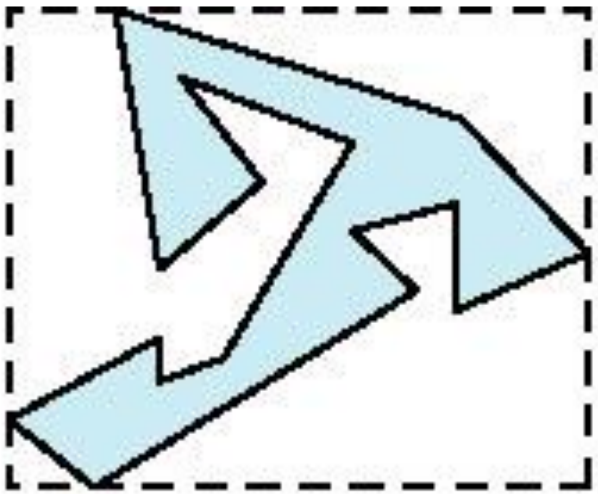
Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

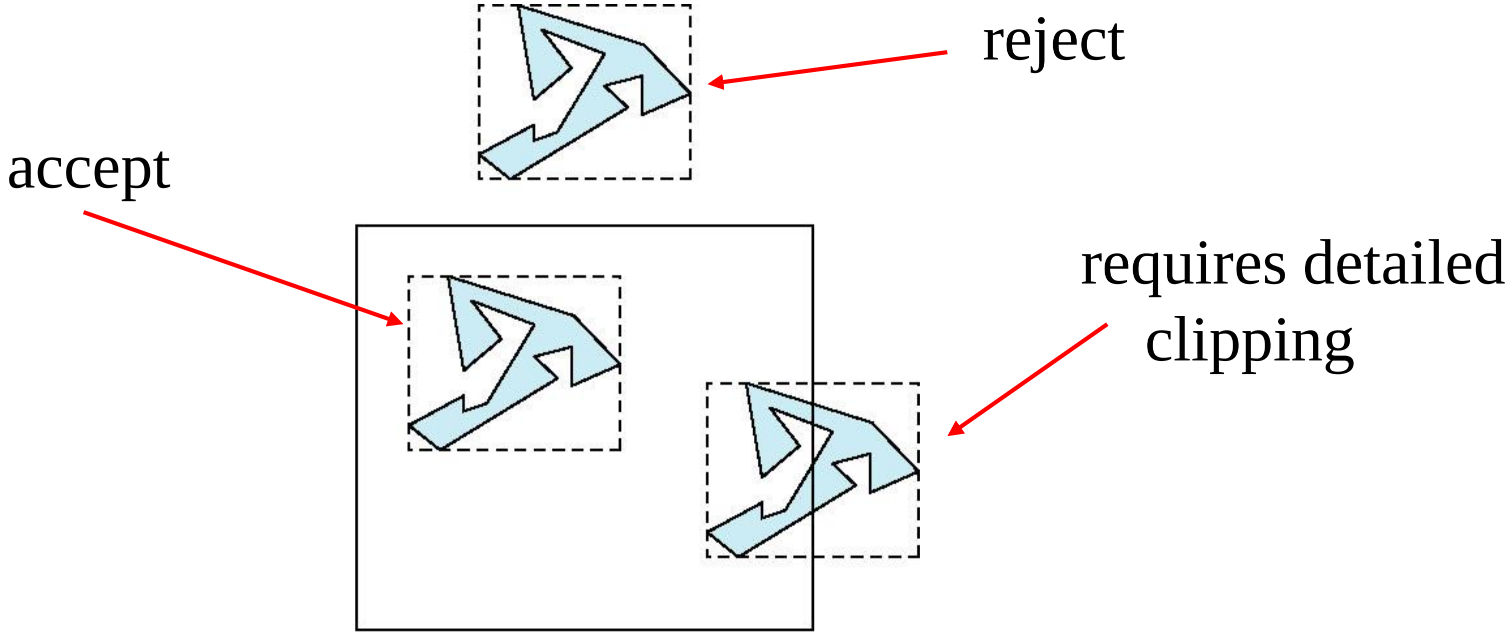
Bounding Boxes

- Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*
  - Smallest rectangle aligned with axes that encloses the polygon
  - Simple to compute: max and min of x and y



Bounding boxes

Can usually determine accept/reject based only on bounding box

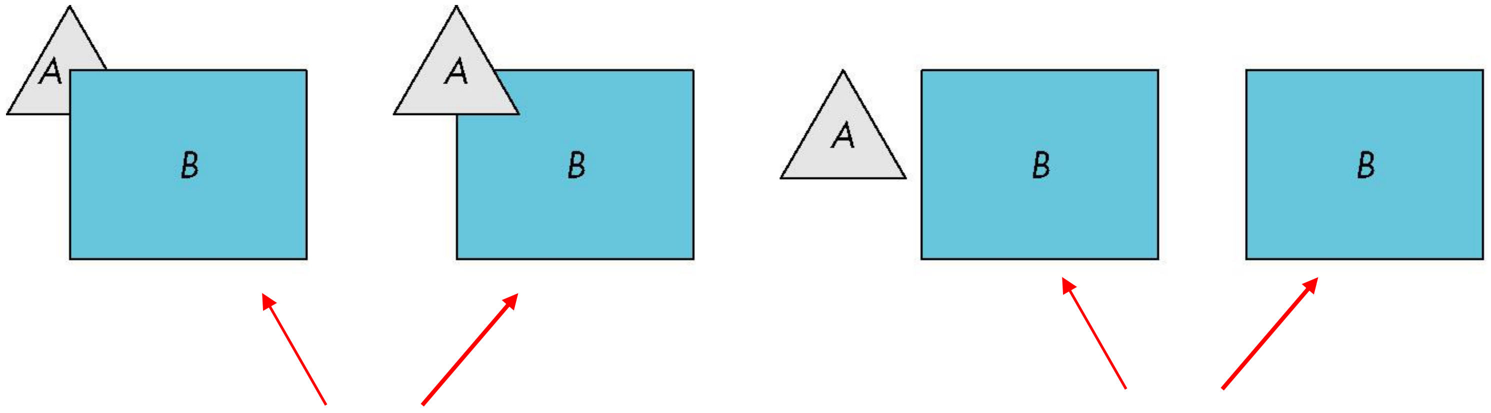


Clipping and Visibility

- Clipping has much in common with hidden-surface removal
- In both cases, we are trying to remove objects that are not visible to the camera
- Often we can use visibility or occlusion testing early in the process to eliminate as many polygons as possible before going through the entire pipeline

Hidden Surface Removal

- Object-space approach: use pairwise testing between polygons (objects)



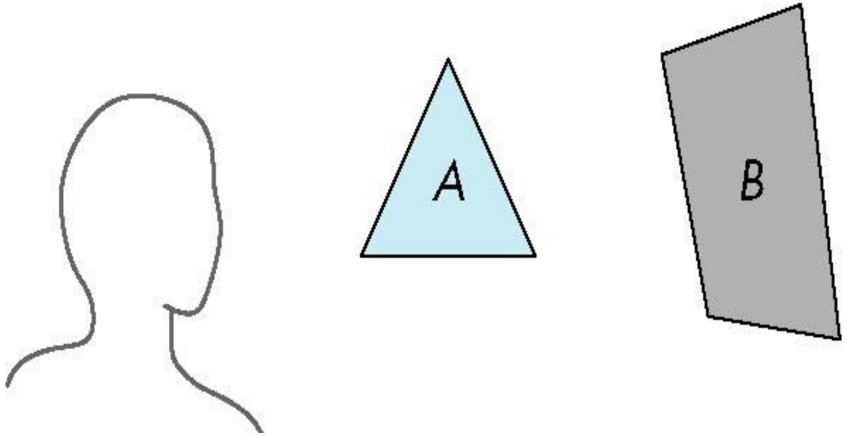
partially obscuring

can draw independently

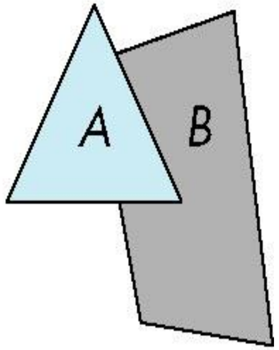
- Worst case complexity $O(n^2)$ for n polygons

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer



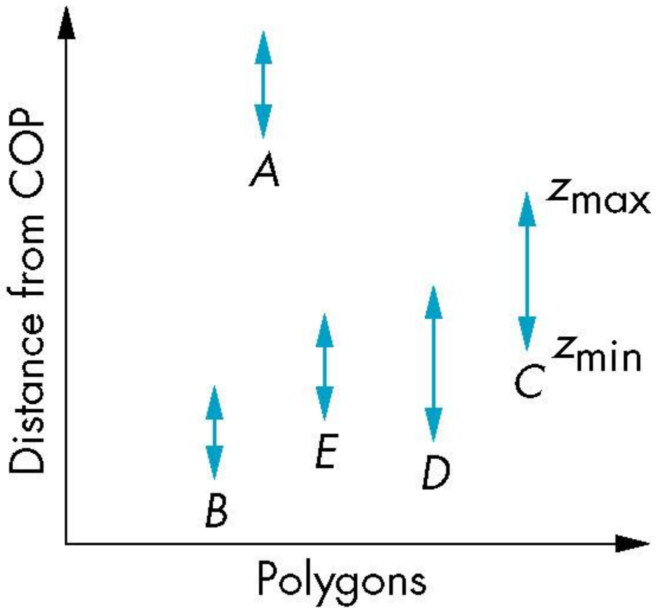
Fill B then A

Depth Sort

- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

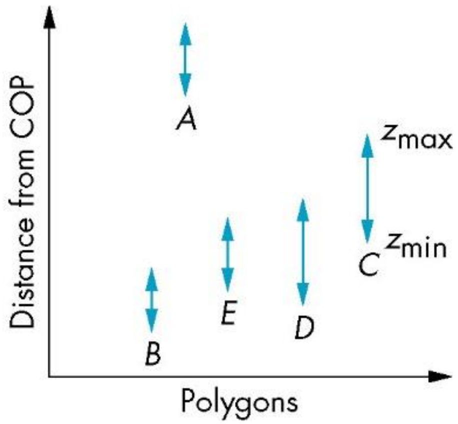
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

