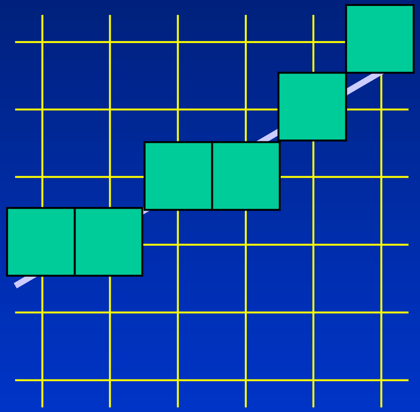
CSE528 Computer Graphics: Theory, Algorithms, and Applications

Hong Qin
Department of Computer Science
Stony Brook University (SUNY at Stony Brook)
Stony Brook, New York 11794-2424
Tel: (631)632-8450; Fax: (631)632-8334
qin@cs.stonybrook.edu
http://www.cs.stonybrook.edu/~qin

Rasterization

Per-pixel operations: ray-casting/ray-tracing

Screen = matrix



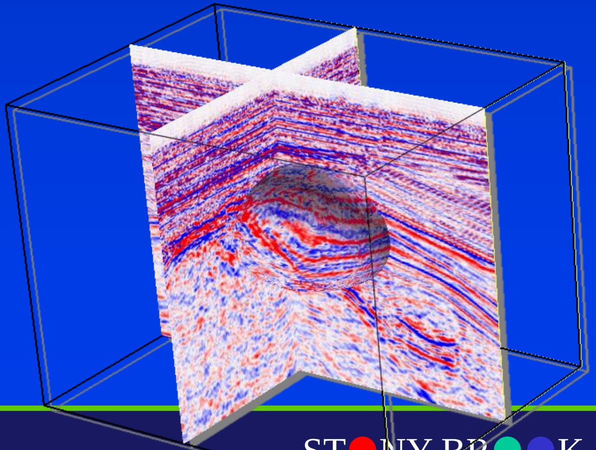
Scan conversion of lines:

- naive version
- Bresenham algorithm (integer-only)

Scan conversion of polygons

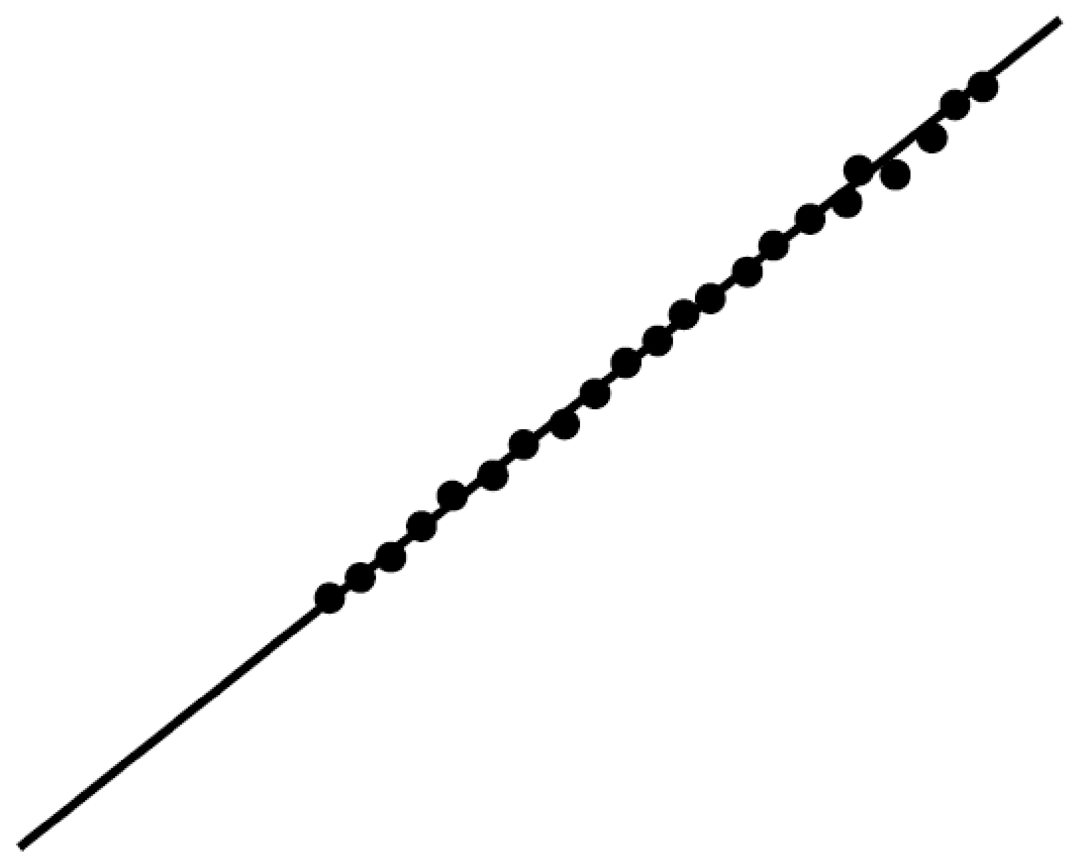
Aliasing / antialiasing

Texturing



Line Drawing (Rasterization)

- Convert continuous line to a set of discretized points
- Rasterization



Drawing of Line Geometry

- Why line drawing – the line is the most fundamental drawing primitive with many uses
  - Charts, engineering drawings, illustrations, 2D pencil-based animation, curve approximation
- Some desirable properties for any line drawing algorithm
  - A line should be straight; endpoint interpolation; uniform density for all lines; efficient
- Our current goal – efficient and correct line drawing algorithm
- Draw-line(x_0, y_0, x_1, y_1)

Algorithm Assumption

- Point samples on 2D integer lattice
- Bi-level display: on or off
- Line endpoints are all integer coordinates
- All line slopes are: |k| <=1
- Lines are ONE pixel thick

Line Geometry

- Explicit representation
- y = mx + b,
- The geometric meanings of these parameters: m – slope of the line; b – where it intercept y-axis (where x = 0)
- More derivations to simplify the equation
  - dy = y1 – y0
  - dx = x1 – x0
  - m = (dy) / (dx)

Simple Algorithm

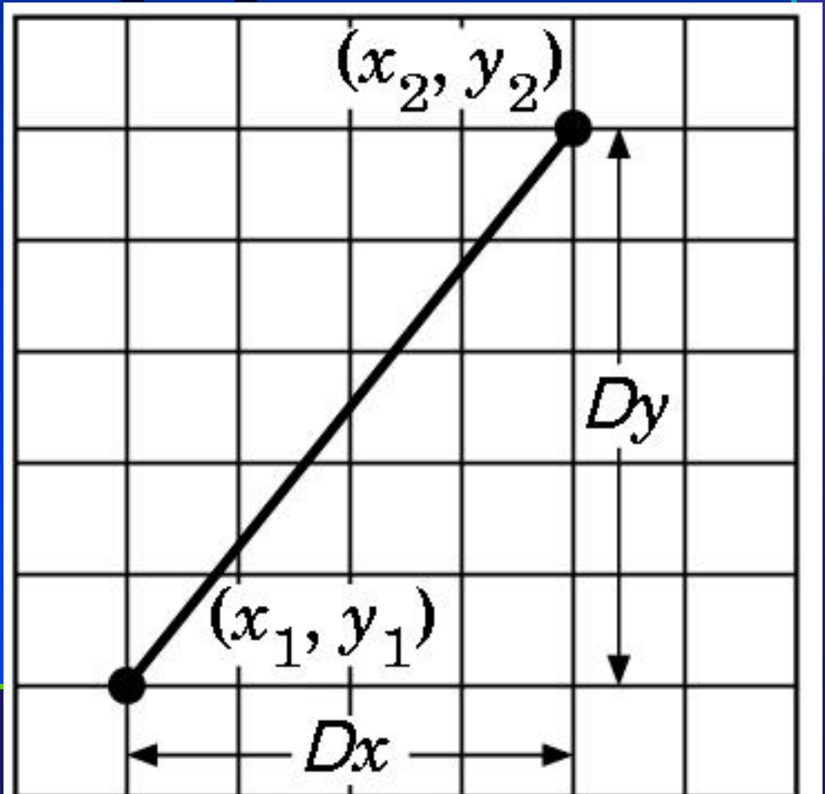
- Draw-line(x0, y0, x1, y1)

```
Let dy = y1 – y0
Let dx = x1 – x0
For x = x0 to x1
    y = rounding-operation(y0 + (x – x0) (dy / dx)
    draw-point(x,y)
End for
```

- Why does the above procedure work?
- Explicit definition of the line geometry
  - $y = (dy / dx) (x – x0) + y0$

Rendering Line Geometry (Rasterization)

- One of the fundamental tasks in computer graphics is 2D line drawing: How to render a line segment from (x_1, y_1) to (x_2, y_2)?
- Where do we start?
- Use the equation $y = mx + h$ (explicit)
- What about horizontal vs. vertical lines?



Further Improvement

- A more efficient algorithm

```
x = x0; y = y0
draw-point(x,y)
For x from x0 + 1 to x1
    $y = y + (dy / dx)$
End for
```

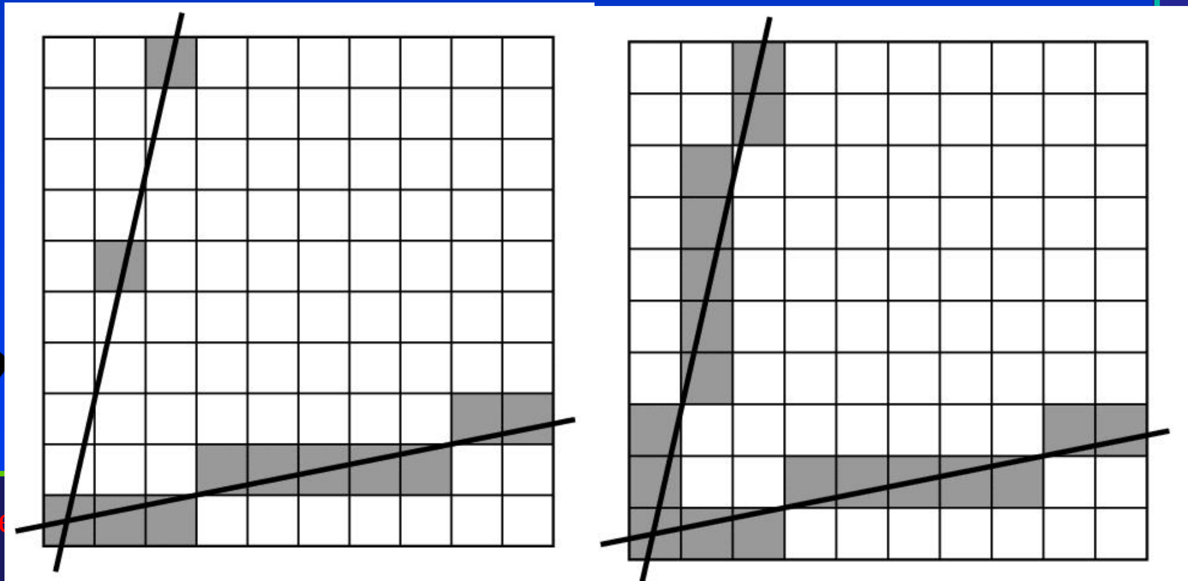
- Note that, $m = (dy / dx)$, and m is a float or double

DDA Algorithm

- So a digital differential analyzer (DDA)

```
for (x=$x_1$; x<=$x_2$; x++)
    y += m;
    draw_pixel(x, y, color)
```

- Handle slopes 0 <= m <= 1; handle others symmetrically
- Does this need floating point operations?



Further Improvement

- We are now seeking an integer-ONLY algorithm to handle all line geometry
- The above procedures will fail
- We must explore new schemes (beyond the line geometry we have already know till now)

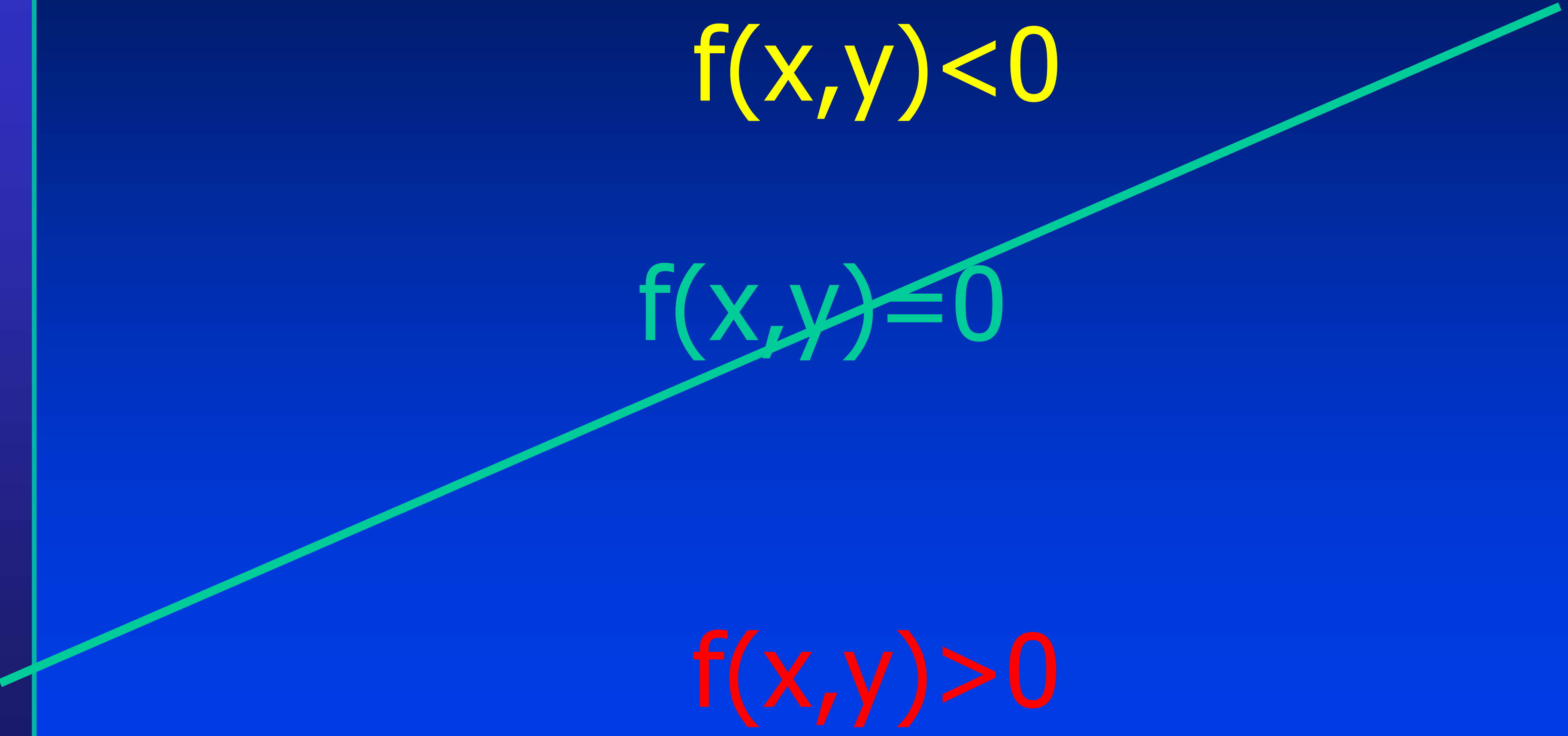
Midpoint Algorithm

- Implicit expression for the line geometry
  - $f(x,y) = (x - x0)*(dy) - (y - y0)*(dx)$
- What does this formulation provide us (compared with the previous derivations)?
- Fundamental ideas – spatial partitioning based on the signs!
  - If $f(x,y) = 0$, then (x,y) is on the line
  - If $f(x,y) > 0$, then (x,y) is below the line
  - If $f(x,y) < 0$, then (x,y) is above the line

Motivation

- Line geometry y=mx+b (explicit representation), not good enough for this task!
- Consider f(x,y)=0 (implicit representation) instead
- Clear geometry meaning and spatial relationship between a point and a line (on the line, below the line, above the line)
- A generic expression f(x,y)=ax+by+c=0
- Where does it come from?

Implicit Representation



f(x,y)<0

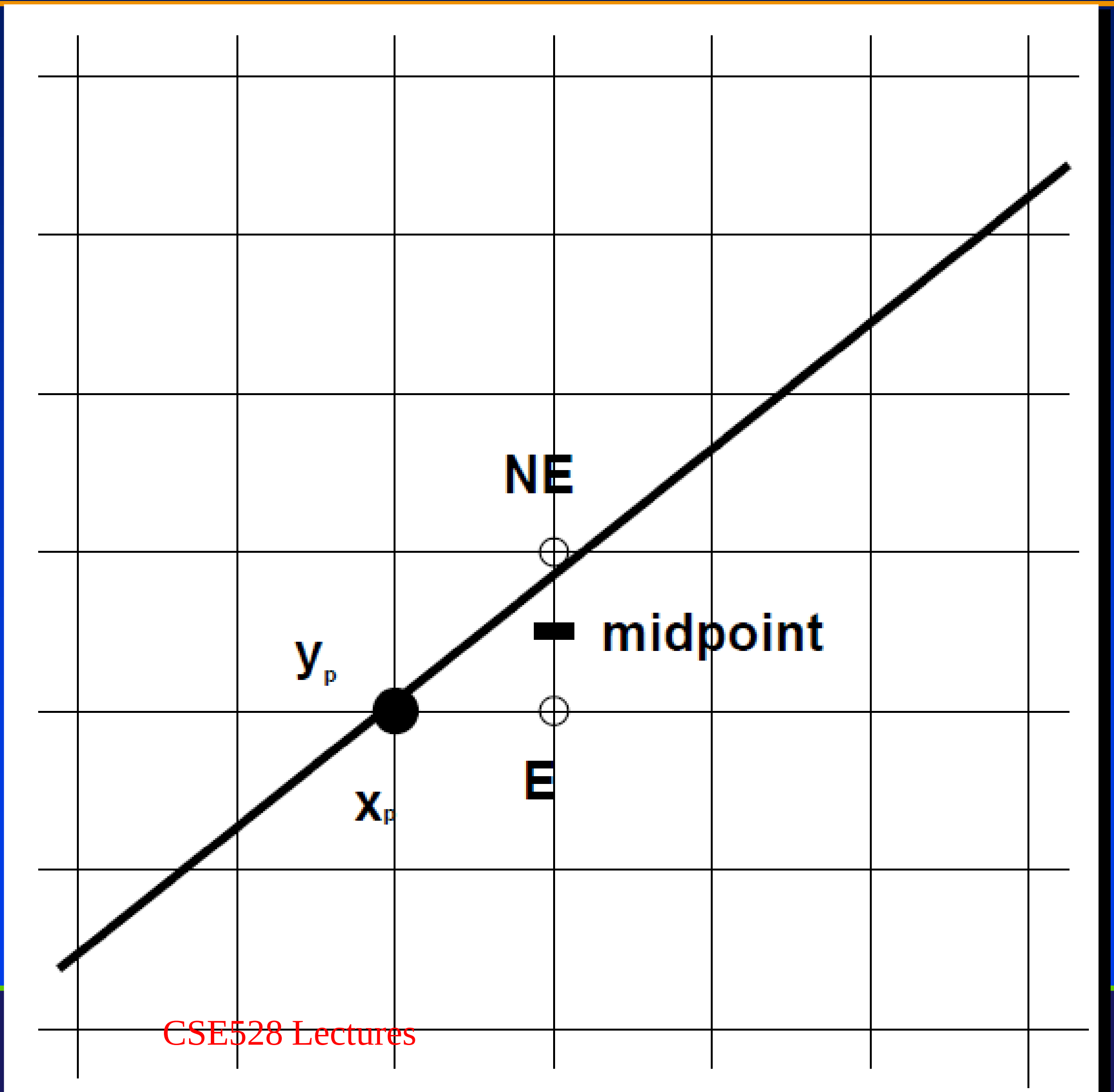
f(x,y)=0

f(x,y)>0

Line Geometry (AGAIN)

- $f(x,y)=(x-x1)dy - (y-y1)dx$
- $dy=y2-y1$
- $dx=x2-x1$
- Please DO understand the geometric meanings of these symbols

Midpoint Motivation



Midpoint Motivation

- We are actually considering d = f(xp + 1, yp +0.5)
- There are three different cases
  - If d < 0, line is below the (current) midpoint, then choose E
  - If d >0, lie is above the midpoint, choose NE
  - If d =0, line is passing through the midpoint, either E or NE

Midpoint Algorithm

- If E is chosen, then the NEW E would be (x+2, y), the NEW NE would be (x+2, y+1); the NEW MIDPOINT is (x+2,y+0.5)
- If NE is chosen, then the NEW E would be (x+2,y+1), the new NE would be (x+2,y+2); the NEW MIDPOINT is (x+2,y+0.5)
- Back to the line geometry derivation…

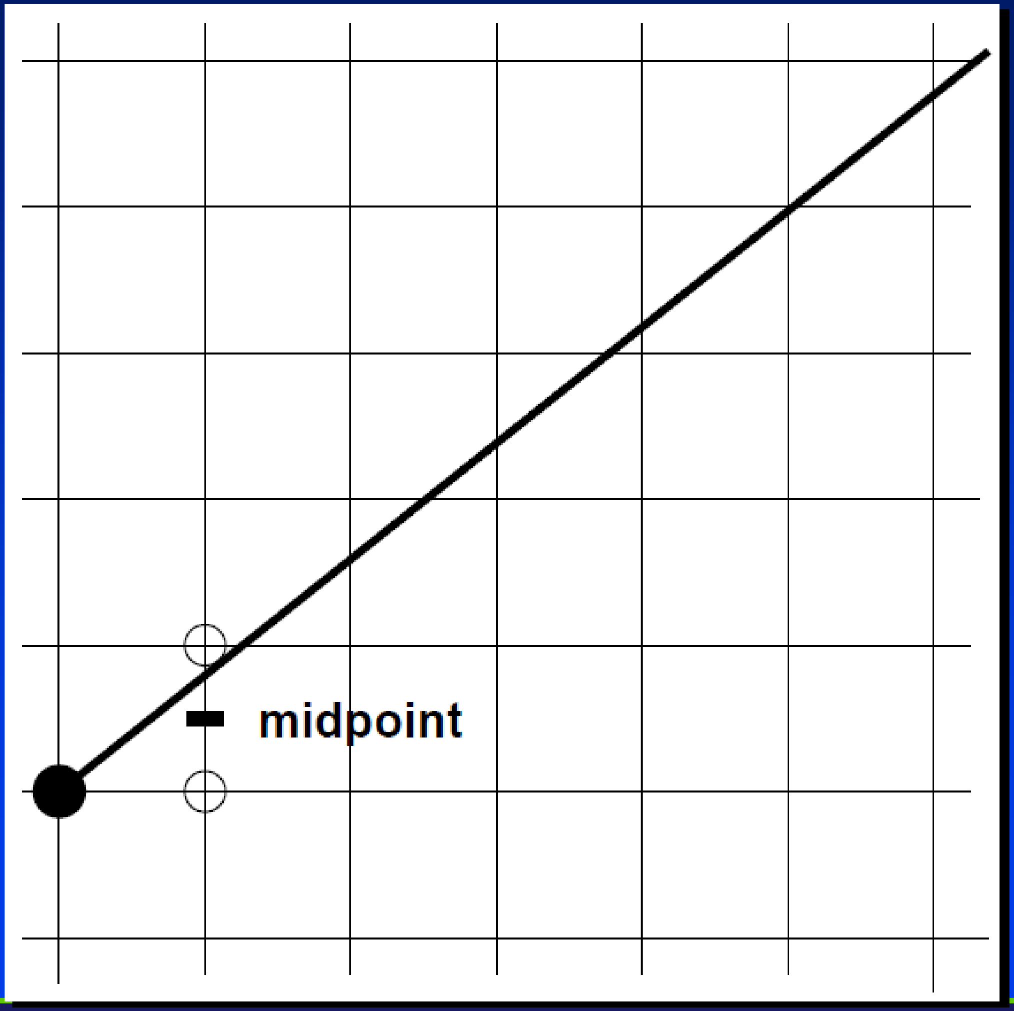
Recursive Algorithm

- Midpoint algorithm is a recursive algorithm!
- For recursive algorithm, we MUST consider the subsequent steps (by traversing all cases respectively)!
- If E is chosen, then the NEW E is (xp + 2, yp), the NEW NE is (xp + 2, yp +1), the NEW midpoint is (xp + 2, yp + 0.5)
  - d_new = f (xp + 2, yp + 0.5)
  - d_old = f (xp + 1, yp +0.5)
  - d_new = d_old + (dy)

Recursive Algorithm

- If NE is chosen, the NEW E is (xp +2, yp +1), the NEW NE is (xp + 2, yp + 2), the NEW midpoint is (xp + 2, y + 1.5)
  - d_new = f(xp + 2, yp + 1.5)
  - d_old = f(xp +1, yp + 0.5)
  - d_new = d_old + (dy – dx)
- This process MUST repeat recursively, stepping along x from x0 to x1

Midpoint Initialization



Initialization

- How about the initialization process
- At the beginning,
  - xp = x0
  - yp = y0
  - d_old = f(x0 +1, y0 +0.5) = (dy) – (dx) * (1/2)

Midpoint Algorithm

- draw-line(x0, y0, x1, y1)
  - Int x0, y0, x1, y1
  - { int dx, dy, inc_E, inc_NE, x, y,
  - real d
  - dx = x1 – x0
  - dy = y1 – y0
  - d = (dy) – (dx) * (1/2)
  - inc_E = dy
  - inc_NE = dy – dx
  - y = y0
  - for x from x0 to x1
  - if d>0, then d = d + inc_NE, y + 1, else d = d + inc_E
  - end for
  - }

Midpoint Algorithm

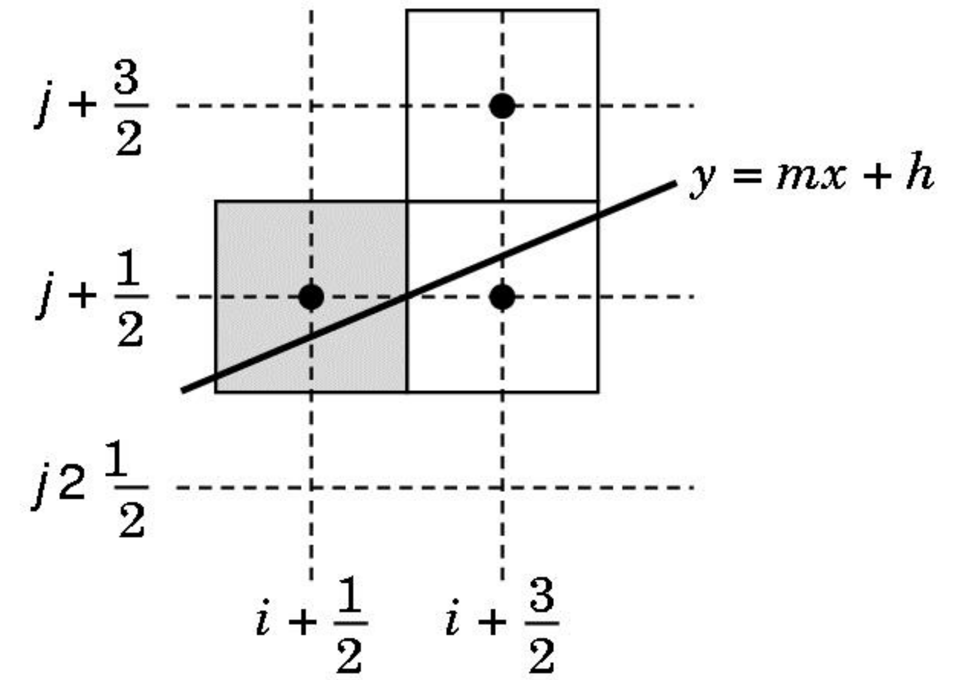
- d is NOT an integer, however, ONLY the sign MATTERS!
- We prefer an integer-ONLY algorithm!!!
  - $g(x,y) = 2\ f(x,y)$
  - d becomes 2d
  - then $d = 2(dy) - (dx)$

Integer-only Algorithm

- Midpoint algorithm is an integer-only algorithm
- The complete c-code implementation is available from the textbook and/or internet!
- The fundamental assumption is that, the line slope is positive, but controllable (its value is no more than 1)
- What about other cases?
- Possible generalizations to cover all cases?

Bresenham's Algorithm

- The DDA algorithm requires a floating point *add* and *round* for each pixel: Can we eliminate?
- Note that at each step we will go E or NE. How to decide which one (from two possible points)?

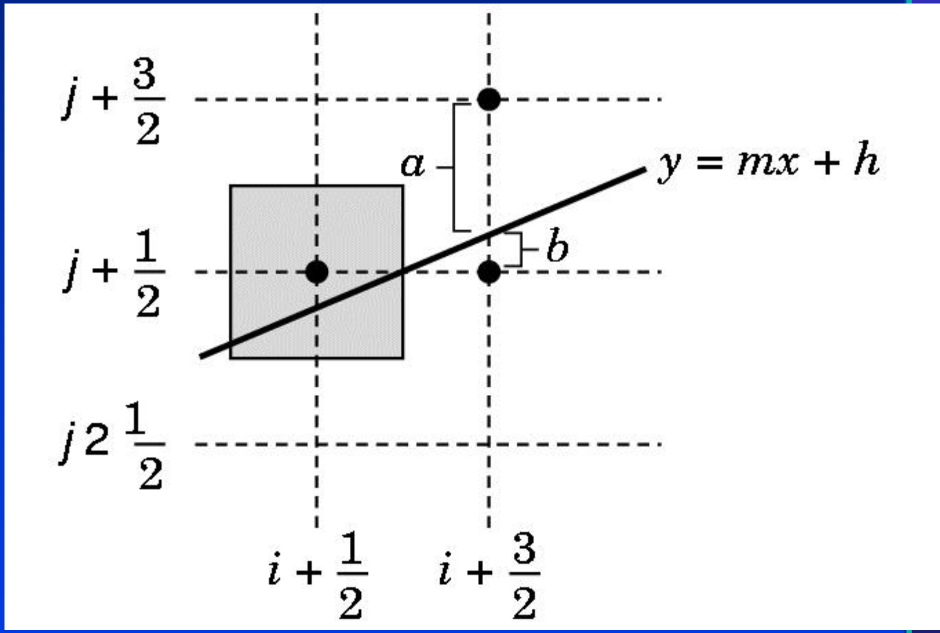


Bresenham’s Algorithm

- Also called the midpoint algorithm
- The key idea: consider d=f(x+1,y+0.5) and only pay attention to its sign!!!
- Midpoint algorithm is a recursive algorithm
- For recursive algorithm, we MUST consider the subsequent step!

Bresenham Decision Variable

- Bresenham algorithm uses decision variable d=a-b, where a and b are distances to NE and E pixels



- If d>=0, go NE;
  if d<0, go E
- Let $d=(x_2-x_1)(a-b) = d_x(a-b)$ [only sign matters]
- Substitute for a and b using line equation to get integer math (but lots of it)
- $d=(a-b)\, d_x = (2j+3)\, d_x - (2i+3)\, d_y - 2(y_1 d_x - x_1 d_y)$
- But note that $d_{k+1}=d_k + 2d_y$ (E) or $2(d_y-d_x)$ (NE)

Bresenham’s Algorithm

- Set up loop computing *d* at x_1, y_1

```
for (x=x1; x<=x2; )
    x++;
    d += 2dy;
    if (d >= 0) {
        y++;
        d -= 2dx; }
    drawpoint(x,y);
```

- Pure integer math, and not much of it
- So easy that it’s built into one graphics instruction (for several points in parallel)

Possible Extensions

- The idea is generalizable to other geometric primitives
- Algorithms for circle-drawing
- Algorithms for ellipses, conic section drawing
- Once again, the book (or the internet) has all the c-code programs for such tasks
- Generations to polynomial curves?

Modifying the Previous Algorithm

- Make it an integer-ONLY algorithm
- Our earlier assumptions
  - slopes: $0 <= (dy) / (dx) <= 1$
  - line endpoints are all integer coordinates
- How about other cases

Handling All Other Cases

- Generalizations
  - negative slope
  - slope larger than 1
- If the slope is larger than 1, we use symmetry to switch x and y (you are NOT displaying (x,y), you should display (y,x))!
- In negative slope, we should use x and (-y)

Extensions to Handle Curves

- Generalizations to handle all cases for line drawing
- Algorithms for circle-drawing
- Algorithms for ellipses, conic section drawing
- Algorithms for cubic curve drawing
- Algorithms to handle any type of curves?

Circles

- Implicit expression of a circle f(x,y)=0

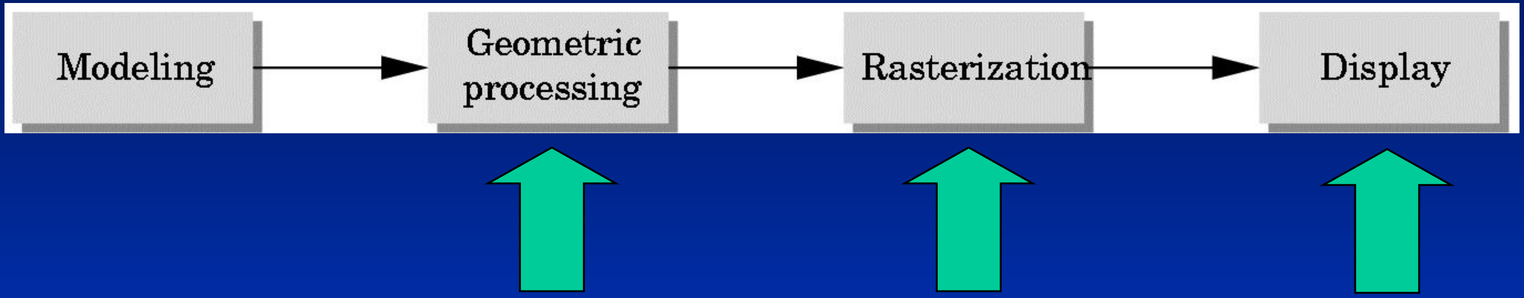
$$f(x,y) = (x - x_0)^2 + (y - y_0)^2 - r^2$$

- Remember the key idea is that, ONLY the sign matters!
  - If f(x,y)=0, then (x,y) is on the circle
  - If f(x,y)>0, then (x,y) is outside the circle
  - If f(x,y)<0, then (x,y) is inside the circle
- Equations for ellipses?
- The key message: the slope is controllable!!!

Scan Conversion

- At this point in the pipeline, we have only polygons and line segments. Render!
- To render, convert to pixels (“fragments”) with integer screen coordinates (ix, iy), depth, and color
- Send fragments into fragment-processing pipeline

Graphics Rendering Pipeline



- Geometric processing: normalization, clipping, hidden surface removal, lighting, projection (*front end*)
- Rasterization or scan conversion, including texture mapping (*back end*)
- Fragment processing and display

Geometric Processing

- **Front-end** processing steps (3D floating point; may be done on the CPU)
  - Evaluators (converting curved surfaces to polygons)
  - Normalization (modeling transformation, convert to world coordinates)
  - Projection (convert to screen coordinates)
  - Hidden-surface removal (object space)
  - Computing texture coordinates
  - Computing vertex normals
  - Lighting (assign vertex colors)
  - Clipping
  - Perspective division
  - Backface culling

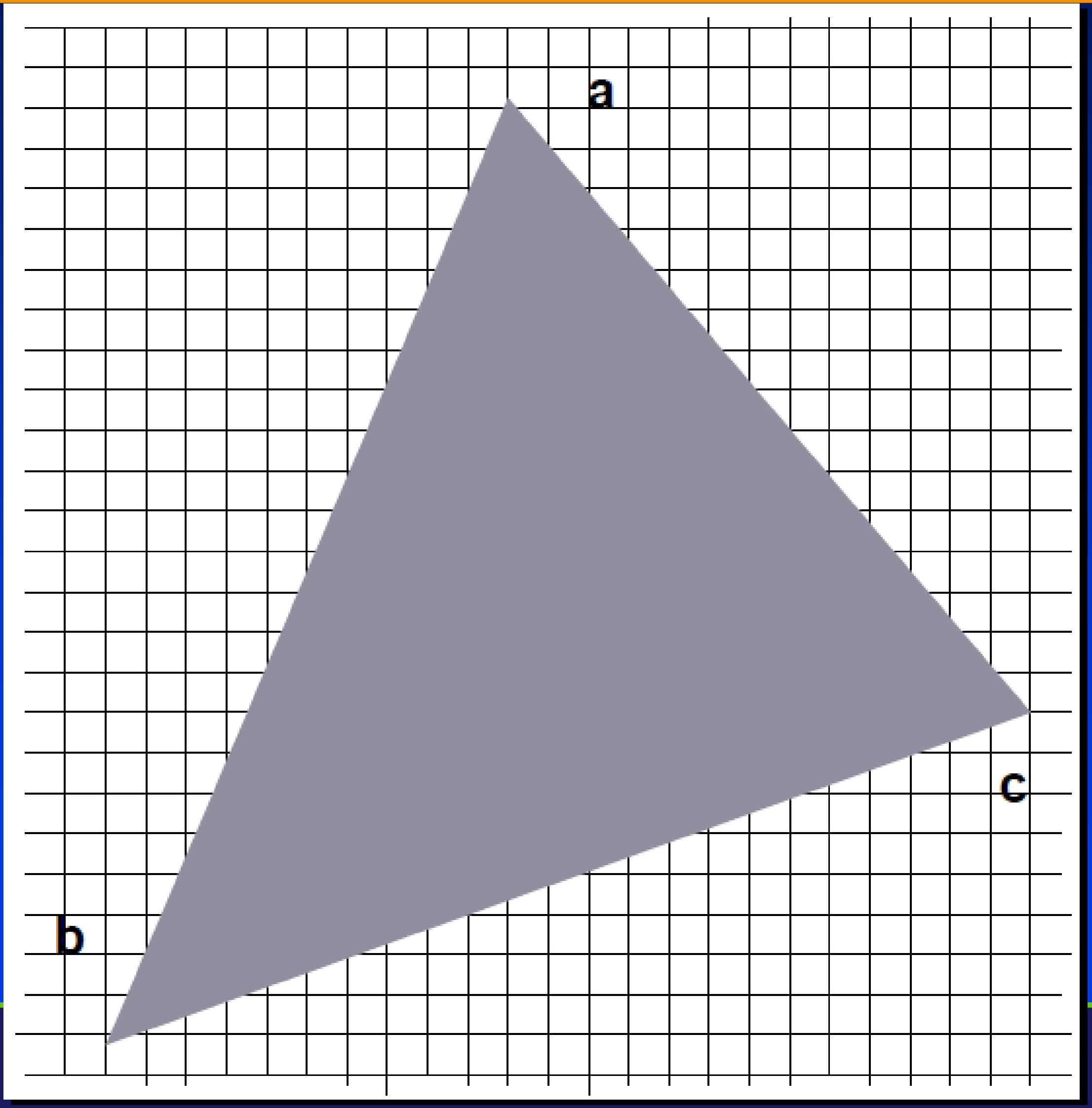
Rasterization

- **Back-end** processing works on 2D objects in screen coordinates
- Processing includes
  - Scan conversion of primitives including shading
  - Texture mapping
  - Fog
  - Scissors test
  - Alpha test
  - Stencil test
  - Depth-buffer test
  - Other fragment operations: blending, dithering, logical operations

Scan Conversion

- The earlier task allows us to draw line segments, polylines, curves, is it sufficient for 2D graphics?
- What are still missing for the rasterization task?
- Wireframe geometry and display is NOT enough
- We must have drawing routines to support the solid-shaded appearance

Scan Conversion



Simple Algorithms

- We start from a simple triangle T: (x1,y1), (x2,y2), and (x3,y3)
- The task is to find all pixels inside T
- Naïve algorithm (the worst algorithm)
  - For each pixel p do
  - If p is inside T, then draw-point(p) end if
  - End for