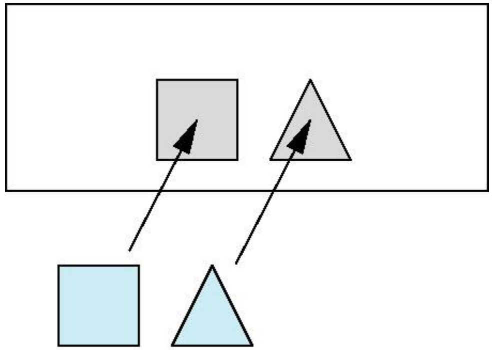


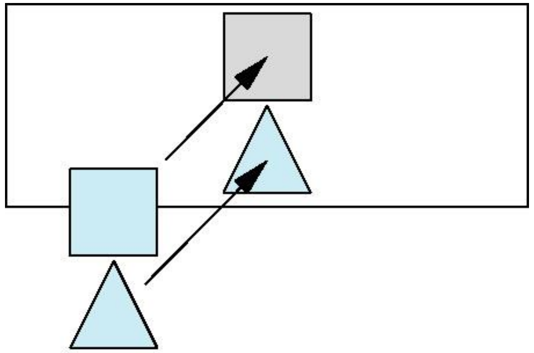
Easy Cases

- A lies behind all other polygons
  - Can render

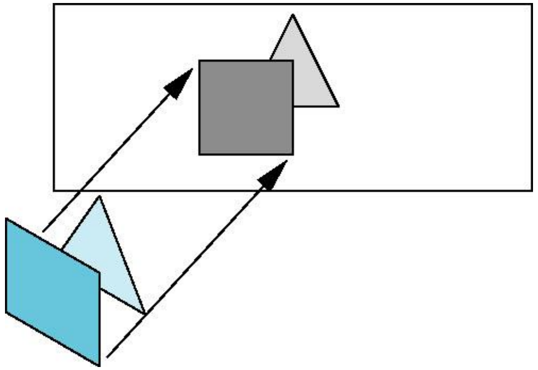


- Polygons overlap in z but not in either x or y
  - Can render independently

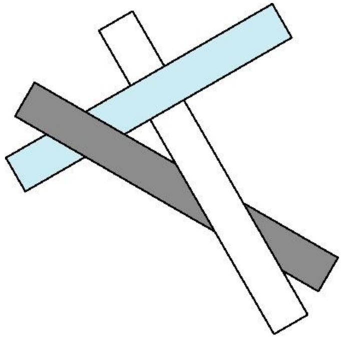




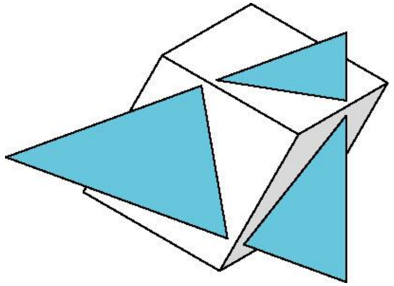
Hard Cases



Overlap in all directions but can one is fully on one side of the other

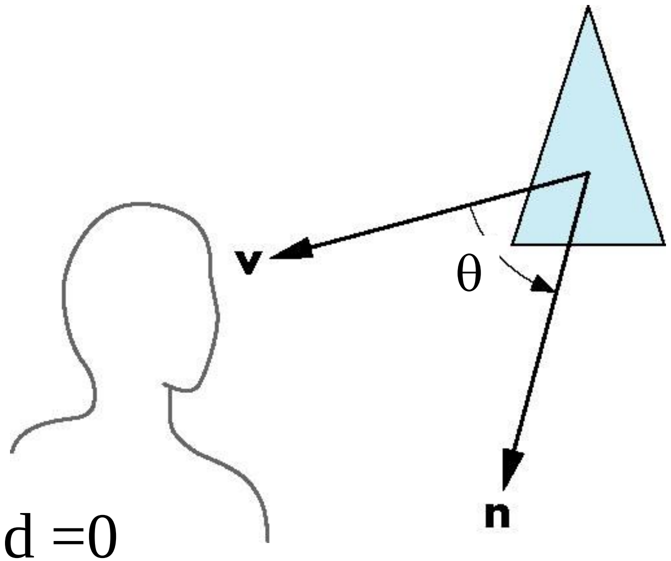


cyclic overlap



penetration

Back-Face Removal (Culling)



•face is visible iff $90 \geq \theta \geq -90$
equivalently $\cos\theta \geq 0$
or $\mathbf{v} \bullet \mathbf{n} \geq 0$

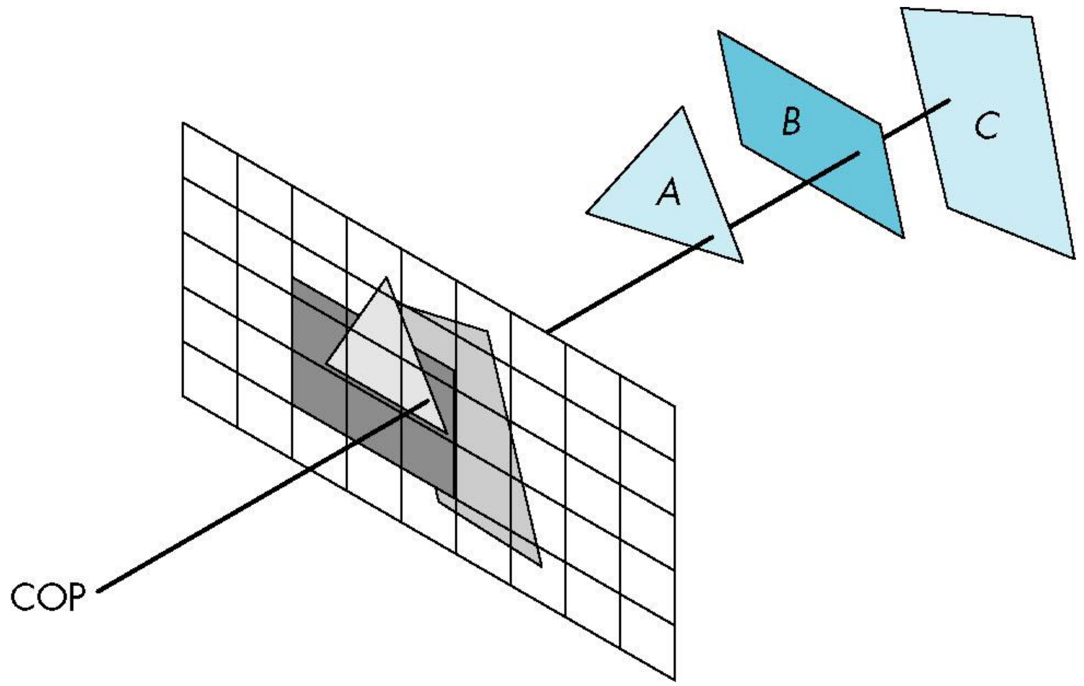
•plane of face has form $ax + by + cz + d = 0$
but after normalization $\mathbf{n} = (\,0\;0\;1\;0)^T$

•need only test the sign of c

•In OpenGL we can simply enable culling
but may not work correctly if we have nonconvex objects

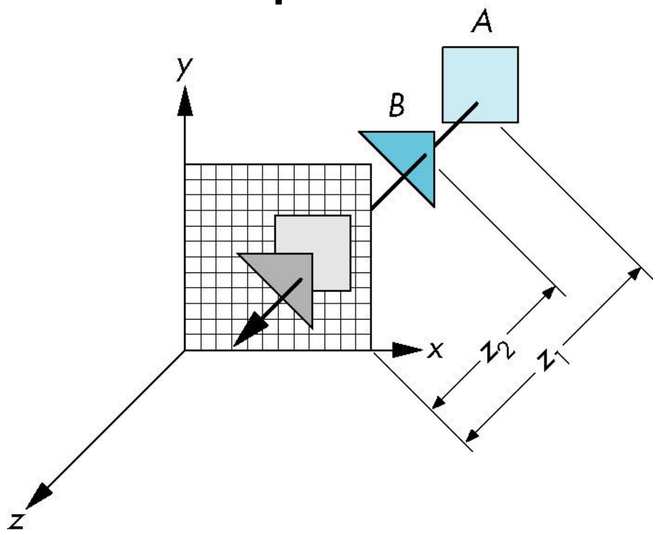
Image Space Approach

- Look at each projector (nm for an $n \times m$ frame buffer) and find closest of k polygons
- Complexity O(nmk)
- Ray tracing
- z-buffer



z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Efficiency

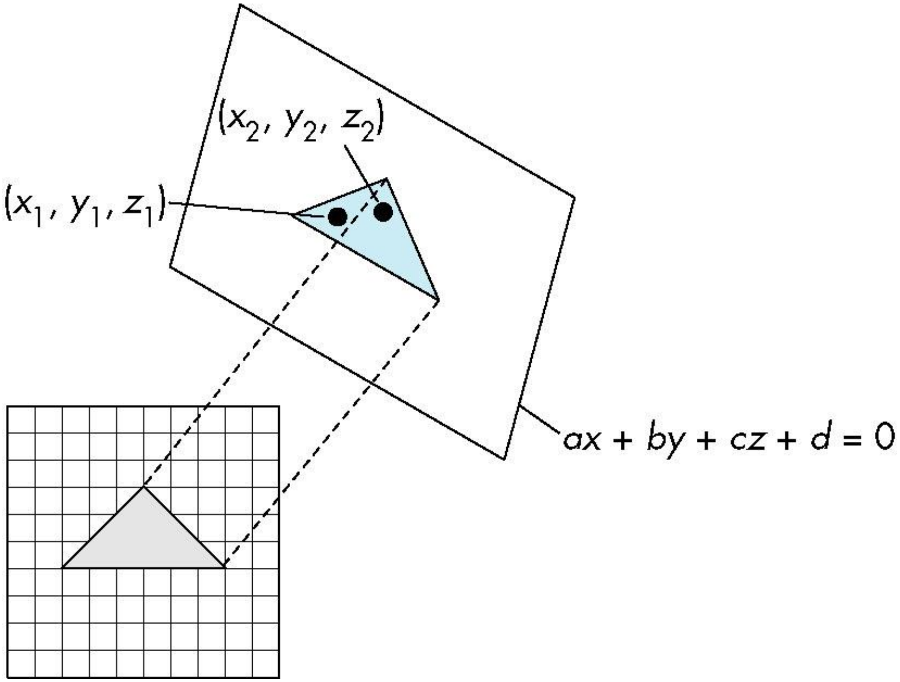
- If we work scan line by scan line as we move across a scan line, the depth changes satisfy $a\Delta x+b\Delta y+c\Delta z=0$

Along scan line

$$\Delta y = 0$$

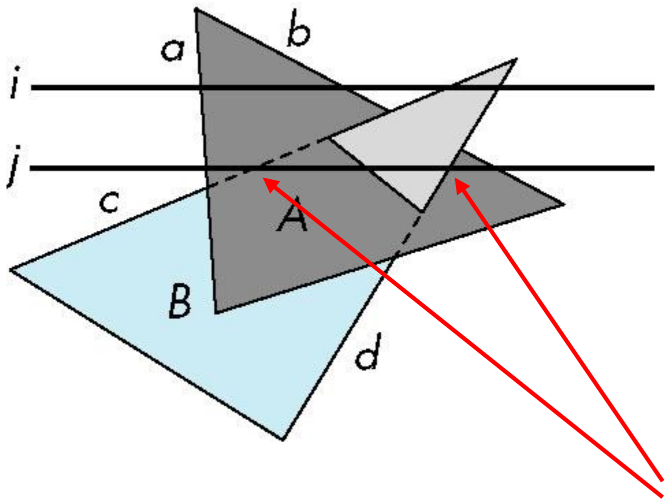
$$\Delta z = -\frac{a}{c}\Delta x$$

In screen space $\Delta x = 1$



Scan-Line Algorithm

- Can combine shading and hsr through scan line algorithm



scan line i: no need for depth information, can only be in no or one polygon

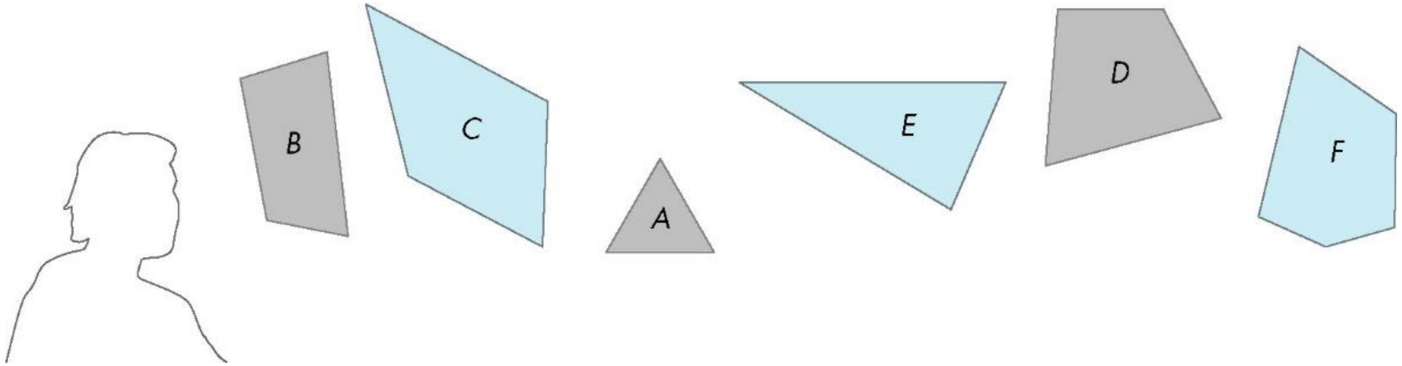
scan line j: need depth information only when in more than one polygon

Implementation

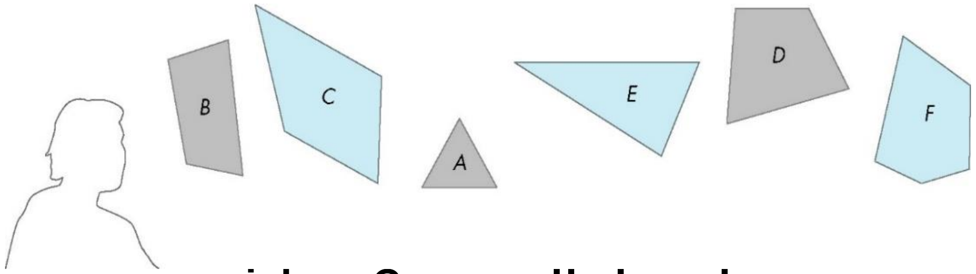
- Need a data structure to store
  - Flag for each polygon (inside/outside)
  - Incremental structure for scan lines that stores which edges are encountered
  - Parameters for planes

Visibility Testing

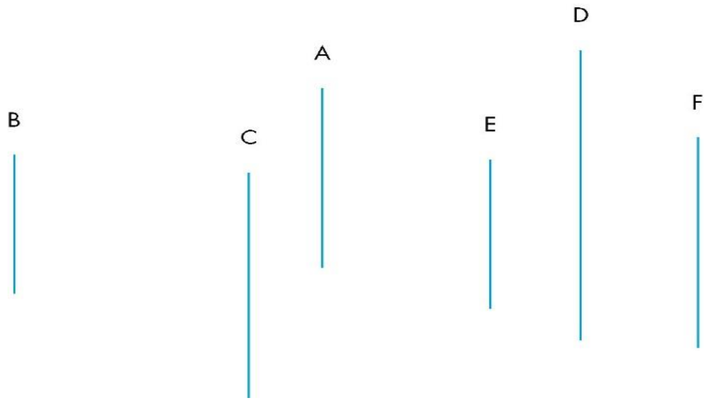
- In many realtime applications, such as games, we want to eliminate as many objects as possible within the application
  - Reduce burden on pipeline
  - Reduce traffic on bus
- Partition space with Binary Spatial Partition (BSP) Tree



Simple Example



consider 6 parallel polygons

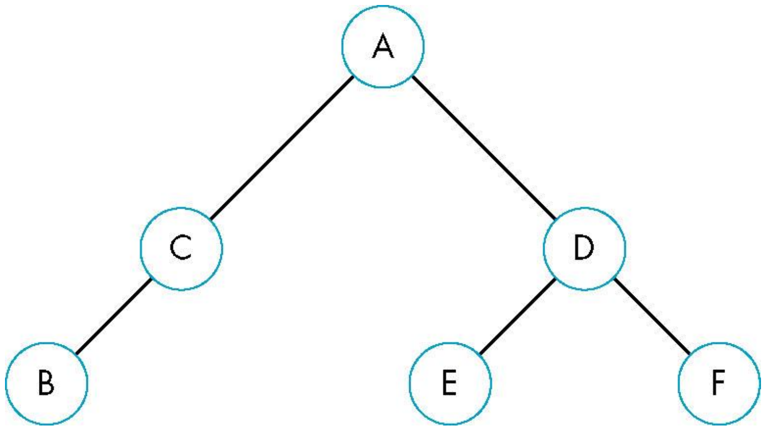


top view

The plane of A separates B and C from D, E and F

BSP Tree

- Can continue recursively
  - Plane of C separates B from A
  - Plane of D separates E and F
- Can put this information in a BSP tree
  - Use for visibility and occlusion testing



Rasterization

Objectives

- Survey Line Drawing Algorithms
  - DDA
  - Bresenham’s Algorithm
- Aliasing and Antialiasing

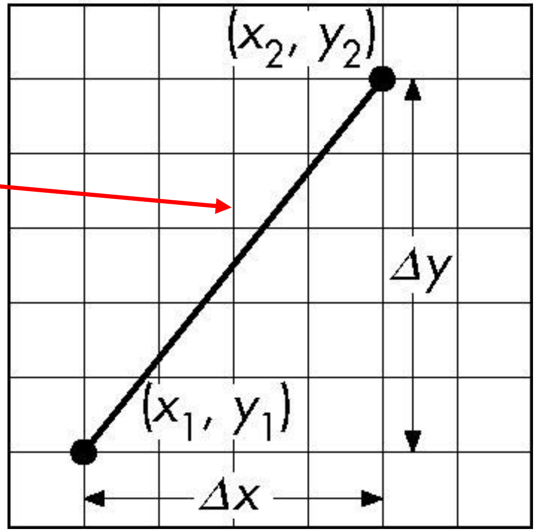
Rasterization

- Rasterization (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Scan Conversion of Line Segments