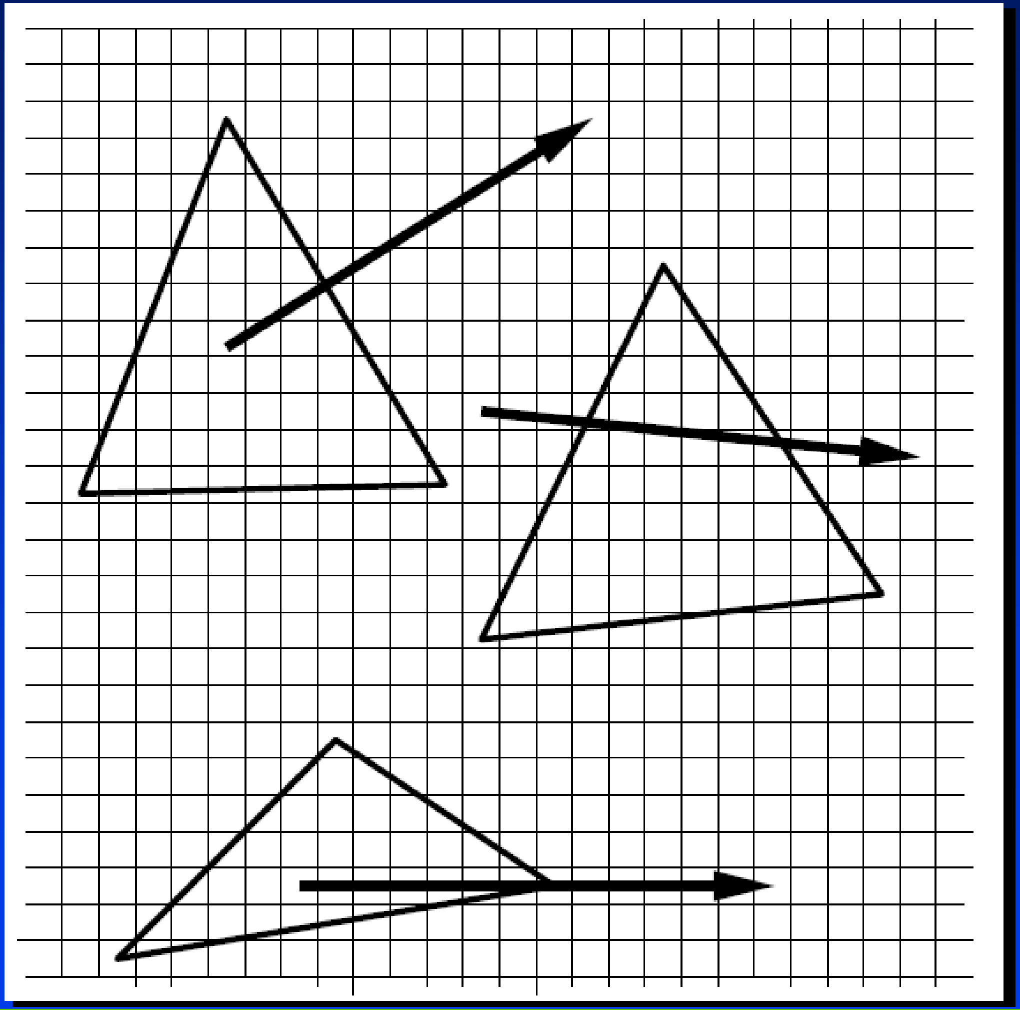
Slight Improvement

- We start from a simple triangle T: (x1,y1), (x2,y2), and (x3,y3)
- We compute its bounding box B first
  - For each pixel p that is inside B do
  - If p is inside T, then draw-point(p) end if
  - End for
- Essentially, the scan conversion MUST solve this problem, given a T and a pixel (or point in general), can we determine: p is a part of T

Ray Casting (Ray Firing)

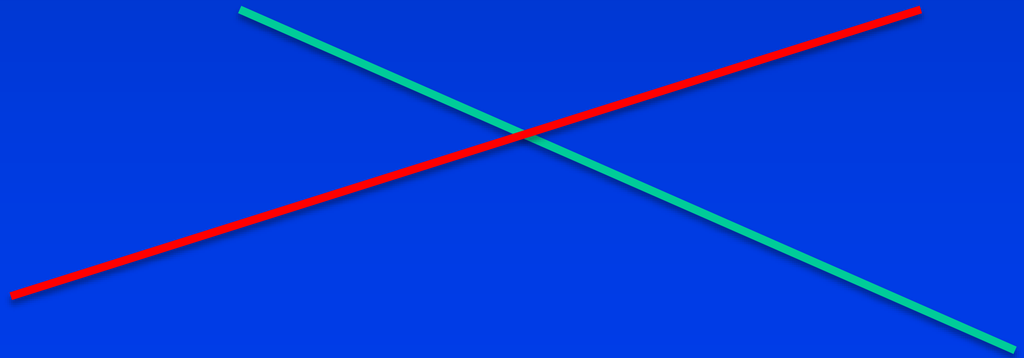
- We start from a simple triangle T: (x1,y1), (x2,y2), and (x3,y3) and a point
  - (1) draw a ray from p outward along any direction
  - (2) count the number of intersections of this ray with triangular boundaries for T
  - (3) If ODD, then p is inside T, otherwise, p is not a part of T
- Is this method correct?

Polygon Scan Conversion



Scan Conversion

- What happens if the ray pass through a vertex of a simple triangle T: (x1,y1), (x2,y2), and (x3,y3)
- How do you actually count the number of intersections with a triangular boundary?
- How do you actually compute the intersection?



Computing Intersections

- Mathematically speaking: $f(x,y)=0$; $g(x,y)=0$, simple solve them for possible solutions
- In reality (computer graphics), we don't really do the above way!
- Why?
- How do we handle this in computer graphics?

Computing Intersections

- First, consider a boundary of a polygon, we do NOT use its explicit representation at all. Instead, we use f(x,y)=ax+by+c=0;
- Second, consider a ray geometry, once again, we do NOT use its explicit representation at all. Instead we are using its parametric representation: ray(p, v) = p + v*t, where t is a spatial parameter, ray(p, v) works for (x,y) simultaneously!

Computing Intersections

- Parametric equation

$$x(t) = x_0 + t(x_1 - x_0)$$

$$y(t) = y_0 + t(y_1 - y_0)$$

- Vector expression

$$\mathrm{p}(t) = \mathrm{p}_0 + t(\mathrm{p}_1 - \mathrm{p}_0)$$

$$\mathrm{p}(t) = (1 - t)\mathrm{p}_0 + t\mathrm{p}_1$$

- The parameter is between 0 and 1 to describe a line segment, the ray can be expressed in the same way

Computing Intersections

- Combine the two equations together (one is the implicit equation, another one is the parametric equation), f(ray(p,v))=0, where t is the ONLY parameter (to be solved)
- What is the geometric meaning of t?
- We are going to have more mathematically rigorous process on the parametric representation and its power and potential later in this course!

Scan Conversion

- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3) and a point
- Consider the edge (v1v2) and formulate the implicit expression for this line

$$l_{1,2}(x,y) = a_{1,2}x + b_{1,2}y + c_{1,2}$$

- Pick a sign so that the evaluation of v3 is negative!
- This defines a half-plane

$$h_{1,2} = \{(x,y) : l_{1,2}(x,y) <= 0\}$$

Scan Conversion

- We start from a simple triangle T: v1=(x1,y1), v2=(x2,y2), and v3=(x3,y3) and a point
- Repeat the similar process for two other edges (v1v2) and (v2v3)

$$T = h_{1,2} \cap h_{1,3} \cap h_{2,3}$$

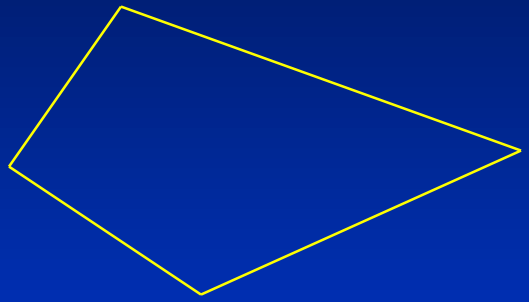
- It is equivalent to say, a pixel (point) is a part of a triangle if this point belongs to three half-planes simultaneously

$$l_{1,2}(p_x, p_y) <= 0$$

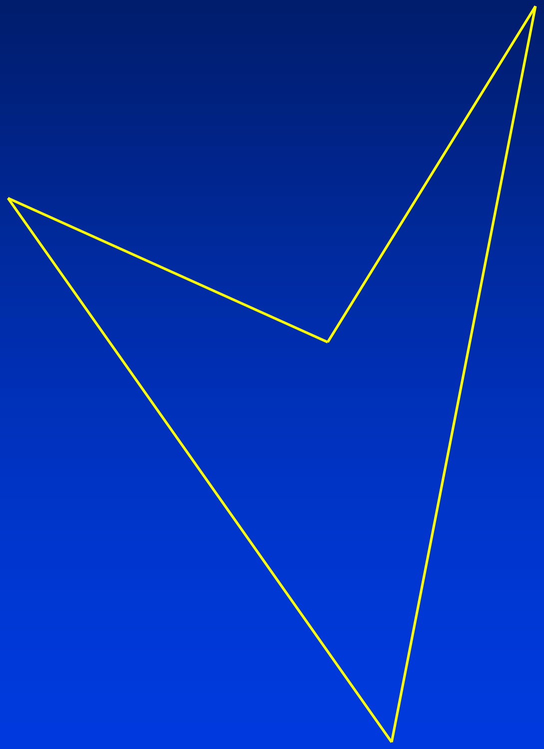
$$l_{1,3}(p_x, p_y) <= 0$$

$$l_{2,3}(p_x, p_y) <= 0$$

- What about Concave polygon?



Convex



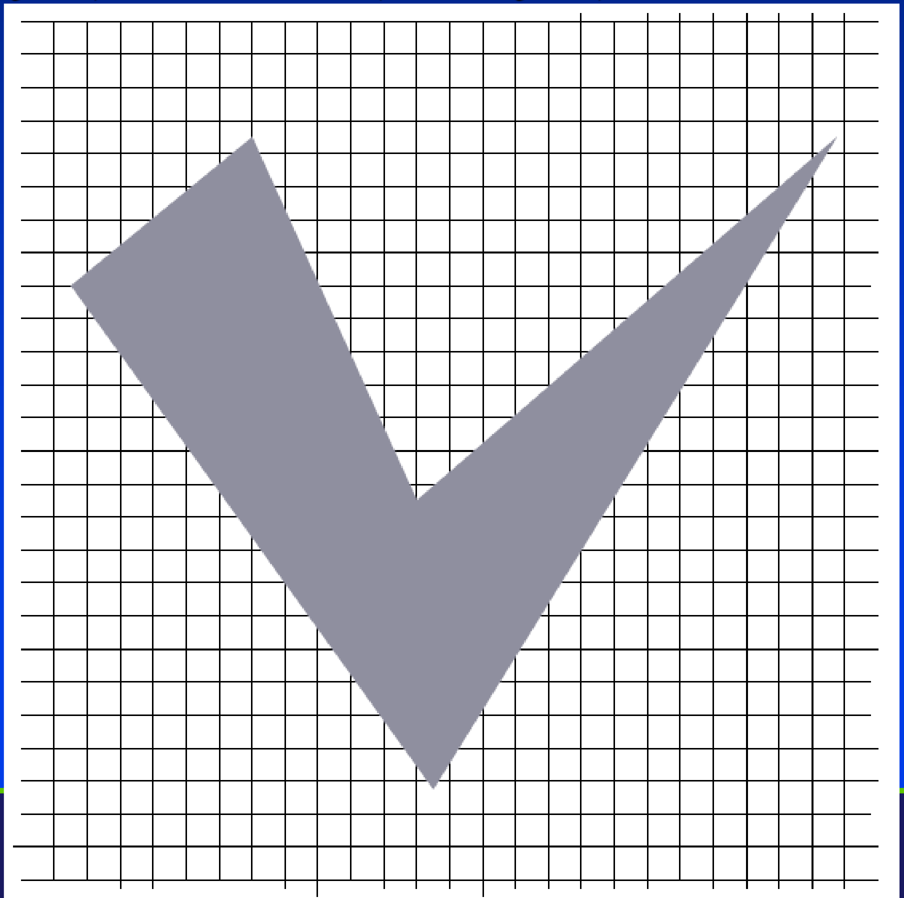
Not Convex

Convex

- A polygon is convex if…
  - A line segment connecting any two points on the polygon is contained in the polygon.
  - If you can wrap a rubber band around the polygon and touch all of the sides, the polygon is convex

Concave Polygon

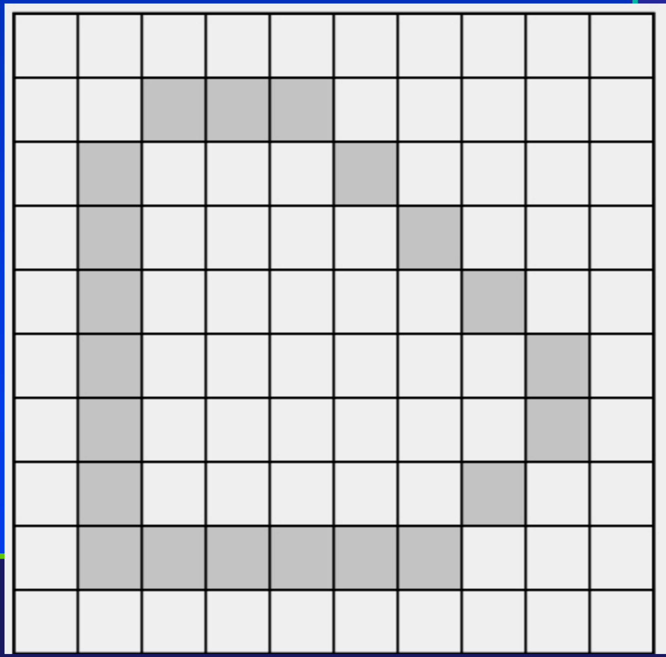
- We now consider a concave polygon T: (x1,y1), (x2,y2), (x3,y3), ……. (xn, yn)



Scan-Converting a Polygon

- General approach: any ideas?
- One idea: *flood fill*
  - – Draw polygon edges
  - – Pick a point (x,y) inside and **flood fill** with DFS

```
flood_fill(x,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);
} }
```



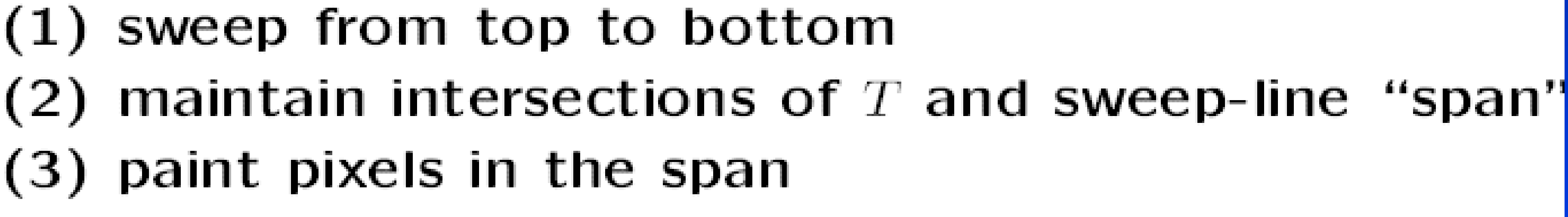
Sweeping Lines

- Our observation – spatial coherence

If $p \in T$, then neighboring pixels are probably in the triangle, too

(Coherence)

- Idea



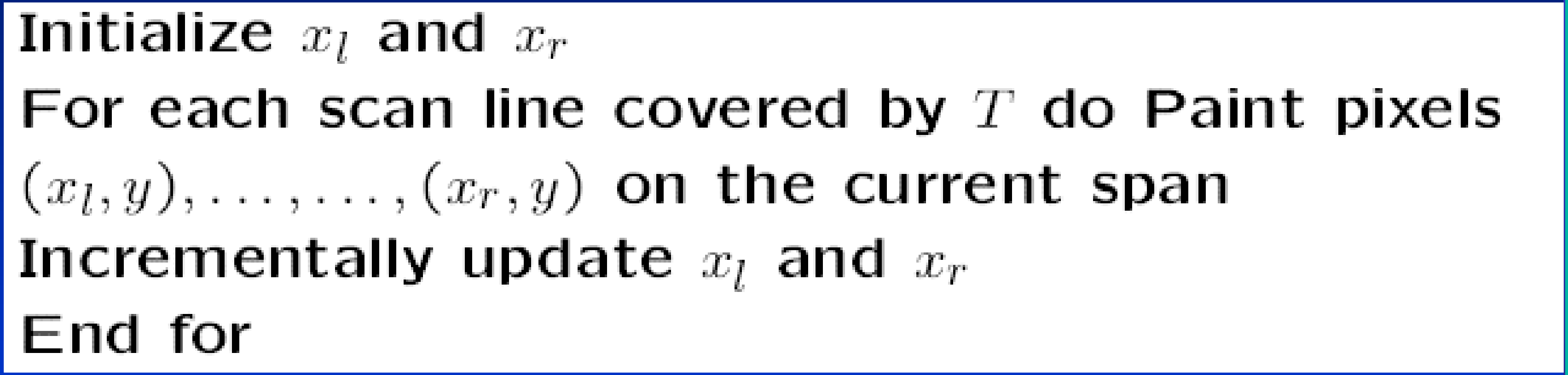
(1) sweep from top to bottom

(2) maintain intersections of T and sweep-line “span”

(3) paint pixels in the span

Sweep-line Algorithm

- Algorithm



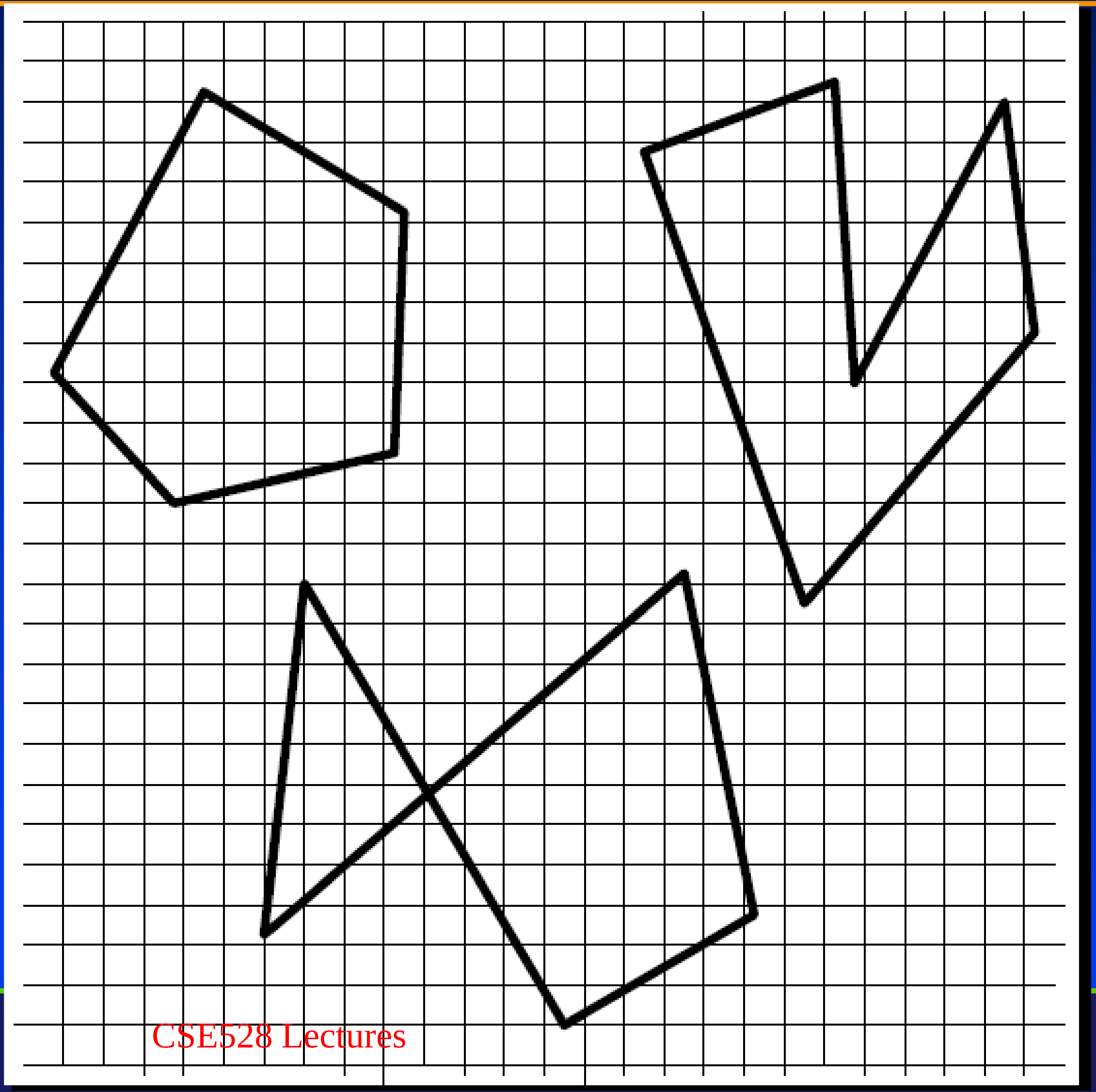
```
Initialize $x_l$ and $x_r$
For each scan line covered by $T$ do Paint pixels
$(x_l, y), \ldots, \ldots, (x_r, y)$ on the current span
Incrementally update $x_l$ and $x_r$
End for
```

- Question:

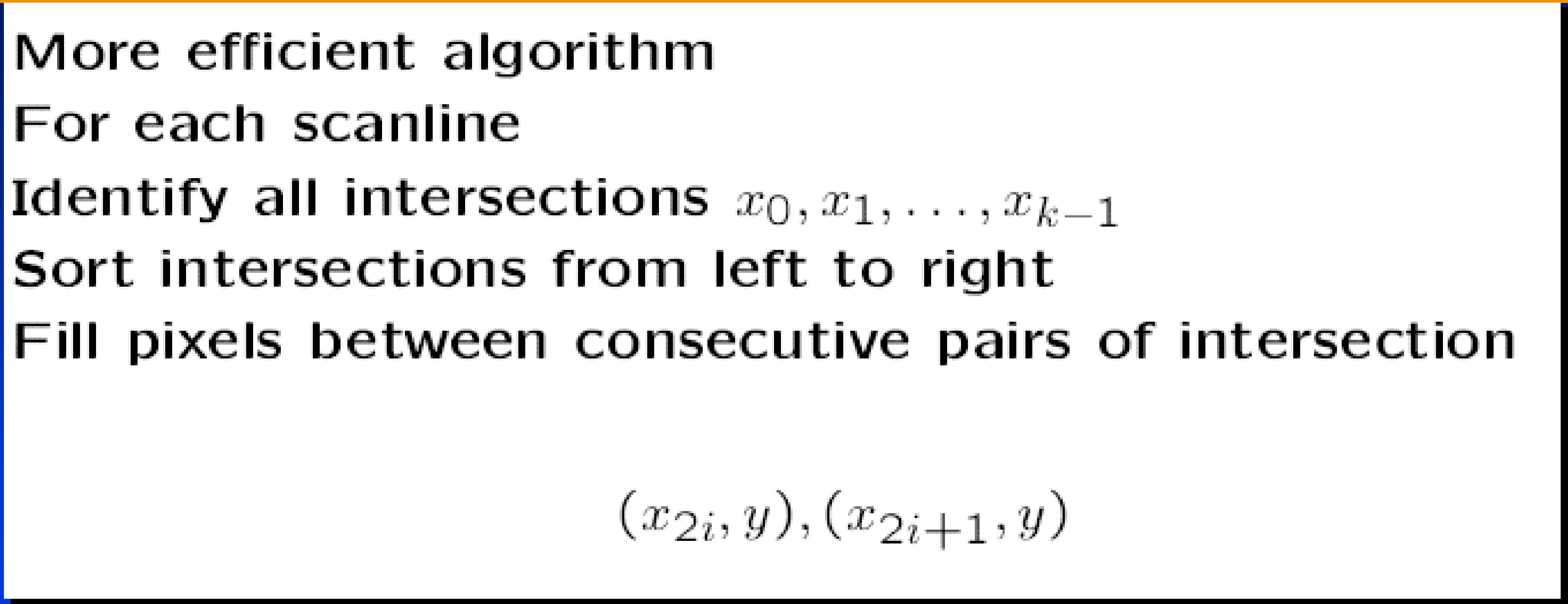
how do we update x_l and x_r ?