- Start with line segment in window coordinates with integer values for endpoints
- Assume implementation has a **`write_pixel`** function

y = mx + h



$$m = \frac{\Delta y}{\Delta x}$$

DDA Algorithm

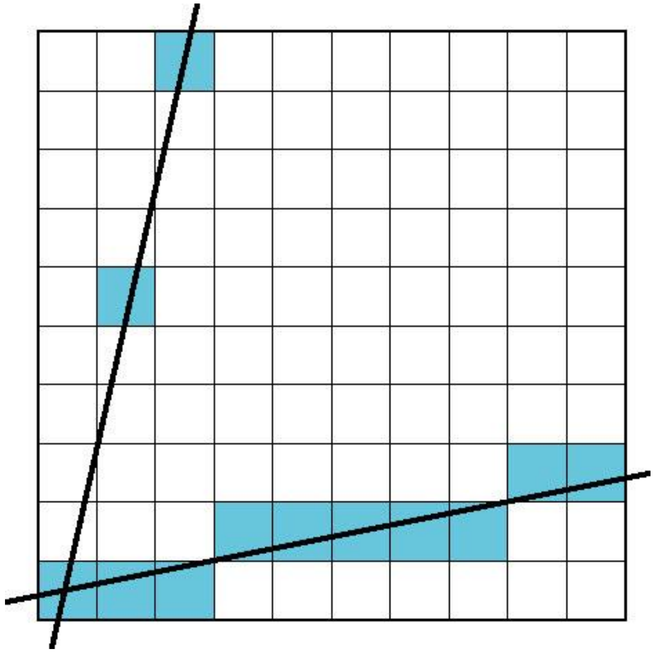
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line $y=mx+h$ satisfies differential equation

    $dy/dx = m = \Delta y/\Delta x = y_2-y_1/x_2-x_1$
- Along scan line $\Delta x = 1$

```
For(x=x1; x<=x2,ix++) {
    y+=m;
  write_pixel(x, round(y), line_color)
}
```

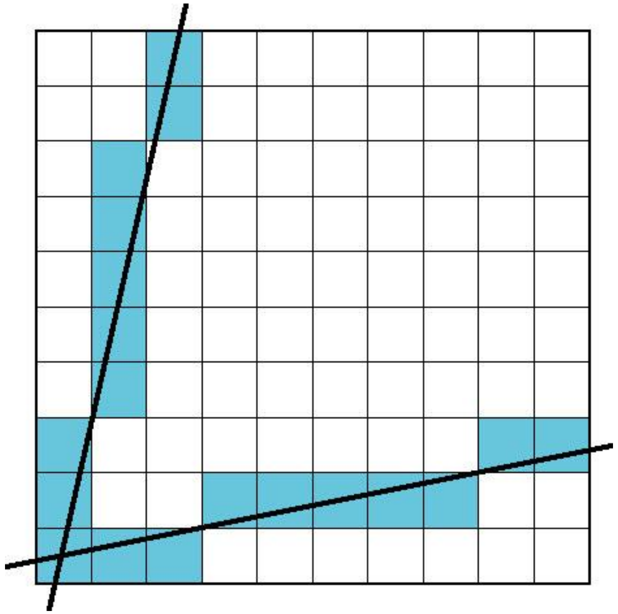
Problem

- DDA = for each x plot pixel at closest y
  - Problems for steep lines



Using Symmetry

- Use for $1 \geq m \geq 0$
- For $m > 1$, swap role of x and y
    - For each y, plot closest x

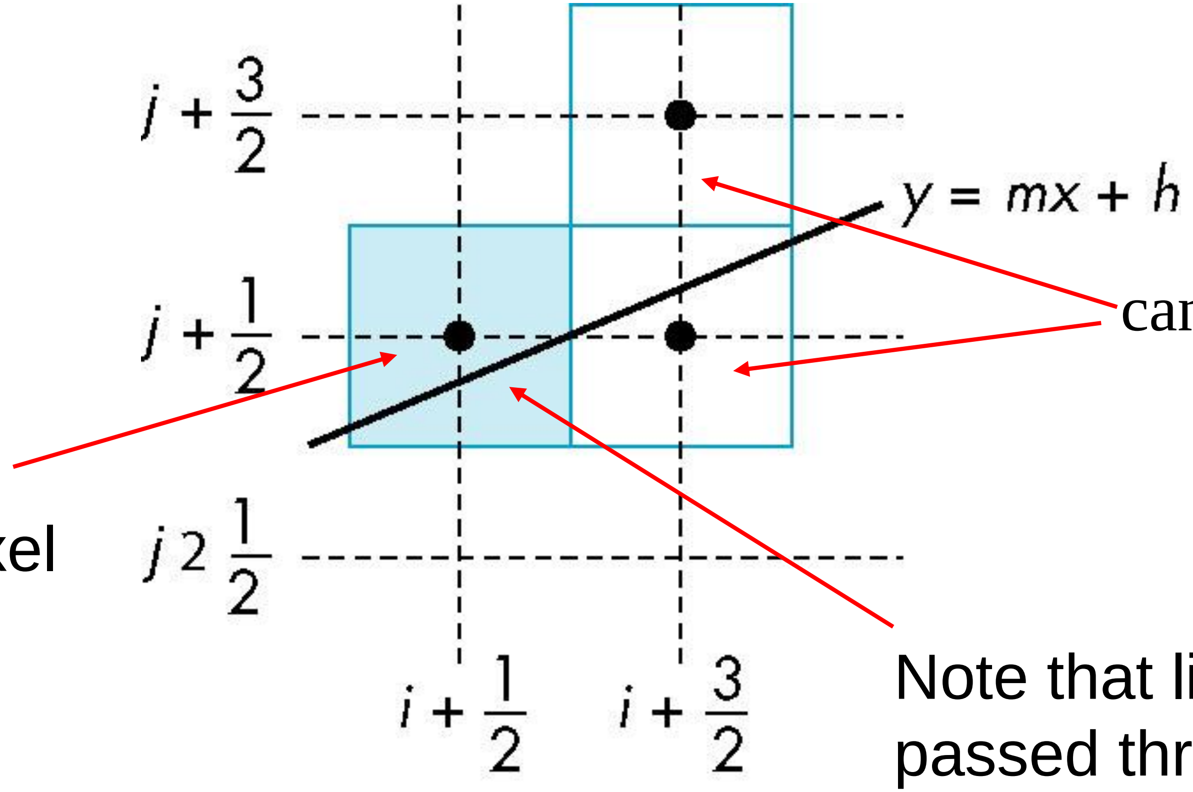


Bresenham's Algorithm

- DDA requires one floating point addition per step
- We can eliminate all fp through Bresenham's algorithm
- Consider only $1 \geq m \geq 0$
  - Other cases by symmetry
- Assume pixel centers are at half integers
- If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer

Candidate Pixels

$1 \geq m \geq 0$



candidates

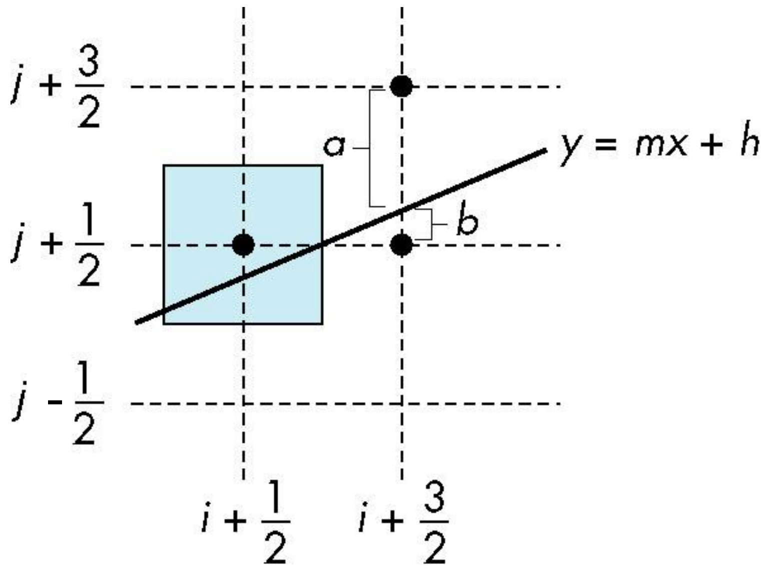
last pixel

Note that line could have passed through any part of this pixel

Decision Variable

$d = \Delta x(b-a)$

d is an integer
d > 0 use upper pixel
d < 0 use lower pixel



Incremental Form

- More efficient if we look at d_k, the value of the decision variable at $x = k$

$$d_{k+1} = d_k - 2\Delta y, \quad \text{if } d_k < 0$$
$$d_{k+1} = d_k - 2(\Delta y - \Delta x), \quad \text{otherwise}$$

- For each x, we need do only an integer addition and a test
- Single instruction on graphics chips

Polygon Scan Conversion

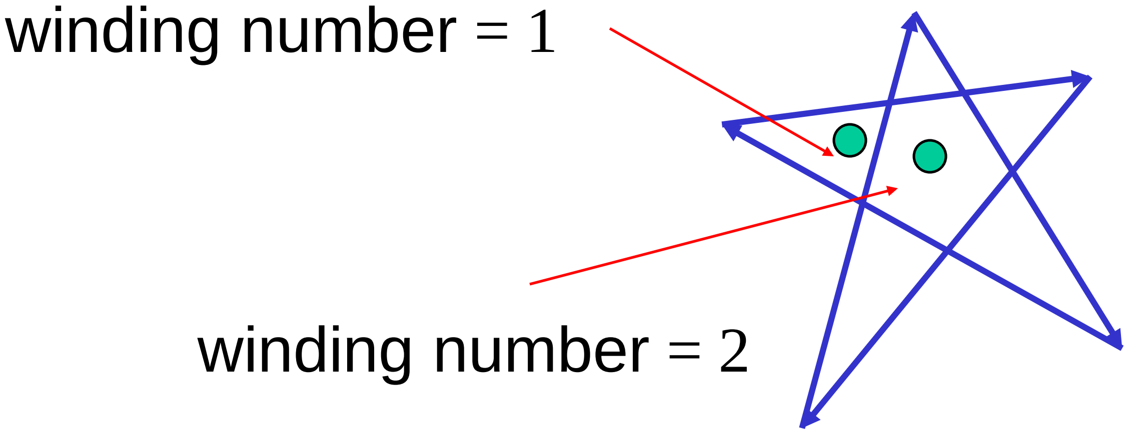
- Scan Conversion = Fill
- How to tell inside from outside
  - Convex easy
  - Nonsimple difficult
  - Odd even test
    - Count edge crossings