- Answer: please recall our line-drawing algorithm!

Polygon Classification



Scan Conversion



More efficient algorithm

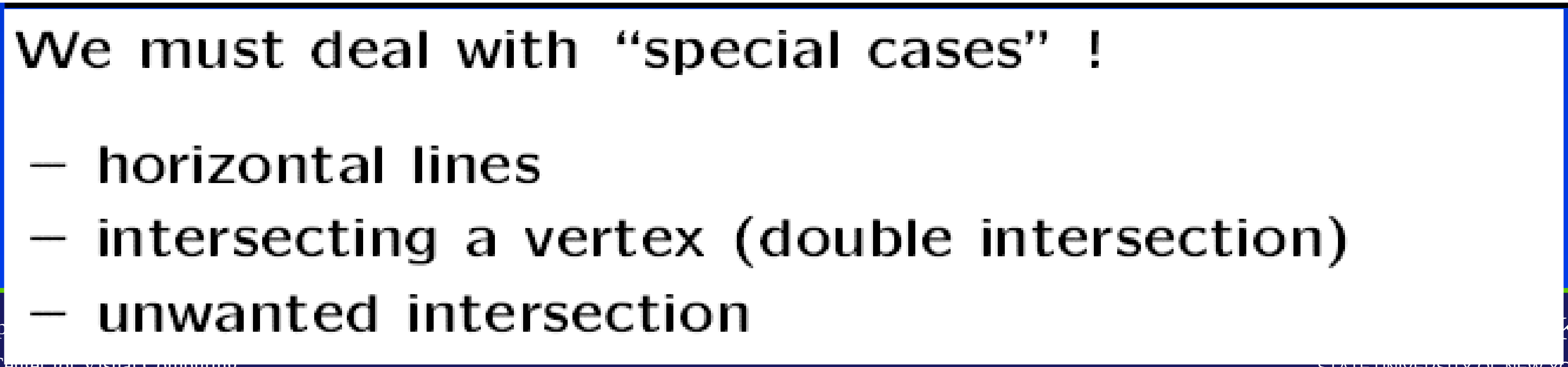
For each scanline

Identify all intersections $x_0, x_1, \ldots, x_{k-1}$

Sort intersections from left to right

Fill pixels between consecutive pairs of intersection

$$(x_{2i}, y), (x_{2i+1}, y)$$



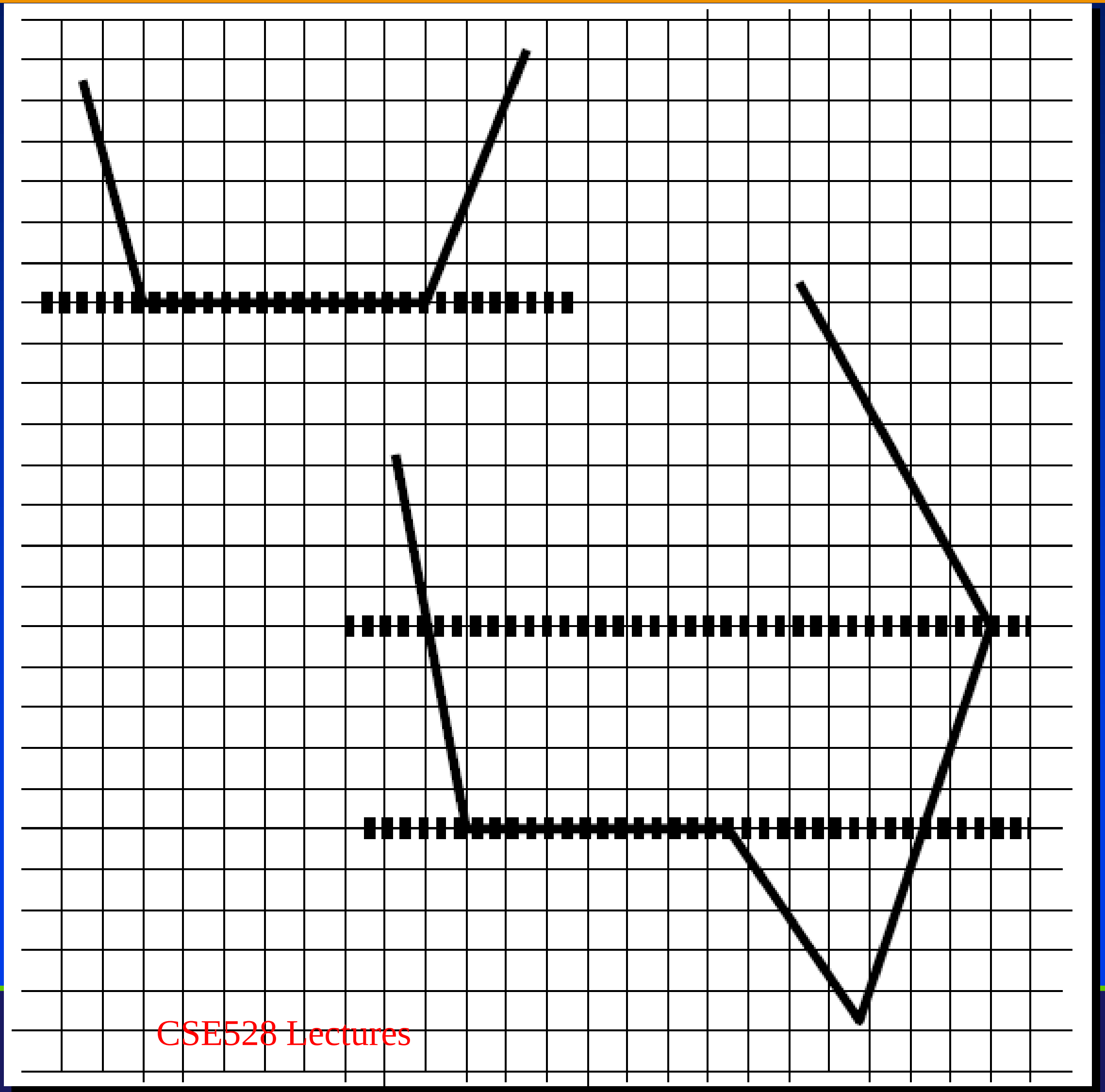
We must deal with "special cases" !

- horizontal lines
- intersecting a vertex (double intersection)
- unwanted intersection

Scan Conversion

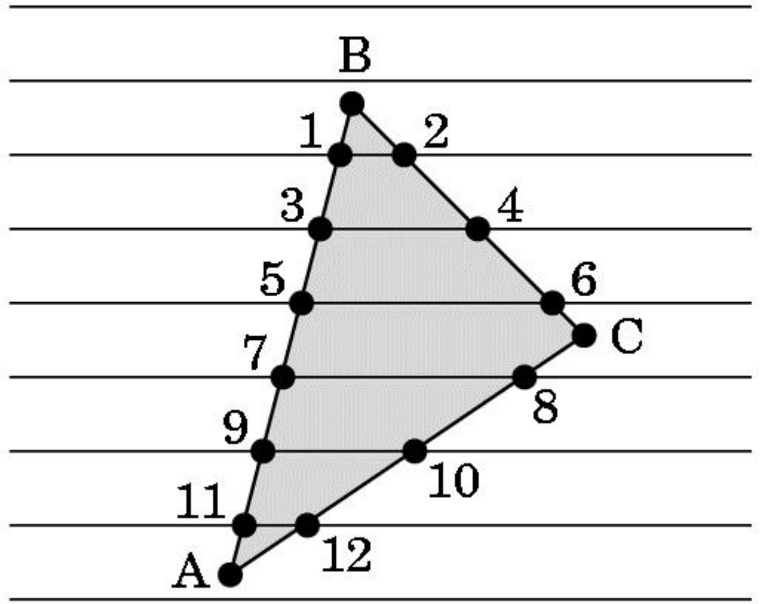
- We must speed up the edge intersection detection
- Data structure for efficient implementation
  - A sorted edge table
  - The active edge list
  - From bottom to the top
- Practical polygon scan conversion – based on polygon triangulation
- Extremely easy to handle for convex polygons
- Triangles are often particularly nice to work with because they are always planar and simple

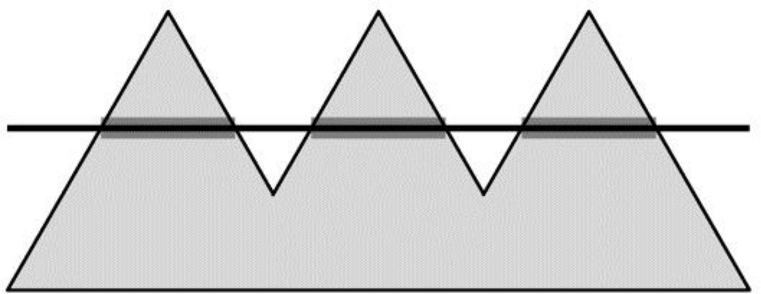
Special Cases



Scan-Line Approach

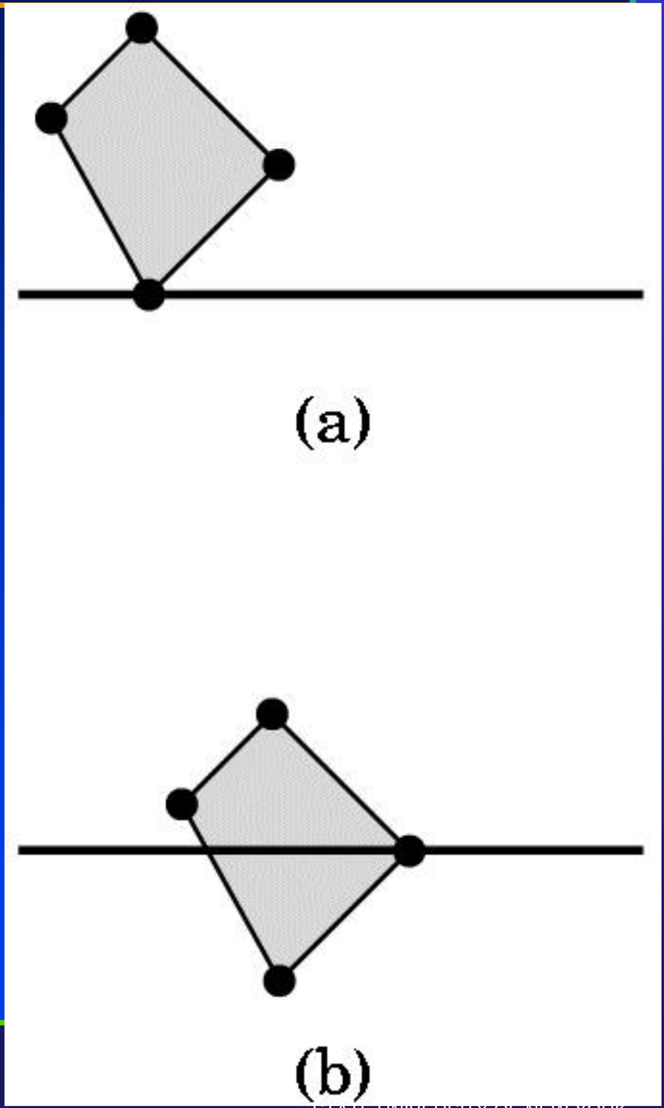
- More efficient way: use a scan-line rasterization algorithm
- For each y value, compute x intersections, fill according to winding rule
- How to compute intersection points?
- How to handle shading?
- Some hardware can handle multiple scanlines in parallel





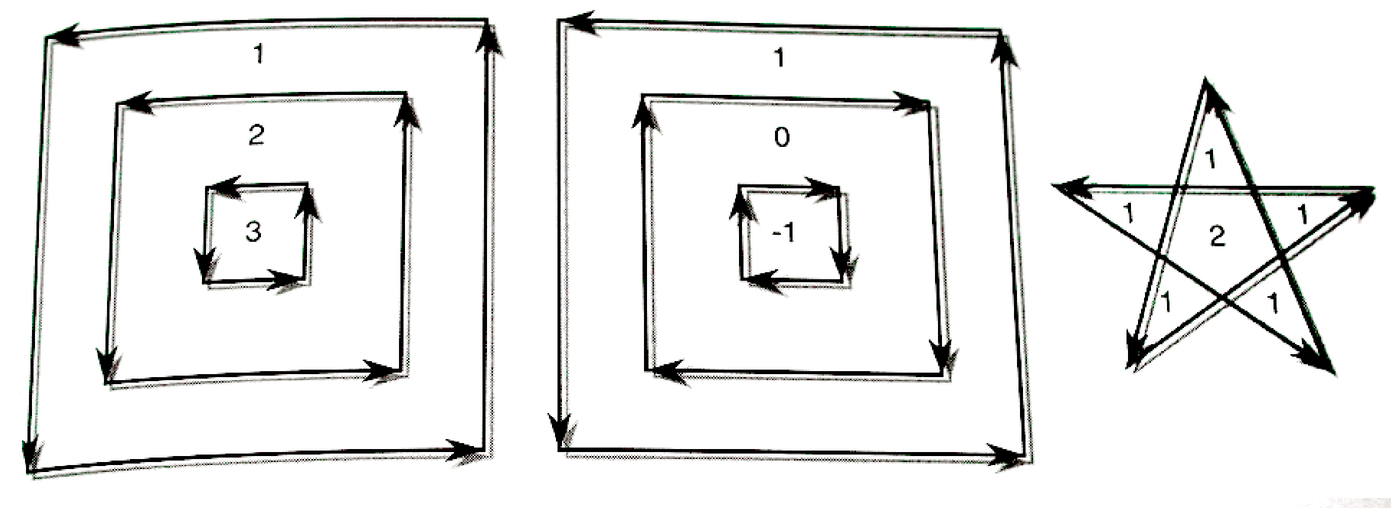
Singularities (Special Cases)

- If a vertex lies on a scanline, does that count as 0, 1, or 2 crossings?
- How to handle singularities?
- One approach: don’t allow. *Perturb* vertex coordinates
- OpenGL’s approach: place pixel centers half way between integers (e.g., 3.5, 7.5), so scanlines never hit vertices



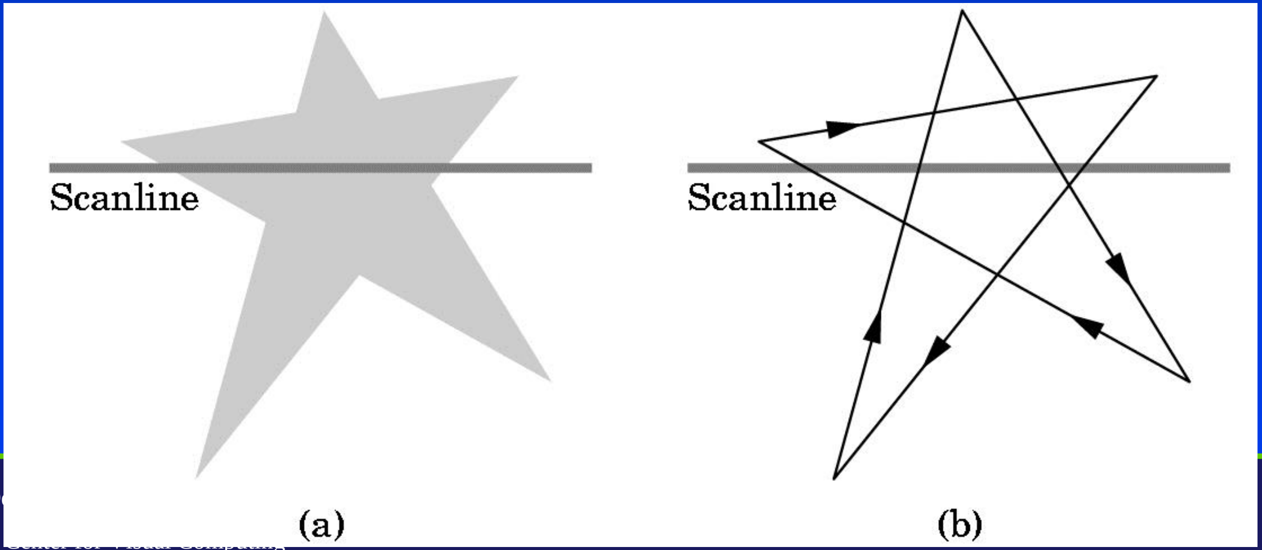
Winding Test

- Most common way to tell if a point is in a polygon: the winding test.
  - Define “winding number” w for a point: signed number of revolutions around the point when traversing boundary of polygon once
  - When is a point “inside” the polygon?

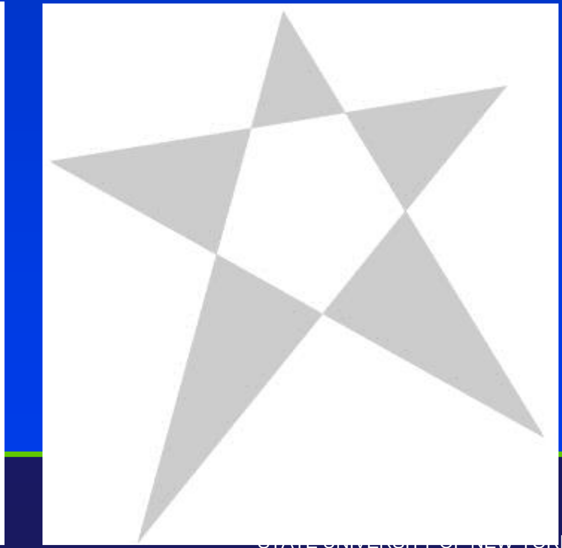


Rasterizing Polygons (Scan Conversion)

- Polygons may be or may not be simple, convex, or even flat. How to render them?
- The most critical thing is to perform inside-outside testing: how to tell if a point is in a polygon?