  - Winding number

odd-even fill

Winding Number

- Count clockwise encirclements of point



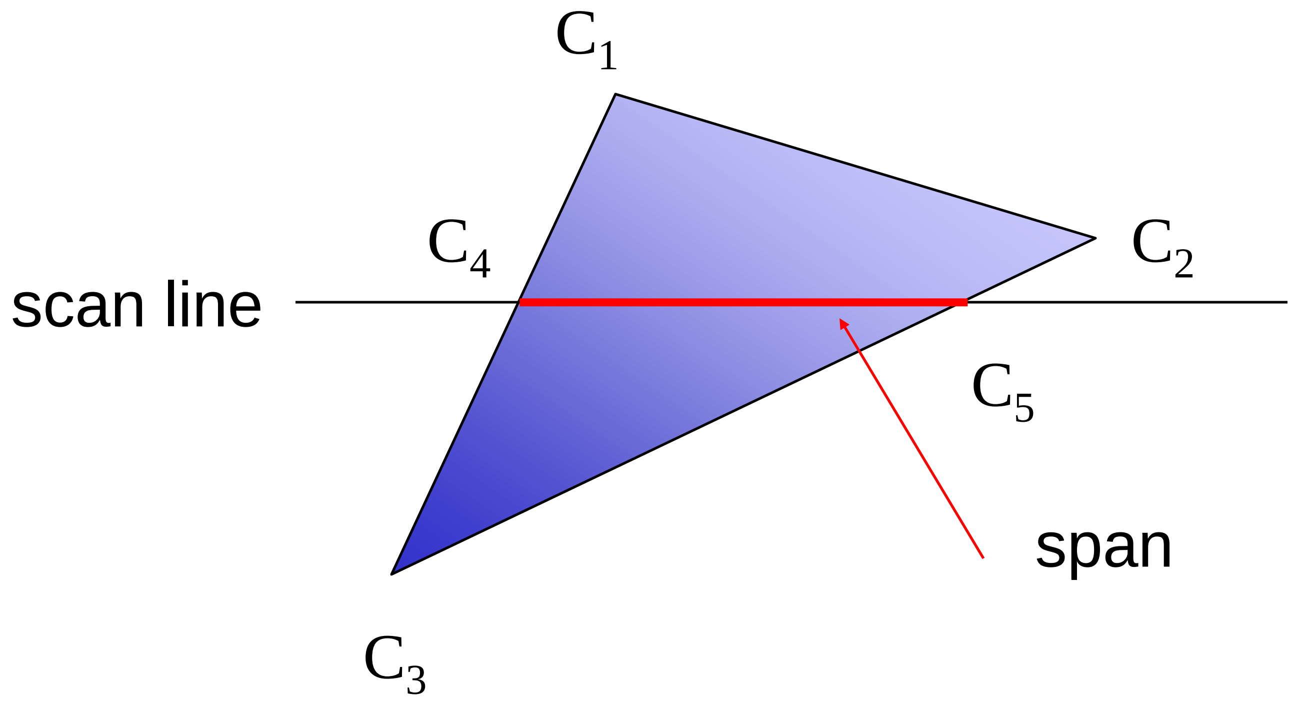
- Alternate definition of inside: inside if winding number $\neq 0$

Filling in the Frame Buffer

- Fill at end of pipeline
  - Convex Polygons only
  - Nonconvex polygons assumed to have been tessellated
  - Shades (colors) have been computed for vertices (Gouraud shading)
  - Combine with z-buffer algorithm
    - March across scan lines interpolating shades
    - Incremental work small

Using Interpolation

C_1 C_2 C_3 specified by **`glColor`** or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



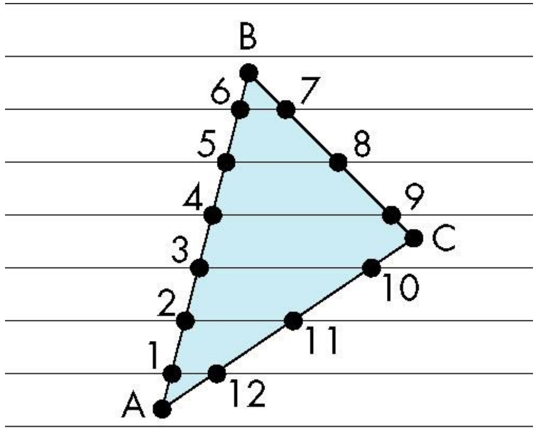
Flood Fill

- Fill can be done recursively if we know a seed point located inside (WHITE)
- Scan convert edges into buffer in edge/inside color (BLACK)

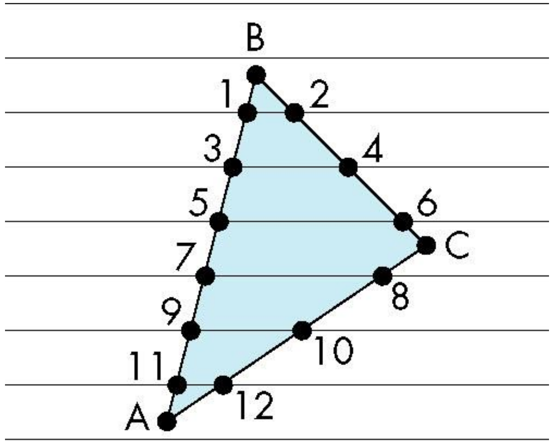
```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
        write_pixel(x,y,BLACK);
        flood_fill(x-1, y);
        flood_fill(x+1, y);
        flood_fill(x, y+1);
        flood_fill(x, y-1);
}   }
```

Scan Line Fill

- Can also fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort by scan line
  - Fill each span