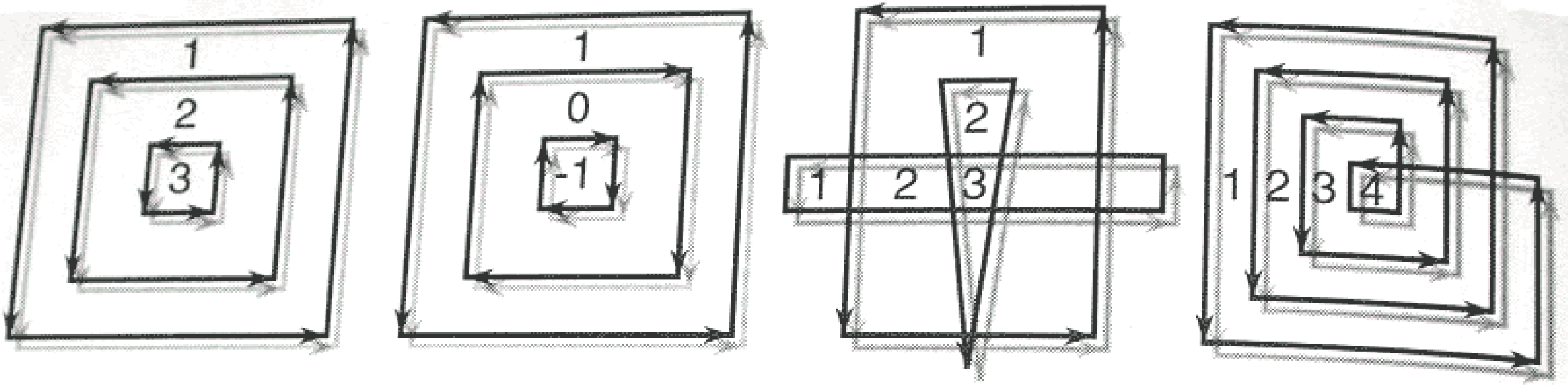


(a) (b)

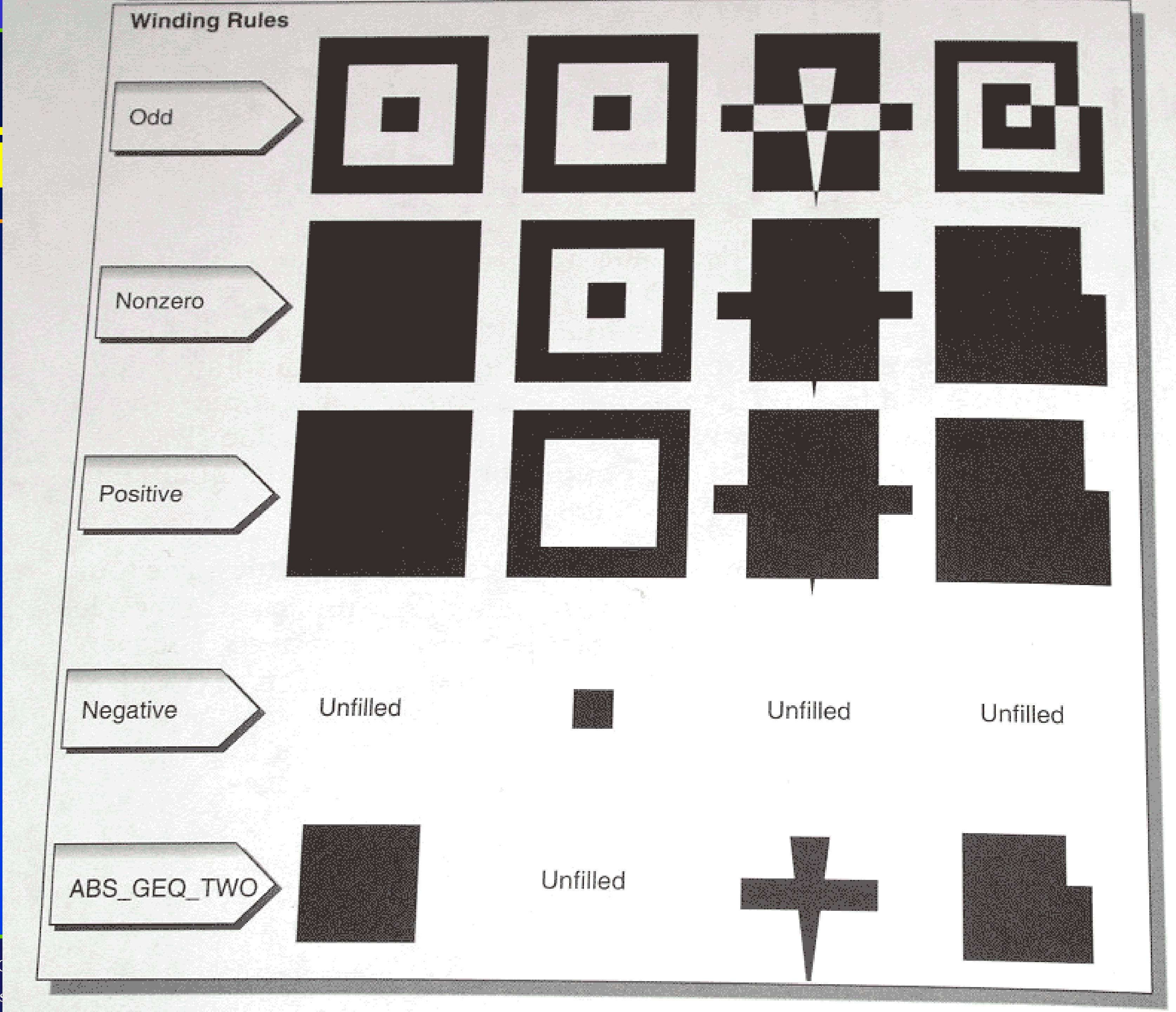


OpenGL and Concave Polygons

- OpenGL guarantees correct rendering only for simple, convex, planar polygons
- OpenGL tessellates concave polygons
- Tessellation depends on winding rule you tell OpenGL to use: Odd, Nonzero, Pos, Neg, ABS_GEQ_TWO



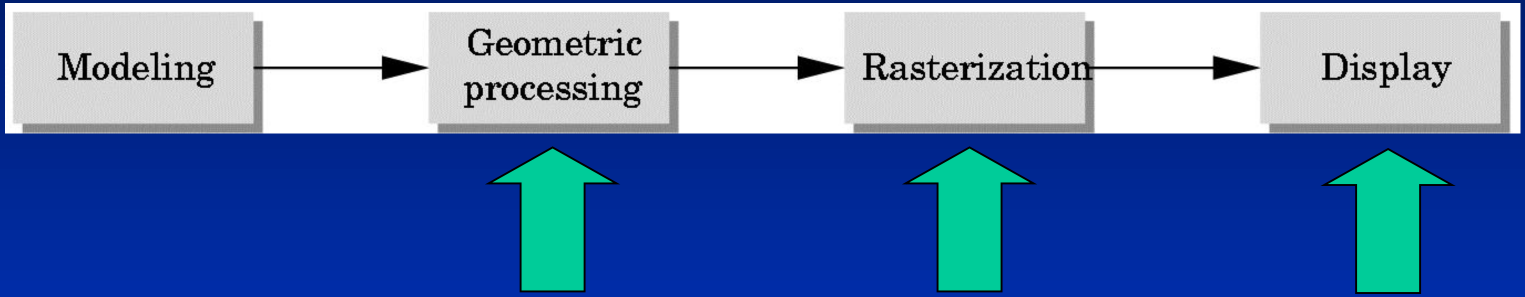
Wi


Winding Rules
Odd
Nonzero
Positive
Negative
Unfilled
Unfilled
Unfilled
ABS_GEQ_TWO
Unfilled

Geometry

Transformations → Lighting → Projection → Clipping

Rendering Pipeline



- Geometric processing: normalization, clipping, hidden surface removal, lighting, projection (*front end*)
- Rasterization or scan conversion, including texture mapping (*back-end*)
- Fragment processing and display

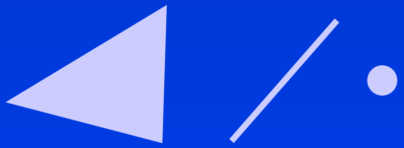
From Models to Rasterization

Application → Geometry → Rasterization

3D Model →

- meshing
- decimation
- collision detection
- animation
- …

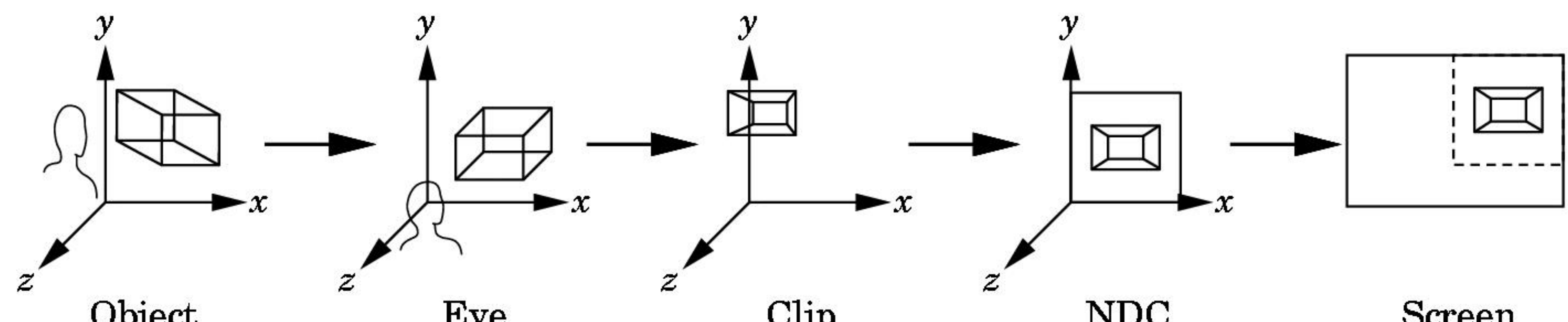
→ Rendering primitives



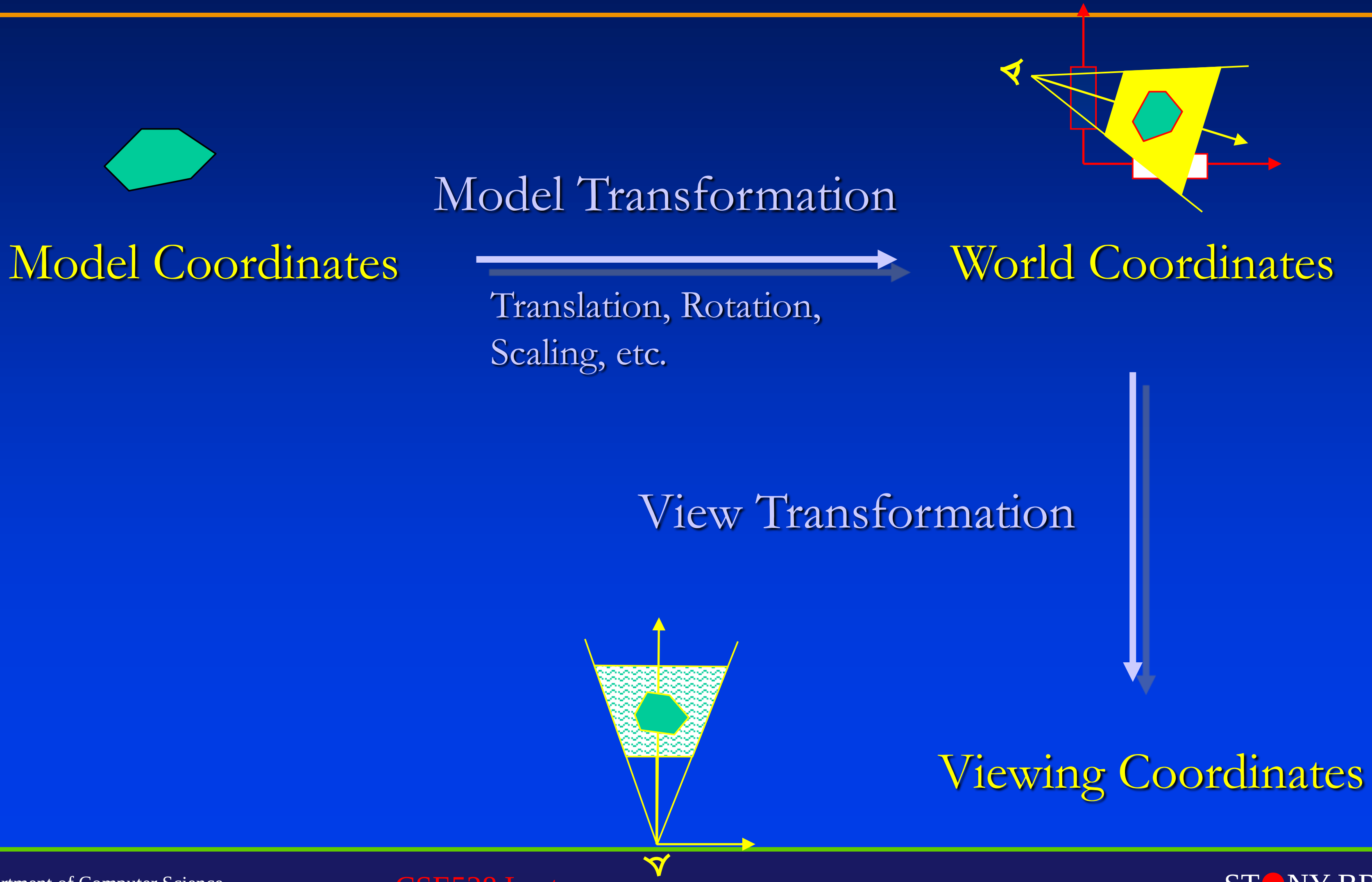
Software-based processing / modifications

Geometric Transformations

- Five coordinate systems of interest:
  - Object coordinates
  - Eye (world) coordinates [after modeling transform, viewer at the origin]
  - Clip coordinates [after projection]
  - Normalized device coordinates [after ÷w]
  - Window (screen) coordinates [scale to screensize]



Geometry: Transformations



Model Coordinates

Model Transformation

Translation, Rotation, Scaling, etc.

World Coordinates

View Transformation

Viewing Coordinates

Geometry: Projection

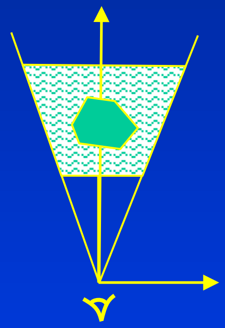
Viewing Coordinates

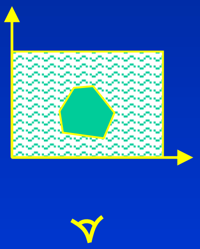
Normalization

Perspective/

Parallel

Virtual Device Coordinates



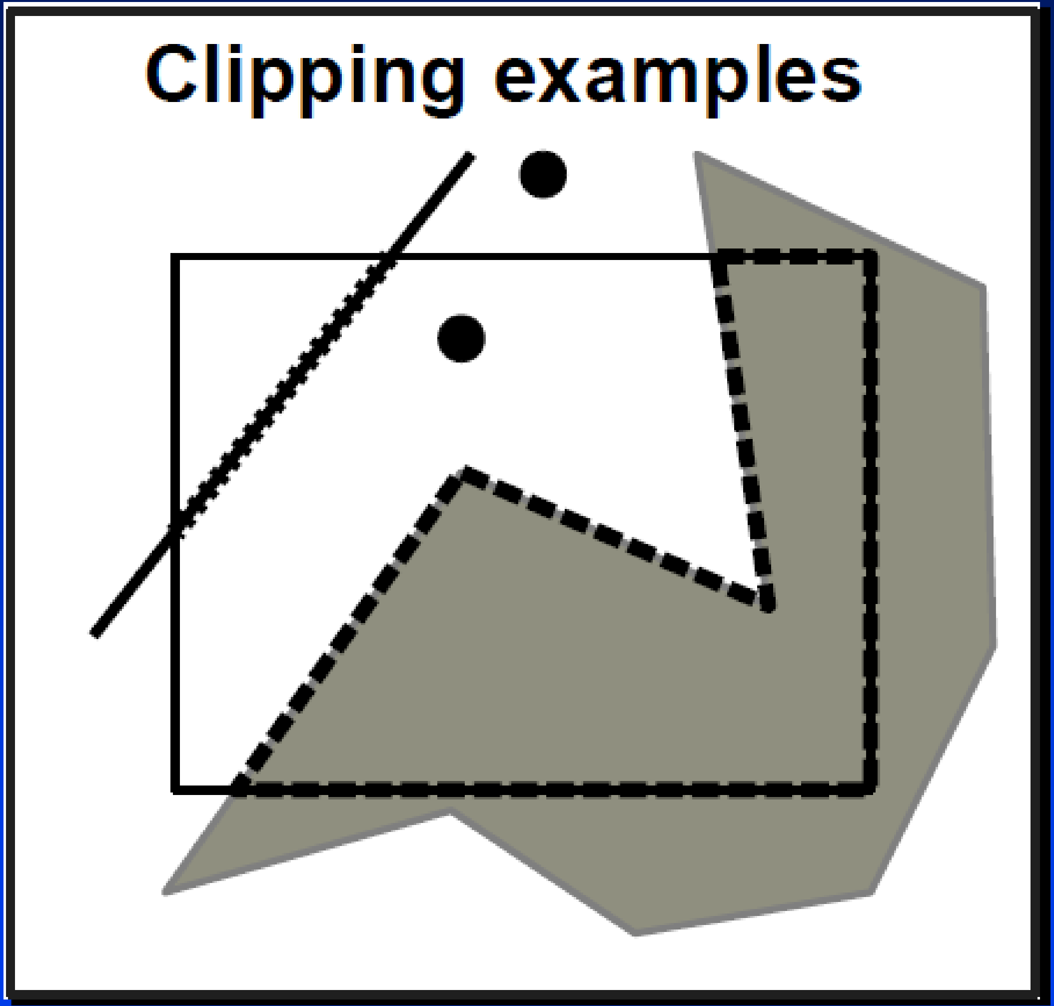


Computer Graphics: Geometric Clipping

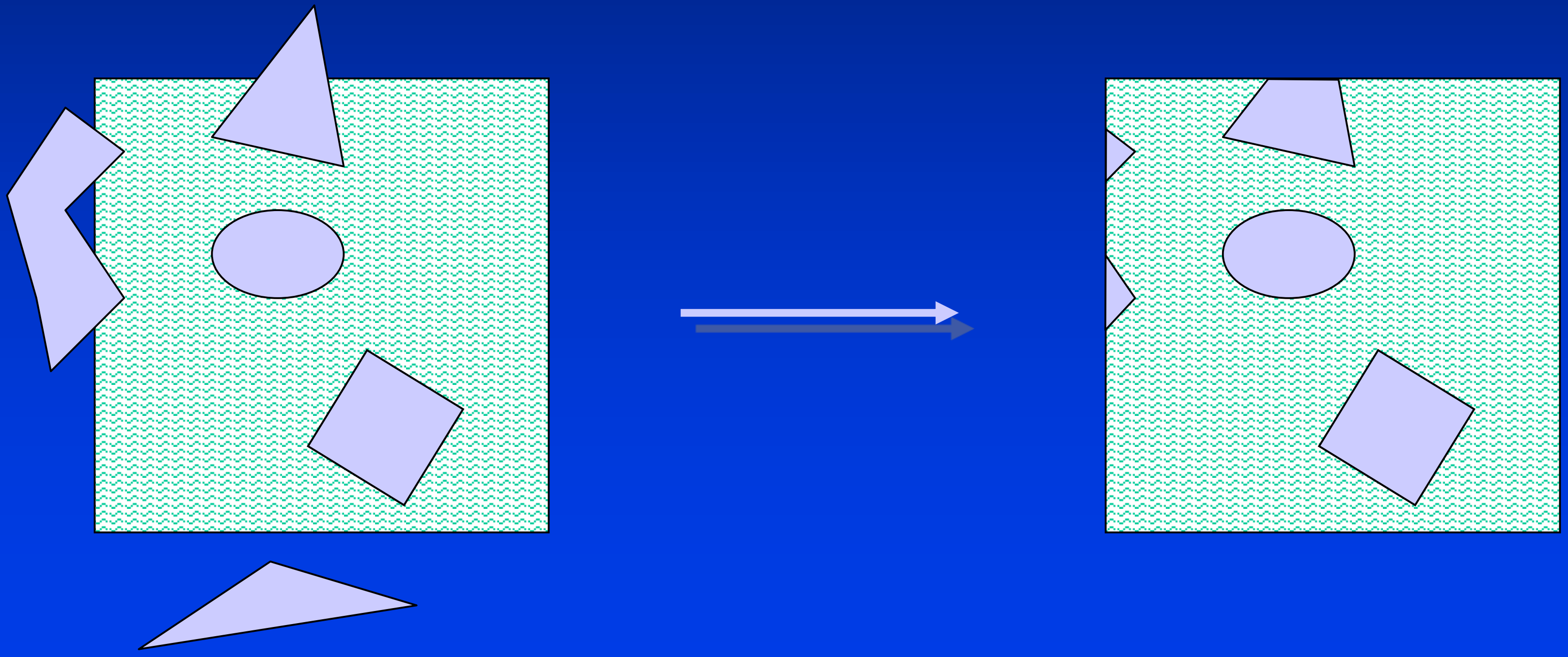
How Do We Define a Window?