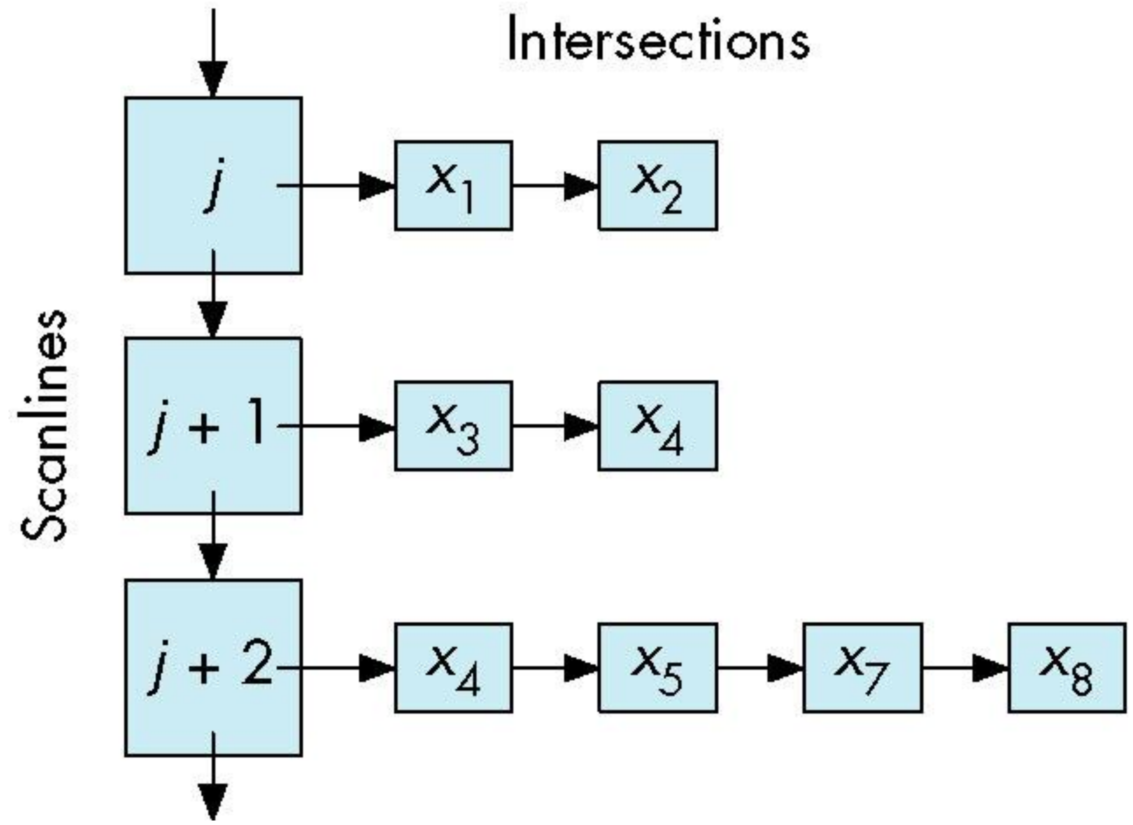


vertex order generated by vertex list



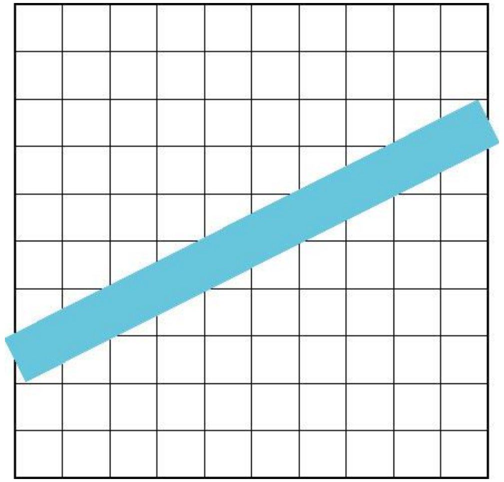
desired order

Data Structure



Aliasing

- Ideal rasterized line should be 1 pixel wide

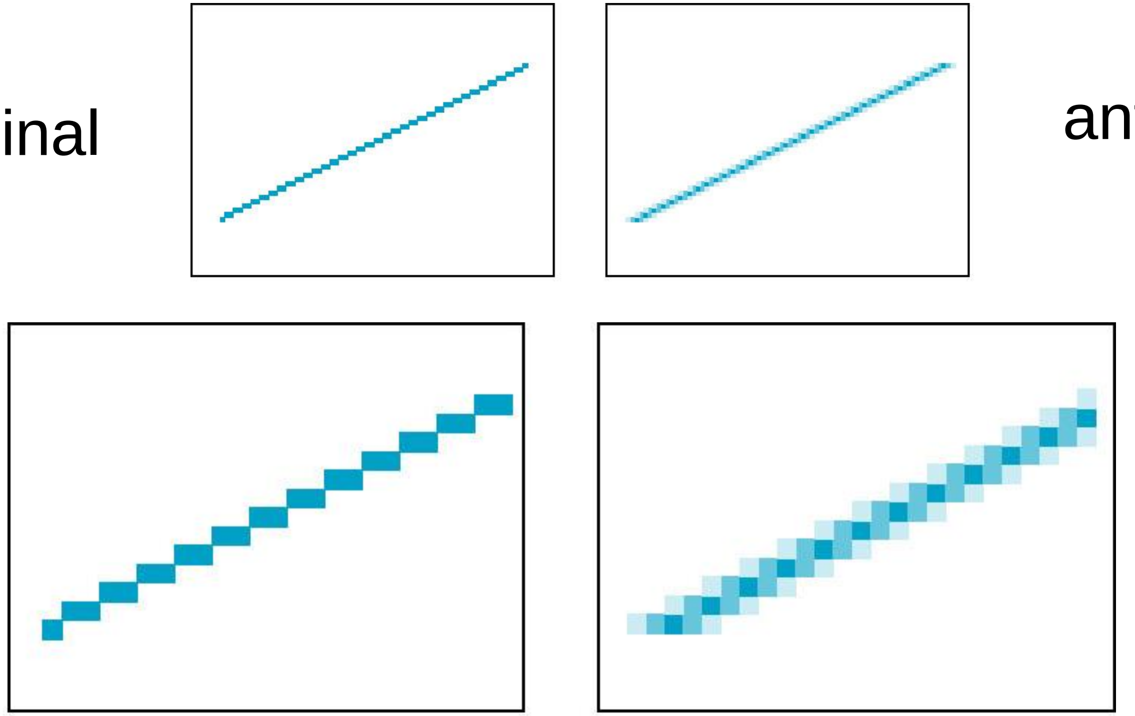


- Choosing best y for each x (or visa versa) produces aliased raster lines

Antialiasing by Area Averaging

- Color multiple pixels for each x depending on coverage by ideal line

original

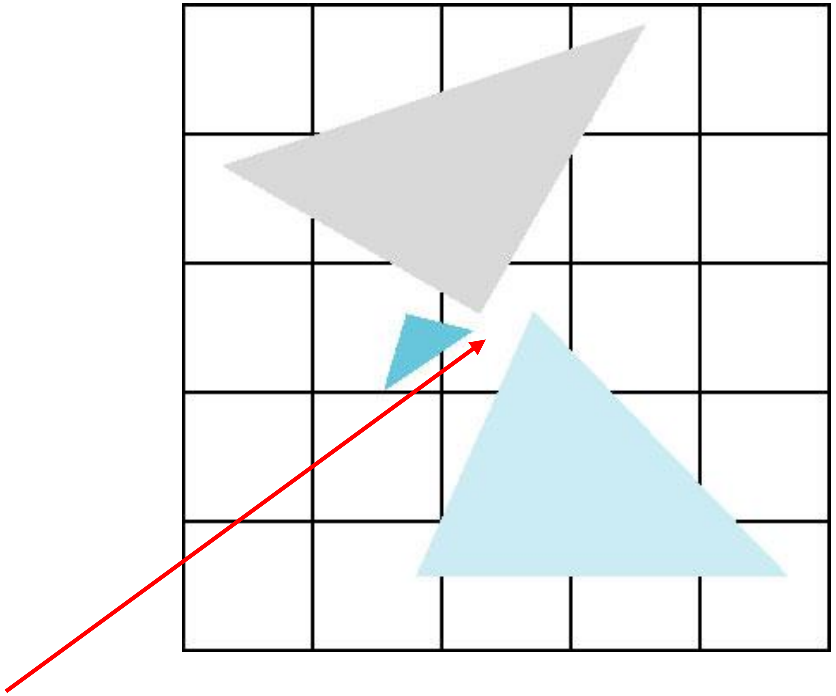


antialiased

magnified

Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color

Display Issues

Objectives

- Consider perceptual issues related to displays
- Introduce chromaticity space
  - Color systems
  - Color transformations
- Standard Color Systems

No Display Can Be Perfect

- An analog display device such as a CRT takes digital input (pixels) and outputs a small spot of color
- A Digital display such as a LCD display outputs discrete spots
- The eye merges (filters) these spots
- Sampling theory shows this process cannot be done perfectly

Perception Review

- Light is the part of the electromagnetic spectrum between ~350-750 nm
- A color $C(\lambda)$ is a distribution of energies within this range
- The human visual system has three types of cones on the retina, each with its own spectral sensitivity
- Consequently, only three values, the *tristimulus values*, are "seen" by the brain

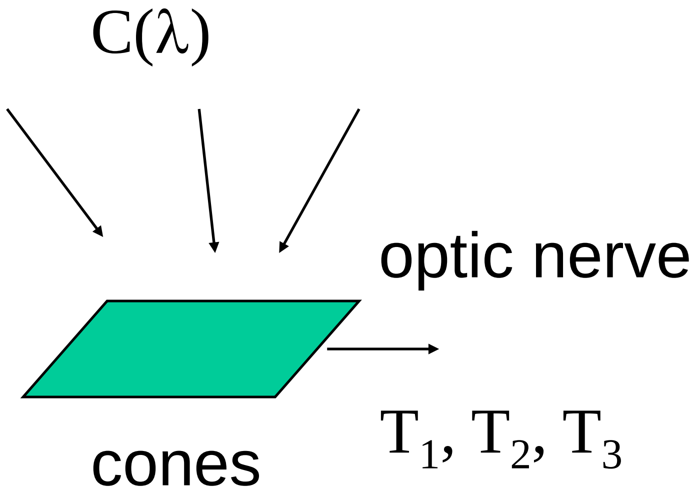
Tristimulus Values

- The human visual center has three cones with sensitivity curves $S_1(\lambda)$, $S_2(\lambda)$, and $S_3(\lambda)$
- For a color $C(\lambda)$, the cones output the tristimulus values

$$T_1 = \int S_1(\lambda)C(\lambda)d\lambda$$

$$T_2 = \int S_2(\lambda)C(\lambda)d\lambda$$

$$T_3 = \int S_3(\lambda)C(\lambda)d\lambda$$



Three Color Theory

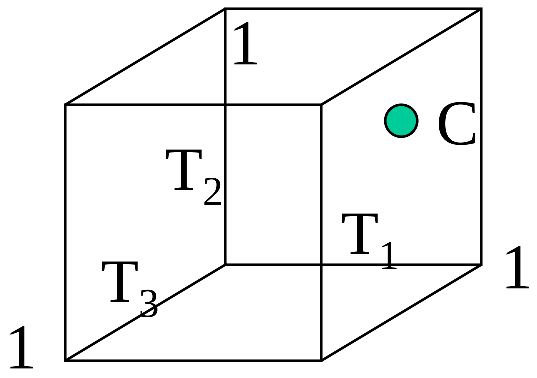
- Any two colors with the same tristimulus values are perceived to be identical
- Thus a display (CRT, LCD, film) must only produce the correct tristimulus values to match a color
- Is this possible? Not always
  - Different primaries (different sensitivity curves) in different systems

The Problem

- The sensitivity curves of the human are not the same as those of physical devices
- Human: curves centered in blue, green, and green-yellow
- CRT: RGB
- Print media: CMY or CMYK
- Which colors can we match and, if we cannot match, how close can we come?