- Window
- Viewport

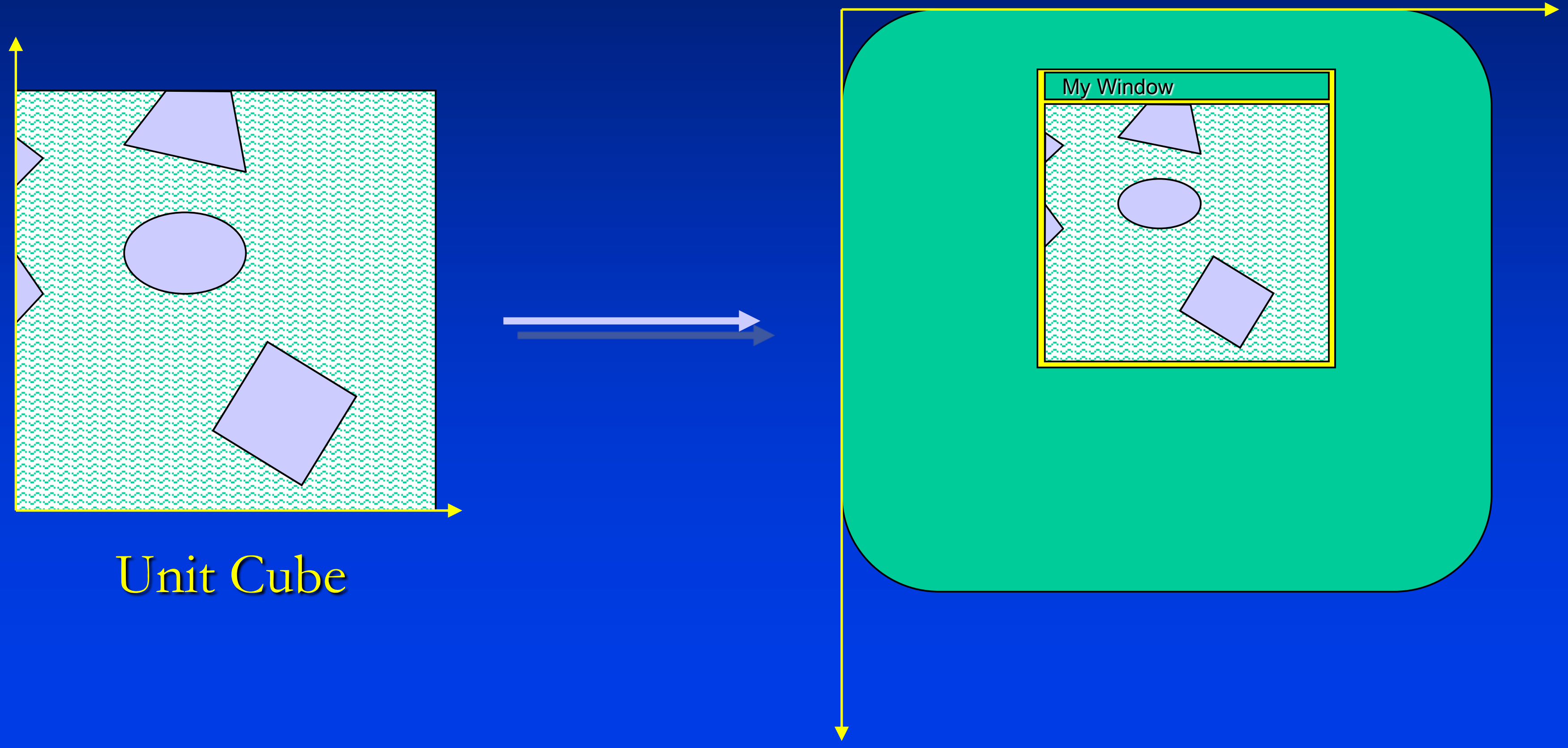
2D Clipping



Geometry: Clipping

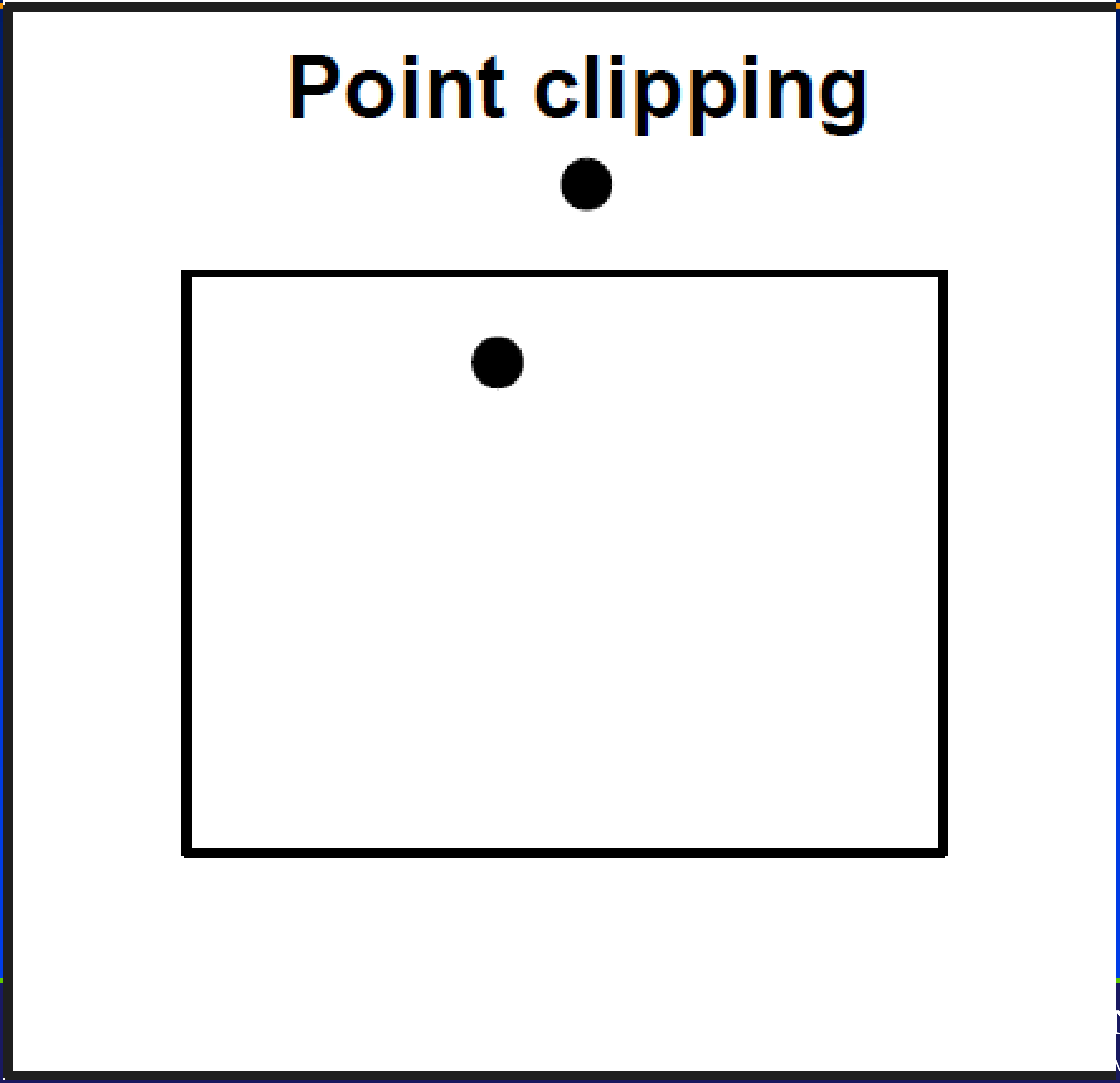


Geometry: Device Coordinates



2D Clipping

- Points
- Lines
- Polygons



2D Clipping

- How to define a window:

x_l

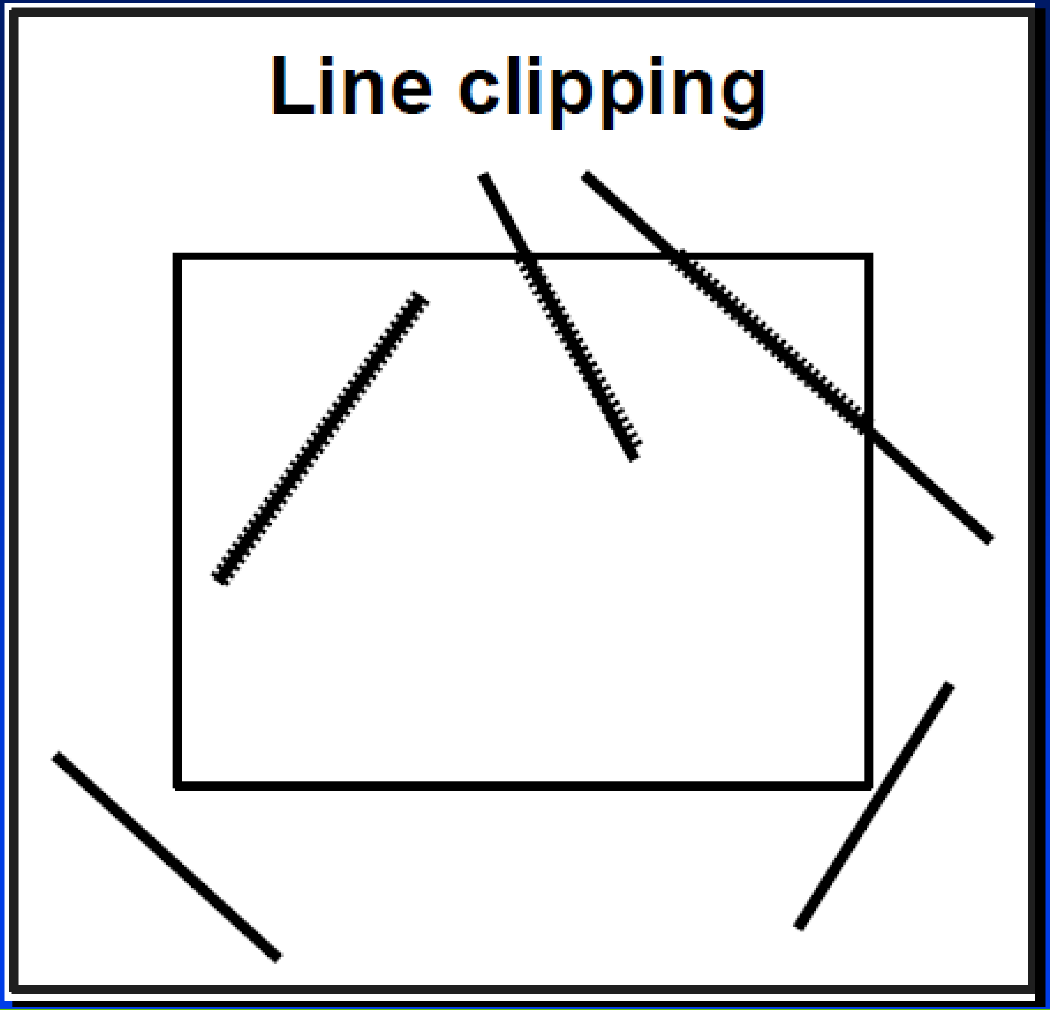
x_r

y_b

y_t

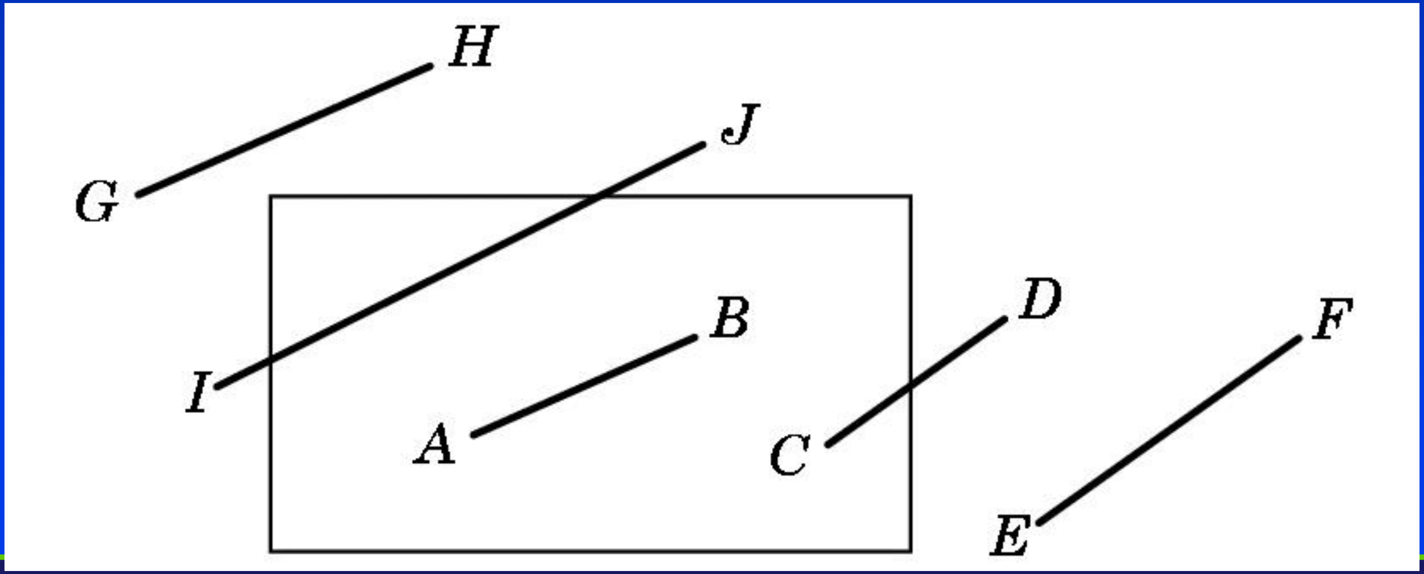
- Point clipping is trivial
- However, pay attention to (1) the homogeneous coordinates; (2) equations of lines

Line Clipping



Line-Segment Clipping Operations

- Clipping may happen in multiple places in the pipeline (e.g., early trivial accept/reject)
- After projection, have lines in plane, with rectangle to clip against



Line Clipping

- Line clipping operations should comprise the following cases
  - Totally plotted
  - Partially plotted
  - NOT plotted at all
- Far from being trivial – even though neither of two vertices is within the window, certain part of the line segment may be still within the window!
- There are many different techniques for line clipping in 2D
- Two fundamental issues: (1) line equations; (2) intersection computation

The Fundamental Operation

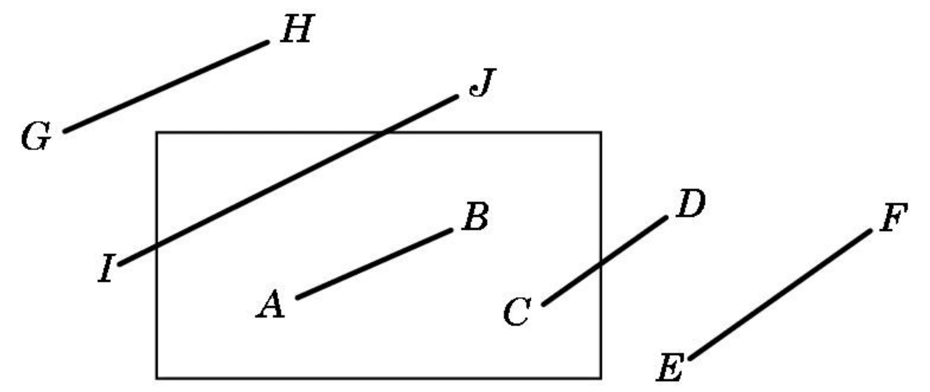
- In geometric clipping, the most fundamental operation is how to compute line-line intersection: (1) whether two lines are intersecting or NOT; (2) if they Do intersect, can you please find such intersection point(s)?
- Equations for a line: (1) explicit representation; (2) implicit representation; or (2) parametric representation?

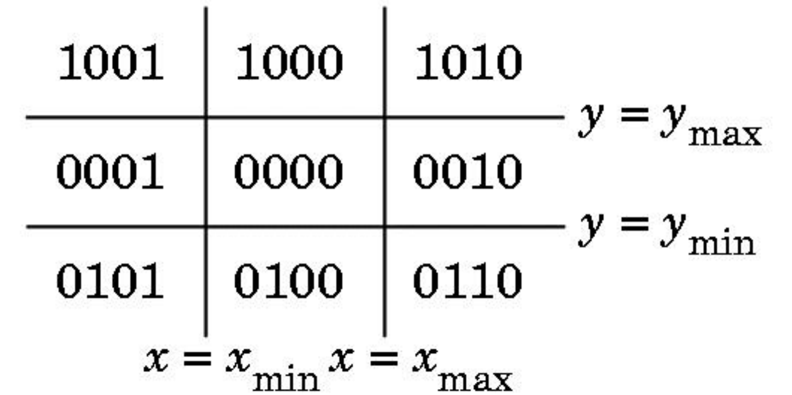
Clipping a Line Segment Against x_{min}

- Given a line segment from (x_1,y_1) to (x_2,y_2), Compute $m=(y_2-y_1)/(x_2-x_1)$
- Line equation: $y = mx + h$ (explicit representation)
- $h = y_1 – m\ x_1$ (y intercept)
- Plug in x_{min} to get y
- Check if y is between y_1 and y_2.
- This might take a lot of floating-point operations. How to minimize the number of such operations?

Cohen-Sutherland Clipping

- For both end-points of a line segment compute a 4-bit *outcode* ($tbrl_1$, $tbrl_2$) depending on whether the current coordinates are outside the clip-rectangle side
- Some situations can be handled easily





1001	1000	1010
0001	0000	0010
0101	0100	0110

Cohen-Sutherland Conditions

- **Cases.**
  - 1. If $tbrl_1=tbrl_2=0$, simply **accept!**
  - 2. If one is zero, one nonzero, compute an intercept. If necessary compute another intercept. Then **accept.**
  - 3. If $tbrl_1$ & $tbrl_2 \neq 0$. If both outcodes are nonzero and the bitwise AND is nonzero, two endpoints lie on same outside side. **Simply reject!**
  - 3. If $tbrl_1$ & $rbrl_2 = 0$. If both outcodes are nonzero and the bitwise AND is zero, may or may not have to draw the line. Intersect with one of the window sides and check the result.

Cohen-Sutherland Results (Performance)

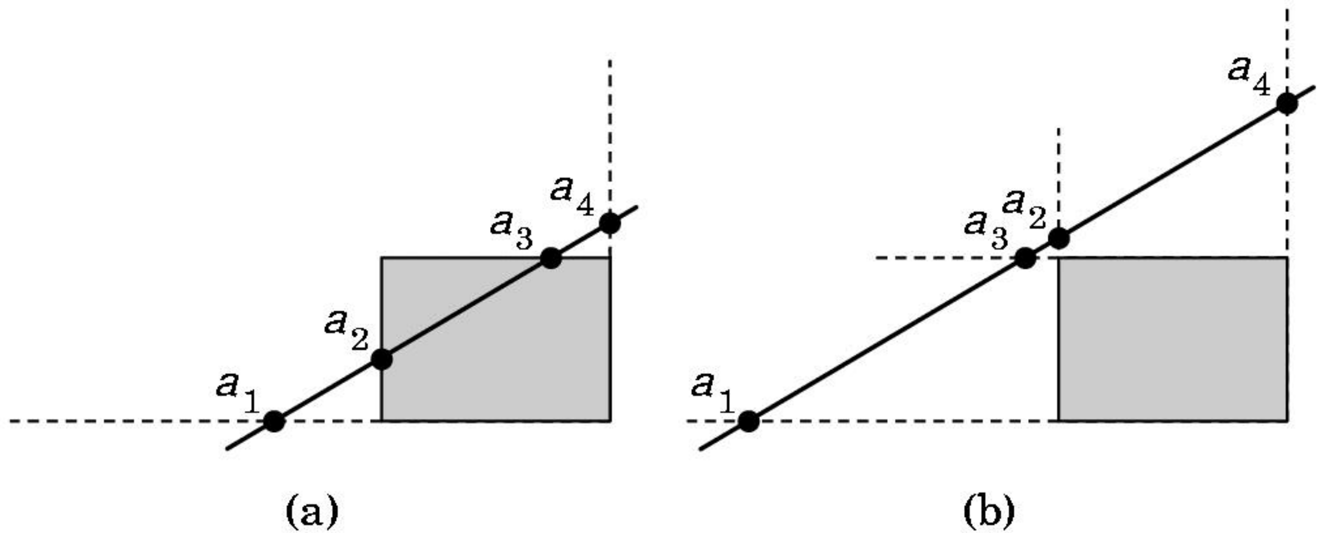
- In many cases, a few integer comparisons and Boolean operations suffice for simple reject or simple accept.
- This algorithm works best when there are many line segments, and most are clipped away
- But note that the y=mx+h form of equation for a line doesn't work for vertical lines (this is actually the limitation of explicit representation of a line)

Parametric Line Representation

- In computer graphics, a parametric representation is almost always used.
- Parametric representation of a line: $p(t) = (1-t)\,p_1 + t\,p_2$
  - Same form for horizontal and vertical lines
  - Parameter values from 0 to 1 are on the segment
  - Values < 0 off in one direction; >1 off in the other direction
  - Vector operations, can be generalized to higher dimensional geometry or general data representation

Liang-Barsky Clipping

- If line is horizontal or vertical, handle easily
- Else, compute four intersection parameters with four rectangle sides
- What if $0<a_1<a_2<a_3<a_4<1$?
- What if $0<a_1<a_3<a_2<a_4<1$?