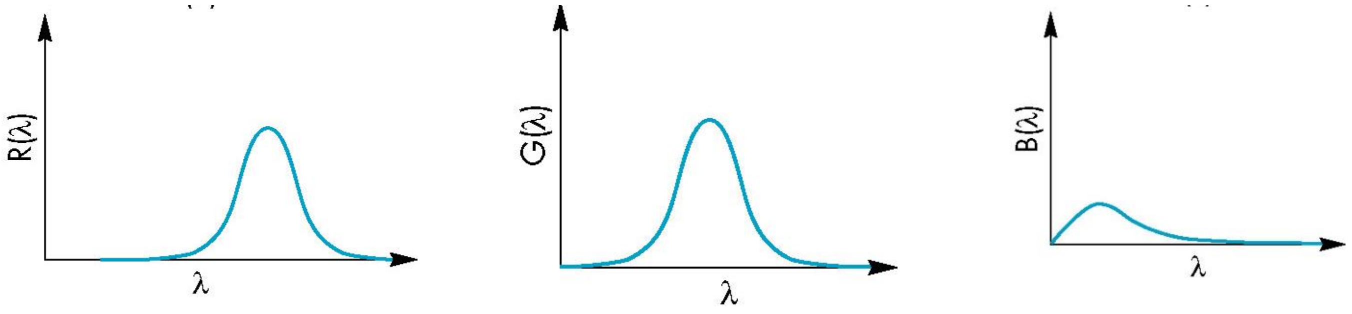
Representing Colors

- Consider a color $C(\lambda)$
- It generates tristimulus values T_1, T_2, T_3
  - Write $C = (T_1, T_2, T_3)$
  - Conventionally,we assume $1 \geq T_1, T_2, T_3 \geq 0$ because there is a maximum brightness we can produce and energy is nonnegative
  - C is a point in color solid



Producing Colors

- Consider a device such as a CRT with RGB primaries and sensitivity curves



- Tristimulus values

$$T_2 = \int G(\lambda)C(\lambda)d\lambda$$

$$T_1 = \int R(\lambda)C(\lambda)d\lambda$$

$$T_3 = \int B(\lambda)C(\lambda)d\lambda$$

Matching

- This T_1, T_2, T_3 is dependent on the particular device
- If we use another device, we will get different values and these values will not match those of the human cone curves
- Need a way of matching and a way of normalizing

Color Systems

- Various color systems are used
  - Based on real primaries:
    - NTSC RGB
    - UVW
    - CMYK
    - HLS
  - Theoretical
    - XYZ
- Prefer to separate brightness (luminance) from color (chromatic) information
  - Reduce to two dimensions