(a)

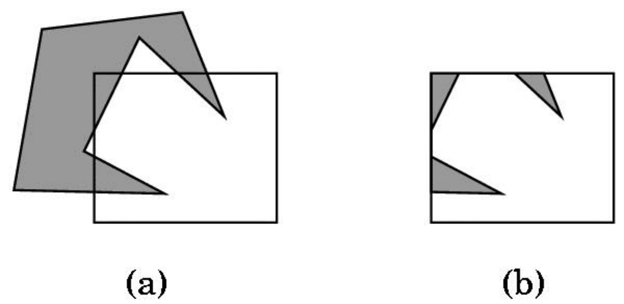
(b)

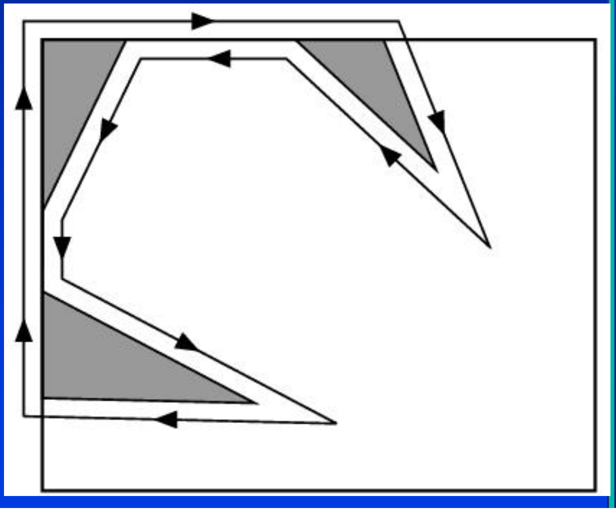
Computing Intersection Parameters

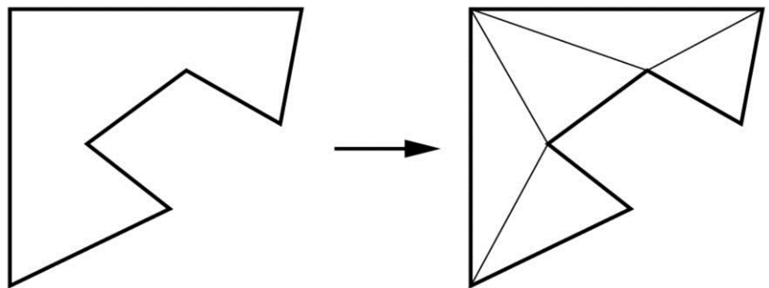
- Line-line intersection computation can be very costly
- Hold off on computing parameters as long as possibly (lazy computation); many lines can be rejected early
- Could compute $a=(y_{max}-y_1)/(y_2-y_1)$
- Can rewrite $a\ (y_2-y_1) = (y_{max}-y_1)$
- Perform work in integer operations by comparing $\mathbf{a\ (y_2-y_1)}$ instead of **a**

Polygon Clipping (Naïve Generalization)

- Clipping a polygon can result in lots of pieces
- Replacing one polygon with many may be a problem in the rendering pipeline
- Could treat result as one polygon: but this kind of polygon can cause other difficulties
- Some systems allow only convex polygons, which don’t have such problems (OpenGL has tessellate function in glu library)

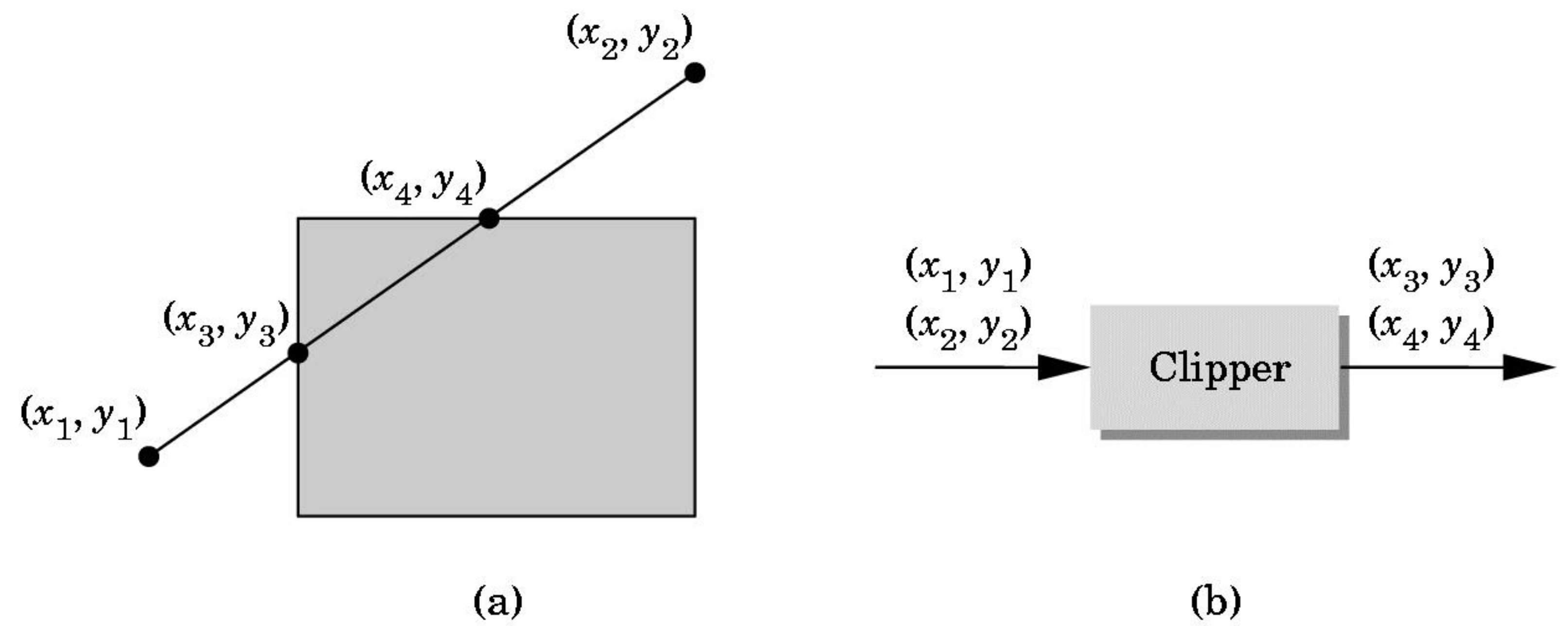






Sutherland-Hodgeman Polygon Clipping

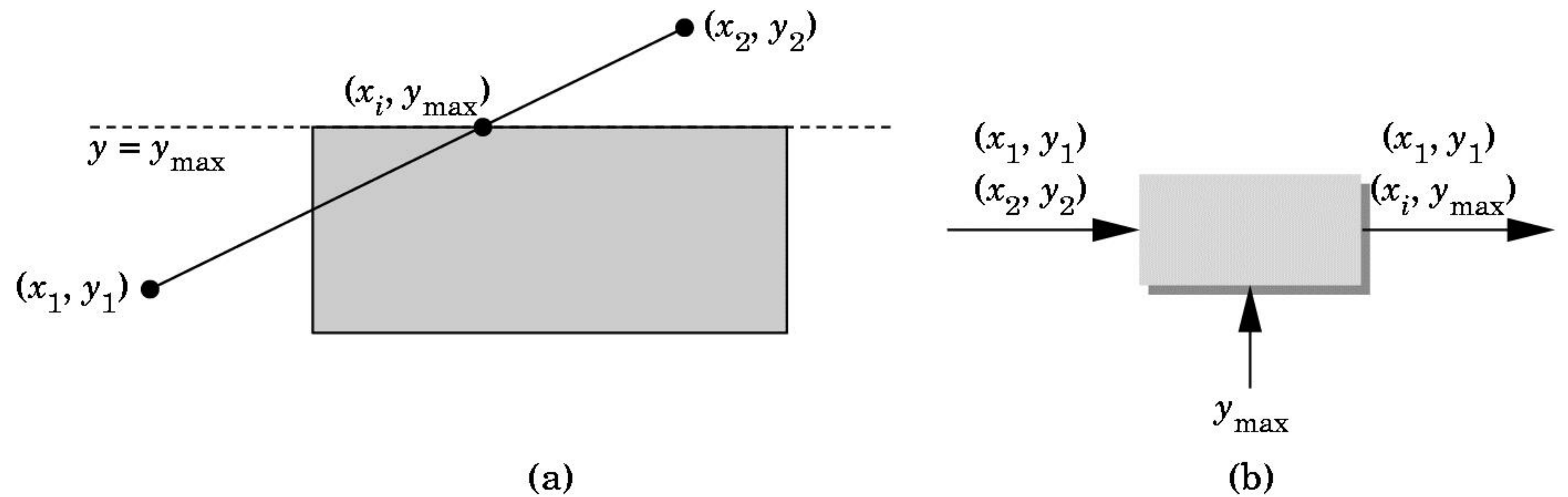
- Could clip each edge of polygon individually
- A more pipelined approach: clip polygon against each side of rectangle in turn (window boundary)
- Treat clipper as “black box” pipeline stage



(a) (b)

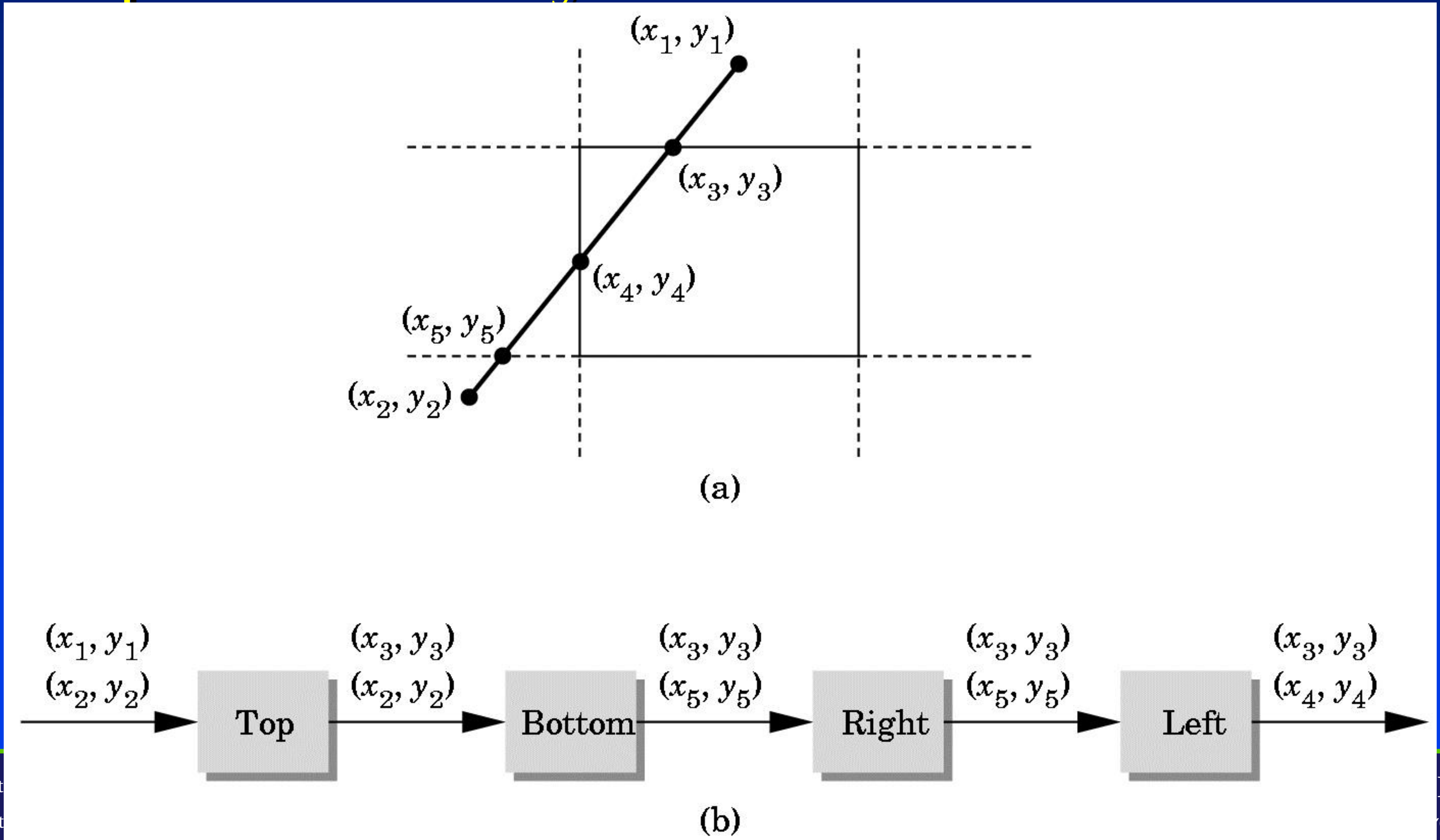
Clip against Each Boundary

- First clip against y_{max}
- $x_3 = x_1 + (y_{max} – y_1)(x_2 – x_1)/(y_2 – y_1)$
- $y_3 = y_{max}$



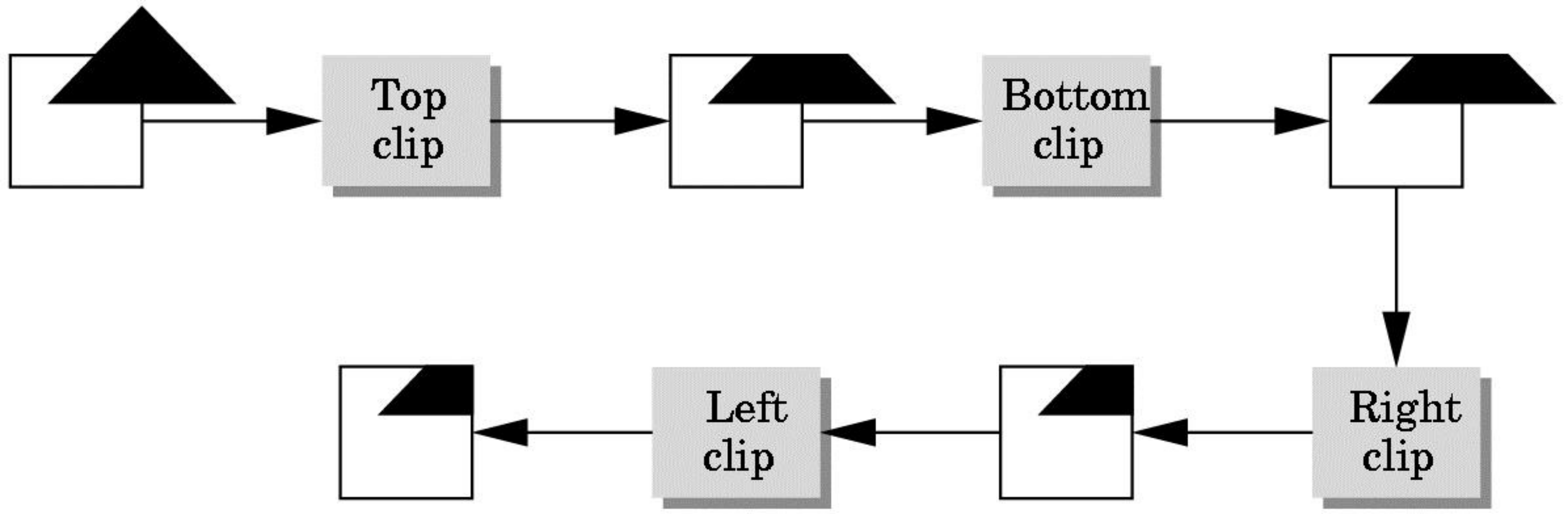
Clipping Pipeline

- Clip each boundary in turn



(Parallel) Clipping in Hardware

- Construct the pipeline stages in hardware so you can perform four clipping stages at once

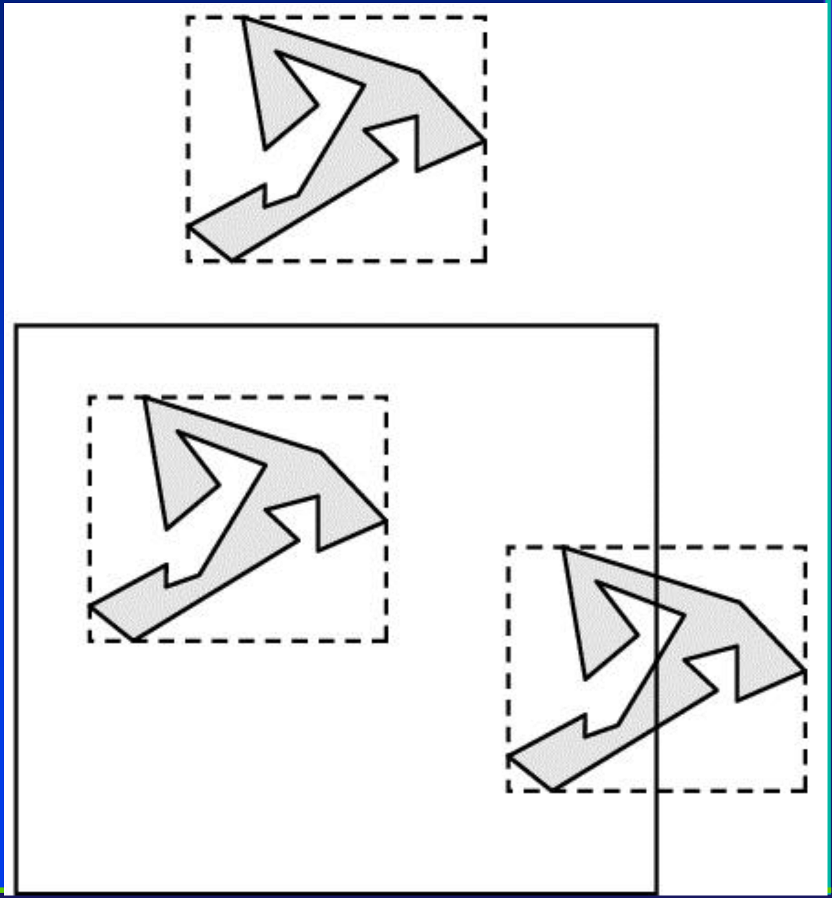


Clipping Complicated Objects

- Suppose you have many complicated objects, such as models of parts of a person with thousands of polygons each
- When and how to clip for maximum efficiency?
- How to clip text? Curves?

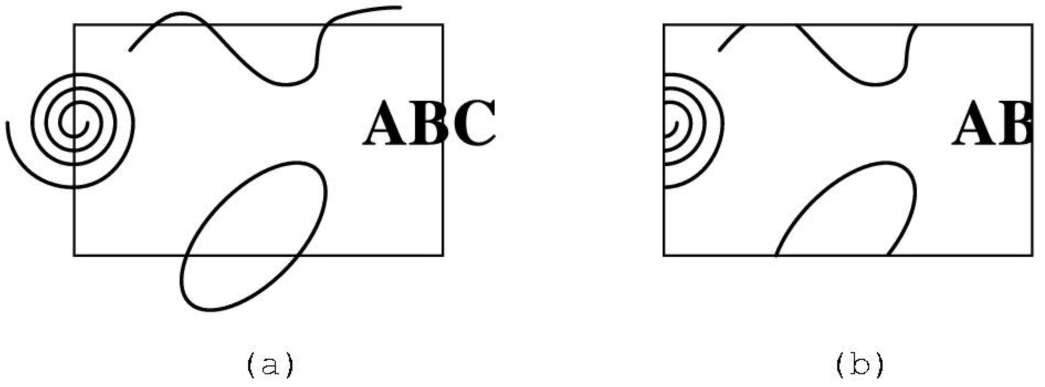
Clipping Other Primitives

- It may help to clip more complex shape early in the pipeline
- This may be simpler and less accurate
- One approach: bounding boxes (sometimes called *trivial accept-reject*)
- This is so useful that modeling systems often store bounding box



Clipping Curves, Text

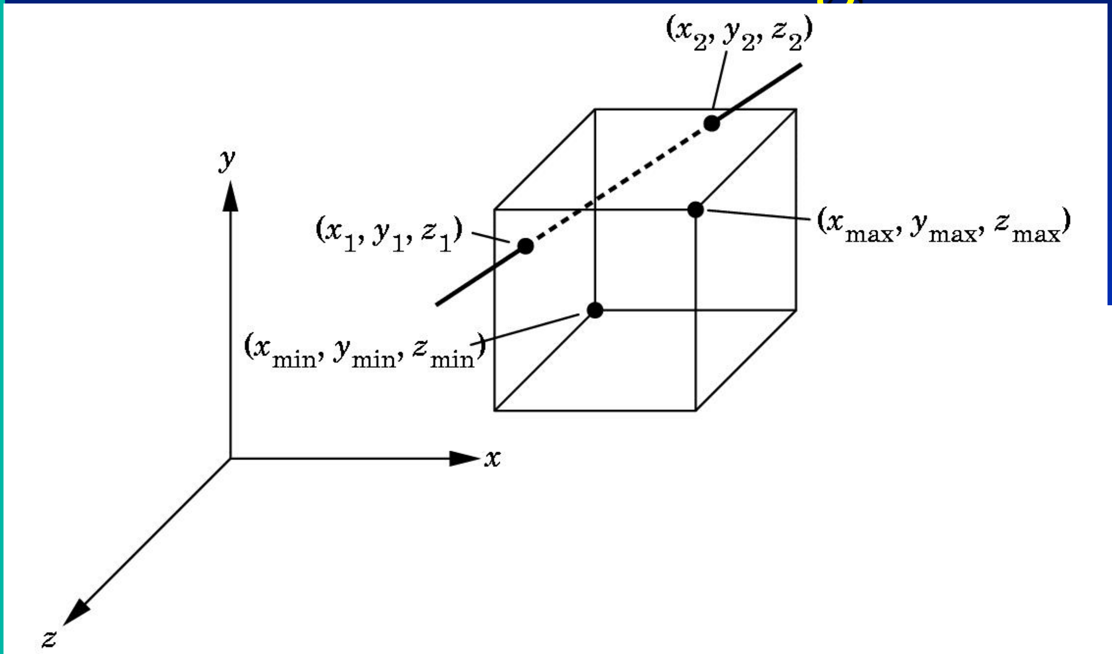
- Some shapes are so complex that they are difficult to clip analytically



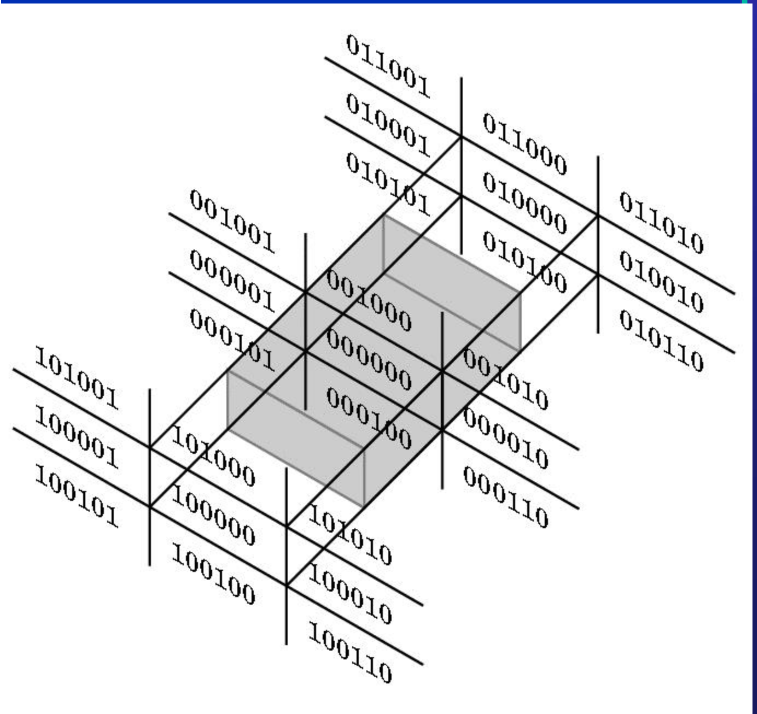
- Can approximate with line segments
- Can allow the clipping to occur in the frame buffer (pixels outside the screen rectangle aren't drawn)
- Called "*scissoring*"
- How does performance compare with others?