Tristimulus Coordinates

For any set of primaries, define

$$t_1 = \frac{T_1}{T_1 + T_2 + T_3}$$

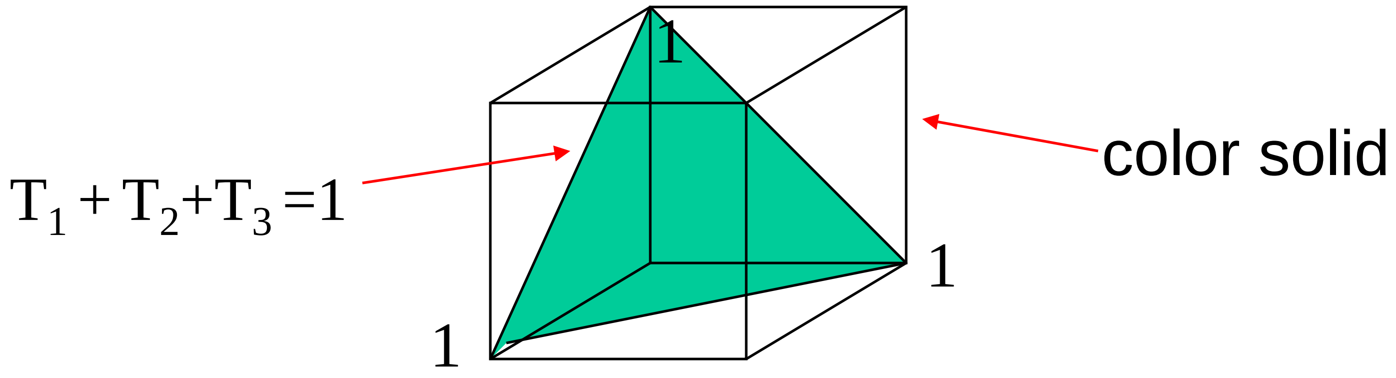
$$t_2 = \frac{T_2}{T_1 + T_2 + T_3}$$

$$t_3 = \frac{T_3}{T_1 + T_2 + T_3}$$

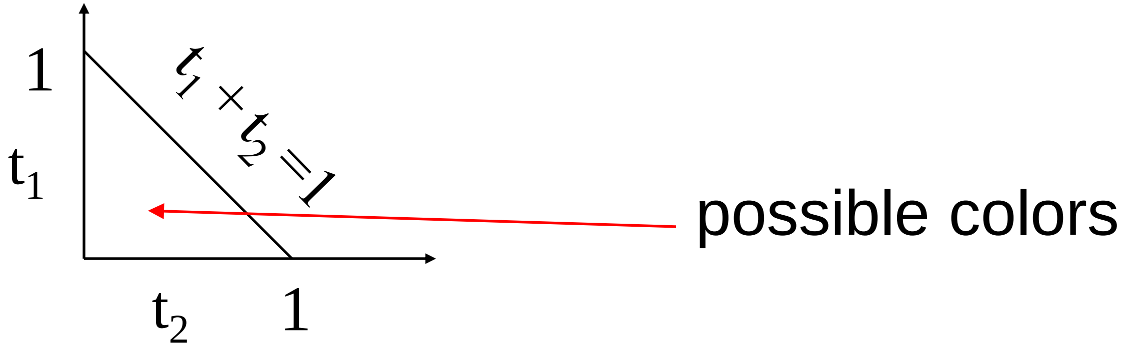
Note

$$t_1 + t_2 + t_3 = 1 \qquad\qquad 1 \geq t_1, t_2, t_3 \geq 0$$

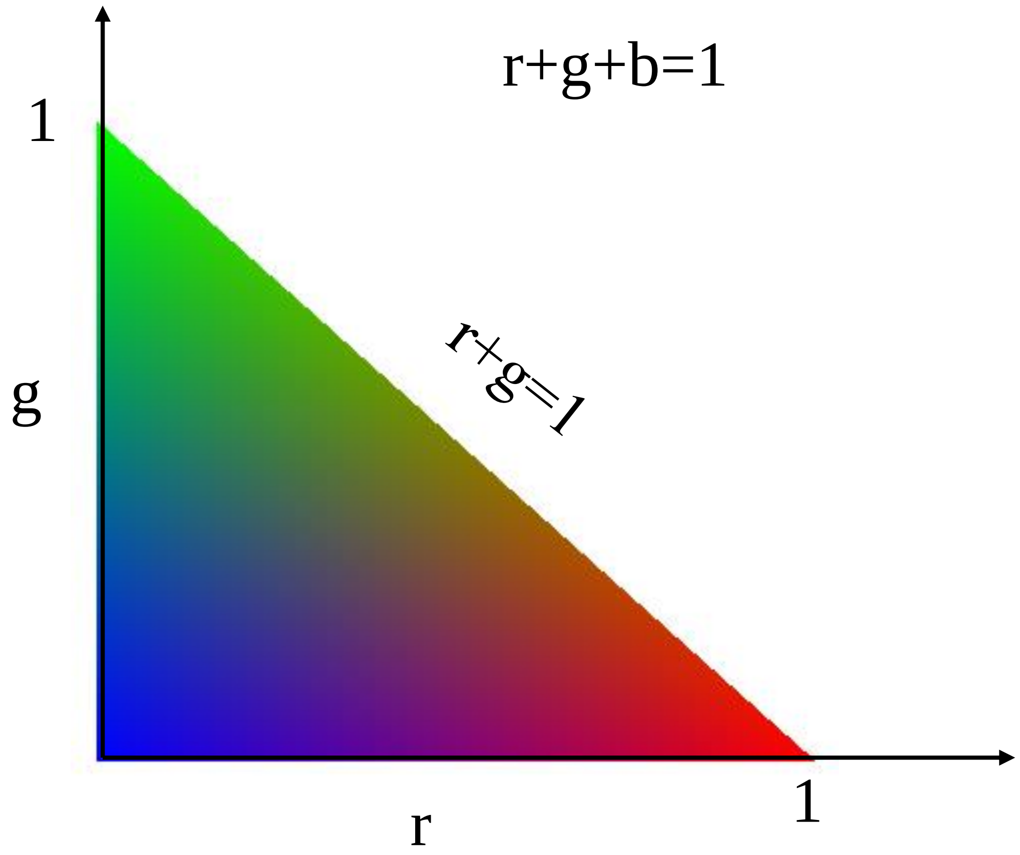
Maxwell Triangle



Project onto 2D: chromaticity space



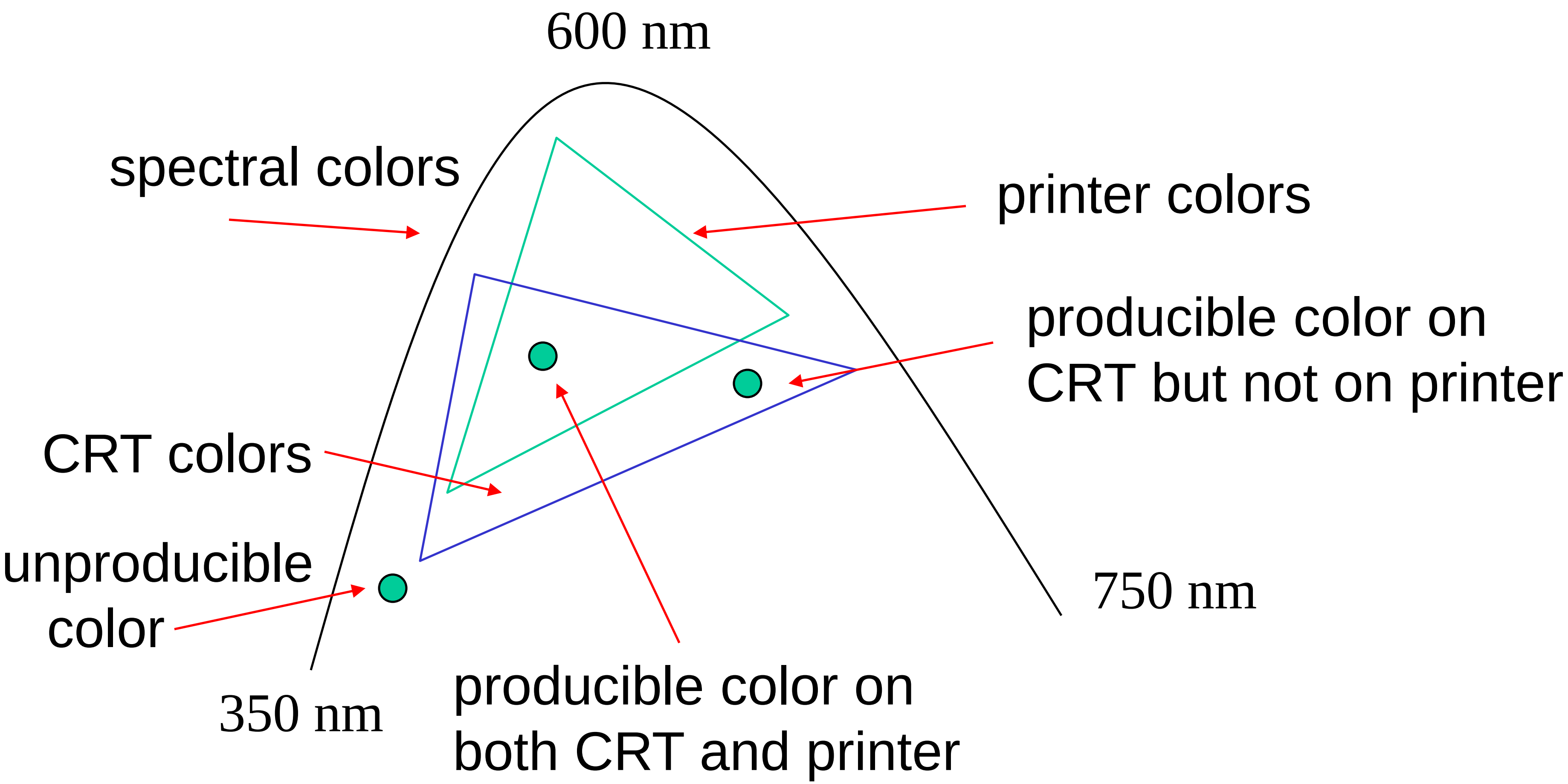
NTSC RGB



Producing Other Colors

- However colors producible on one system (its color gamut) is not necessarily producible on any other
- Not that if we produce all the pure spectral colors in the 350-750 nm range, we can produce all others by adding spectral colors
- With real systems (CRT, film), we cannot produce the pure spectral colors
- We can project the color solid of each system into chromaticity space (of some system) to see how close we can get

Color Gamuts



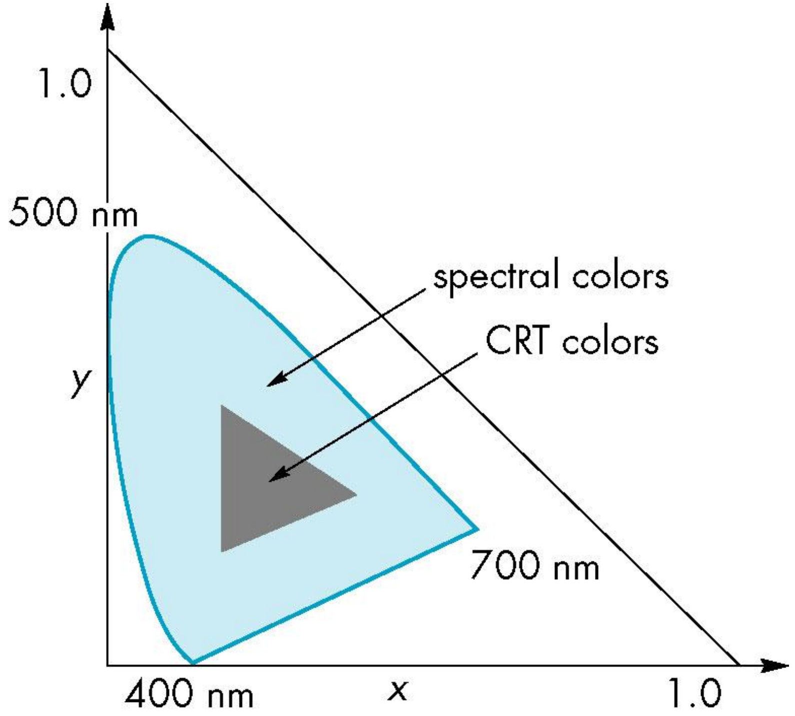
XYZ

- Reference system in which all visible pure spectral colors can be produced
- Theoretical systems as

there are no corresponding

physical primaries

- Standard reference system

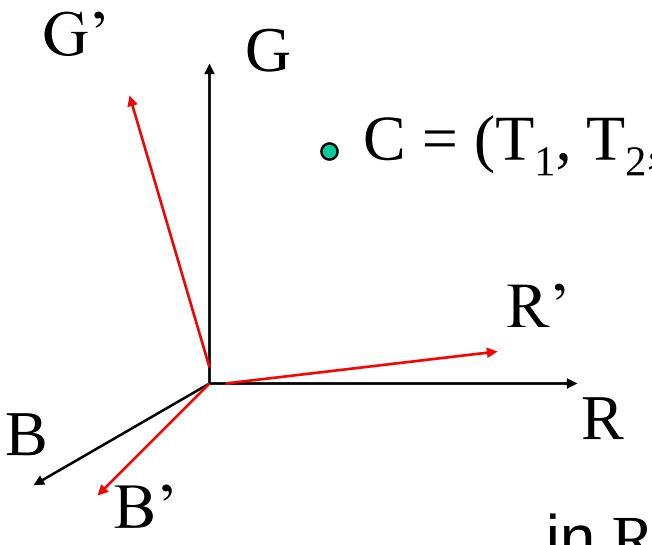


Color Systems

- Most correspond to real primaries
  - National Television Systems Committee (NTSC) RGB matches phosphors in CRTs
- Film both additive (RGB) and subtractive (CMY) for positive and negative film
- Print industry CMYK (K = black)
  - K used to produce sharp crisp blacks
  - Example: ink jet printers

Color Transformations

- Each additive color system is a linear transformation of another



$C = (T_1, T_2, T_3) = (T'_1, T'_2, T'_3)$

in R'G'B' system

in RGB system

RGB, CMY, CMYK

- Assuming 1 is max of a primary

  $C = 1 – R$

  $M = 1 – G$

  $Y = 1 – B$

- Convert CMY to CMYK by

  $K = \min(C, M, Y)$

  $C' = C – K$

  $M' = M – K$

  $Y' = Y - K$

Color Matrix

- Exists a 3 x 3 matrix to convert from representation in one system to representation in another

$$\begin{bmatrix} T'_1 \\ T'_2 \\ T'_3 \end{bmatrix} = \mathbf{M} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

- Example: XYZ to NTSC RGB
  - find in colorimetry references
- Can take a color in XYZ and find out if it is producible by transforming and then checking if resulting tristimulus values lie in (0,1)

YIQ

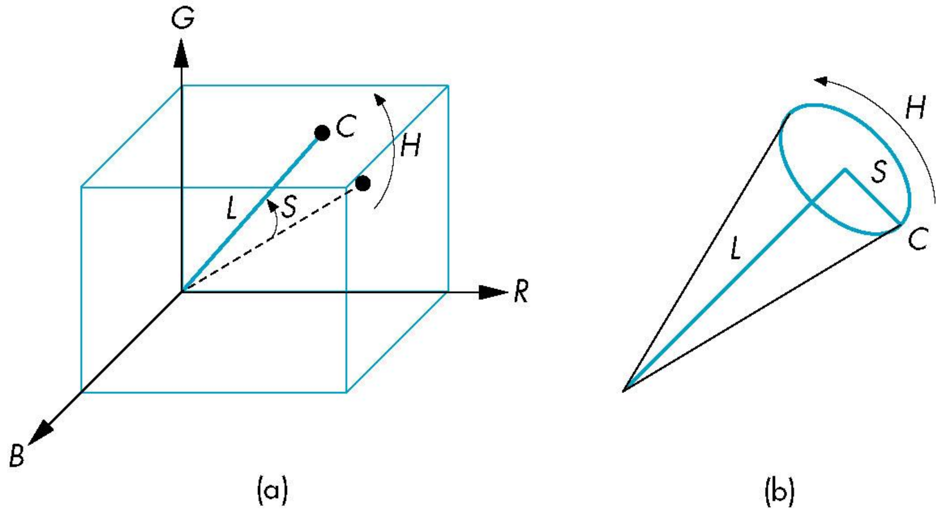
- NTSC Transmission Colors
- Here Y is the luminance
  - Arose from need to separate brightness from chromatic information in TV broadcasting

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- Note luminance shows high green sensitivity

Other Color Systems

- UVW: equal numerical errors are closer to equal perceptual errors
- HLS: perceptual color (hue, saturation, lightness)
  - Polar representation of color space
  - Single and double cone versions



Gamma

- Intensity vs CRT voltage is nonlinear

$$I = cV^{\gamma}$$

- Can use a lookup table to correct
- Human brightness response is logarithmic
  - Equal steps in gray levels are not perceived equally
  - Can use lookup table
- CRTs cannot produce a full black
  - Limits contrast ratio

sRGB

- Standard for Internet
- Adjust colors to match standard gamma of panels
  - match gamma over most of the range
  - enhance less bright colors
- OpenGL (soon WebGL?) can input sRGB and convert to RGB for processing and then back to sRGB