Clipping in 3D (Generalizations)

- Cohen-Sutherland regions



- Clip before perspective division



Geometric Processing

- **Front-end** processing steps (3D floating point; may be done on the CPU)
  - Evaluators (converting curved surfaces to polygons)
  - Normalization (modeling transformation, convert to world coordinates)
  - Projection (convert to screen coordinates)
  - Hidden-surface removal (object space)
  - Computing texture coordinates
  - Computing vertex normals
  - Lighting (assign vertex colors)
  - Clipping
  - Perspective division
  - Backface culling

Rasterization

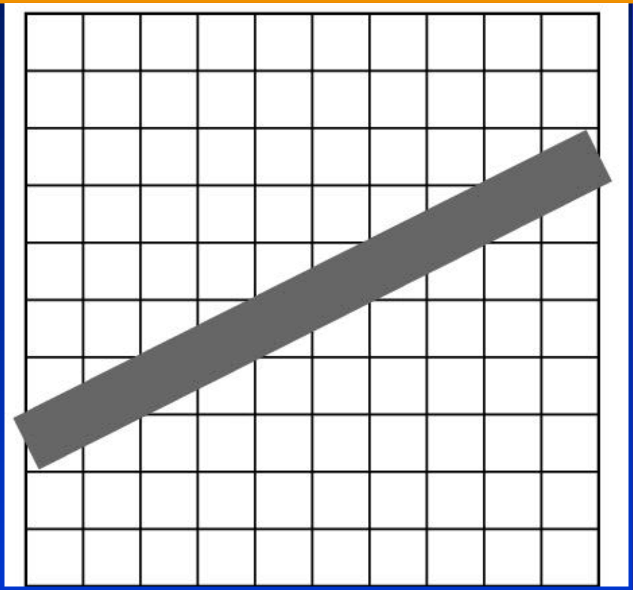
- **Back-end** processing works on 2D objects in screen coordinates
- Processing includes
  - Scan conversion of primitives including shading
  - Texture mapping
  - Fog
  - Scissors test
  - Alpha test
  - Stencil test
  - Depth-buffer test
  - Other fragment operations: blending, dithering, logical operations

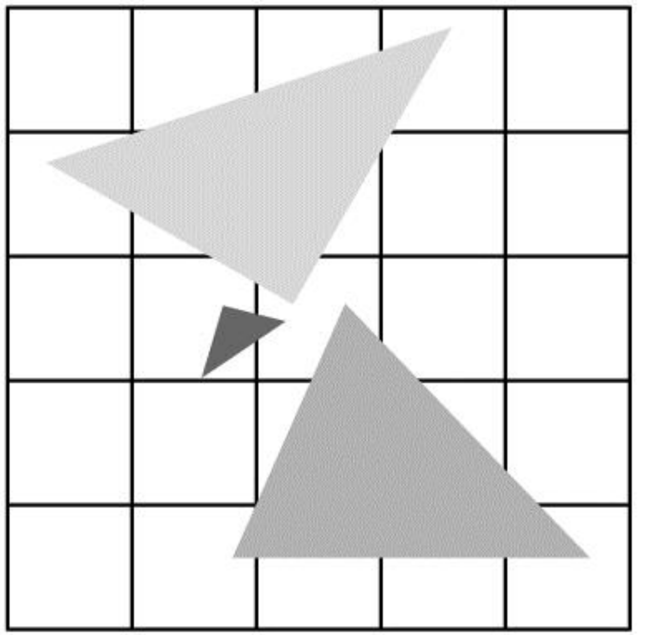
Display

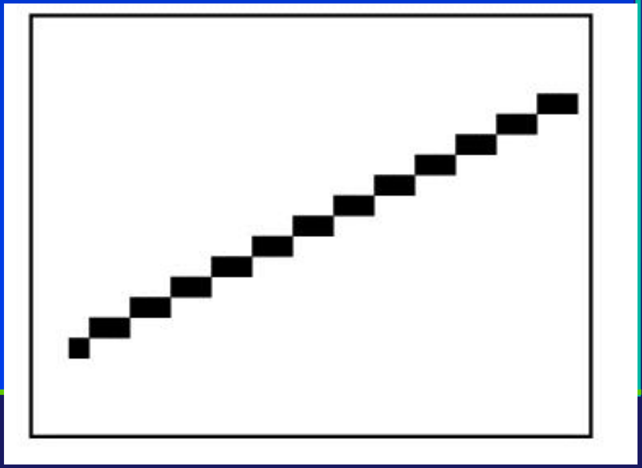
- RAM DAC converts frame buffer to video signal
- Other considerations:
  - Color correction
  - Antialiasing

Aliasing

- How to render the line with reduced aliasing?
- What to do when polygons share a pixel?

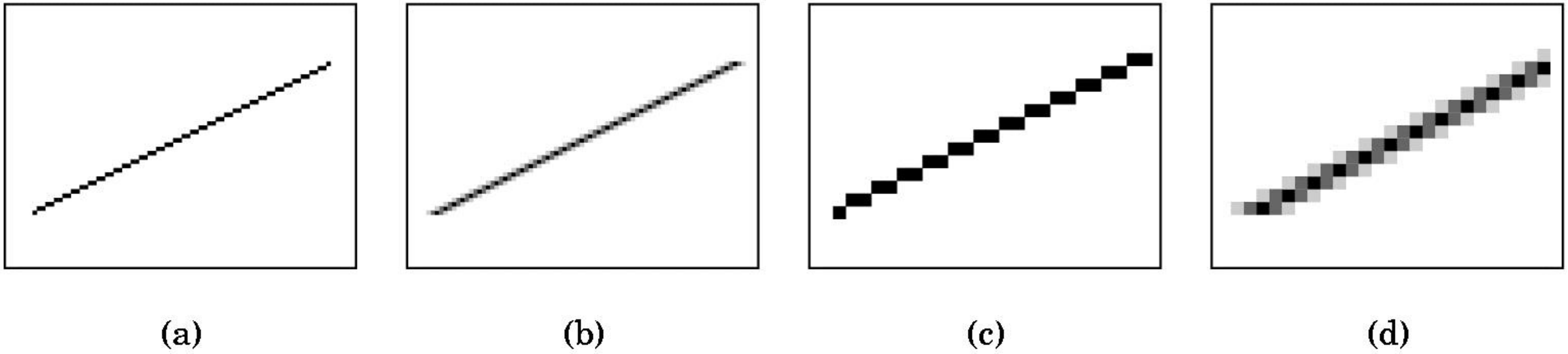






Anti-Aliasing

- Simplest approach: area-based weighting
- Fastest approach: averaging nearby pixels
- Most common approach: supersampling (patterned or with *jitter*)
- Best approach: weighting based on distance of pixel from center of line; Gaussian fall-off

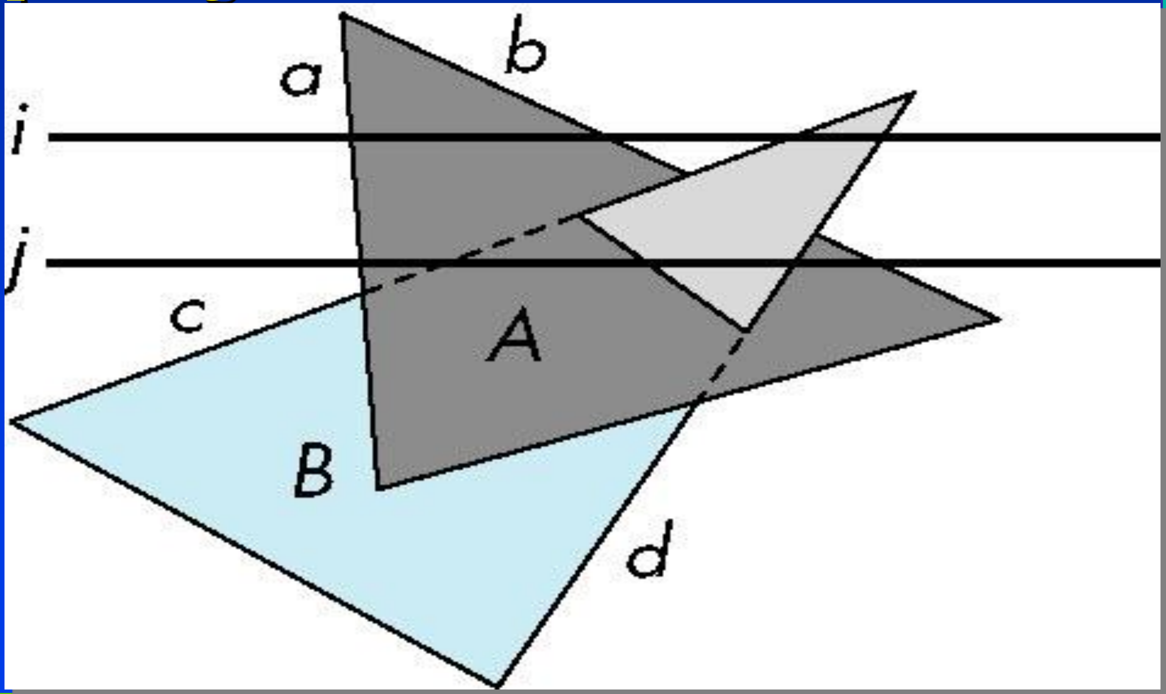


Temporal Aliasing

- Need *motion blur* for motion that doesn't flicker at slow frame rates
- Common approach: *temporal supersampling*
  - render images at several times within frame time interval
  - average results

Scan-line Algorithm

- Work one scan line at a time
- Compute intersections of faces along scanlines
- Keep track of all "open segments" and draw the closest



- More on HSR later

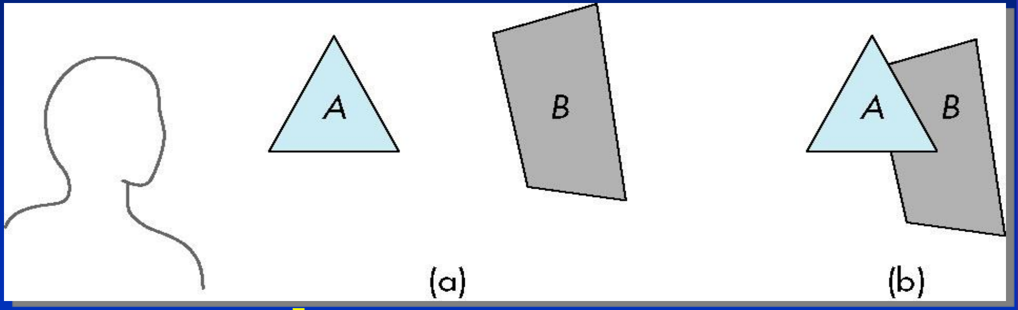
Hidden Surface Removal

- Object-space vs. Image-space
- The main image-space algorithm: z-buffer
- Drawbacks
  - Aliasing
  - Rendering invisible objects

- How would *object-space hidden surface removal* work?

Depth Sorting

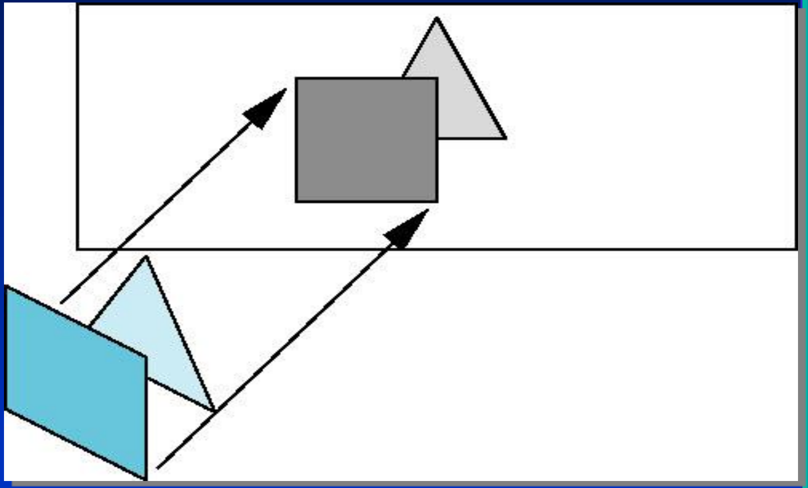
- The *painter's algorithm:* draw from back to front

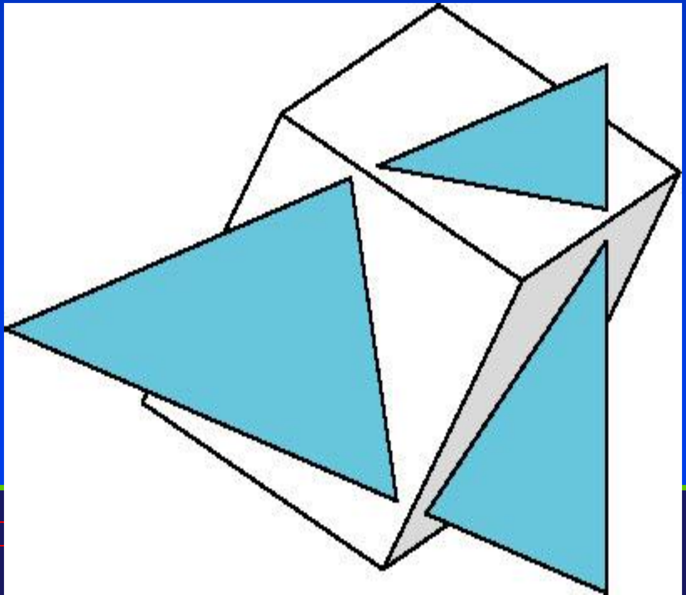


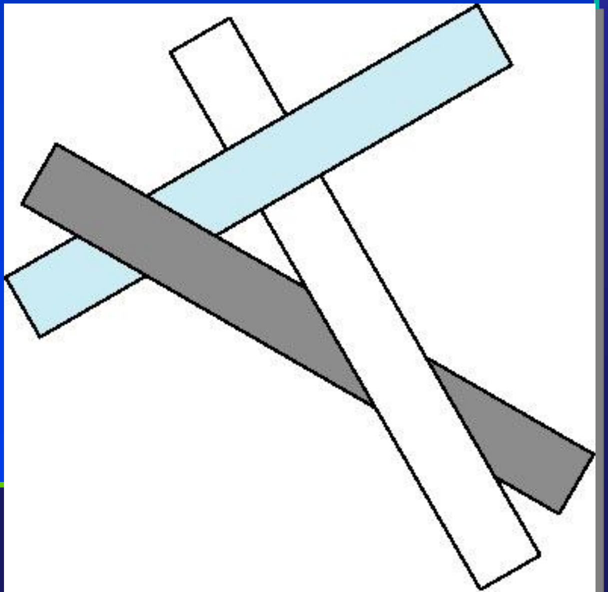
- Depth-sort hidden surface removal:
  - sort display list by z-coordinate from back to front
  - render/display
- Drawbacks
  - it takes some time (especially with bubble sort!)
  - it doesn't work

Depth-Sort Difficulties

- Polygons with overlapping projections
- Cyclic overlap
- Interpenetrating polygons
- What to do?





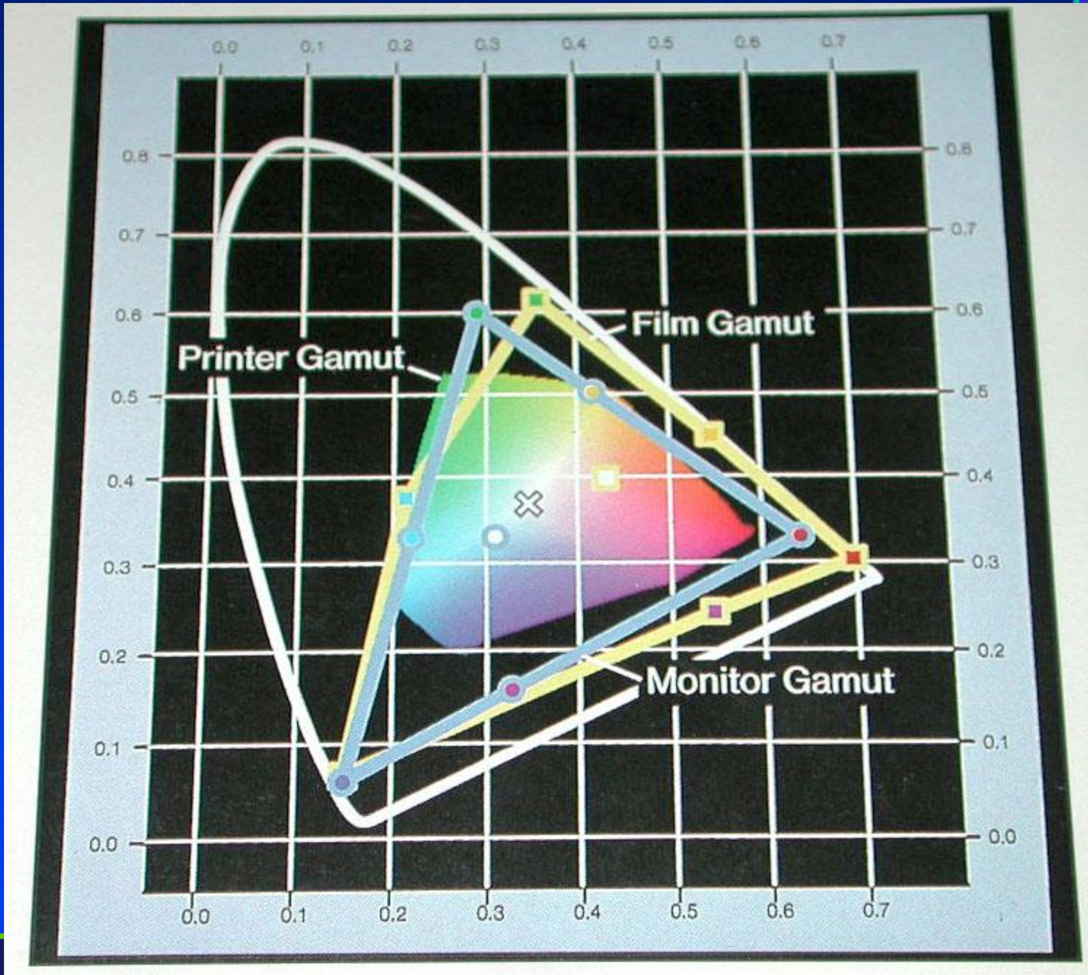


Display Considerations

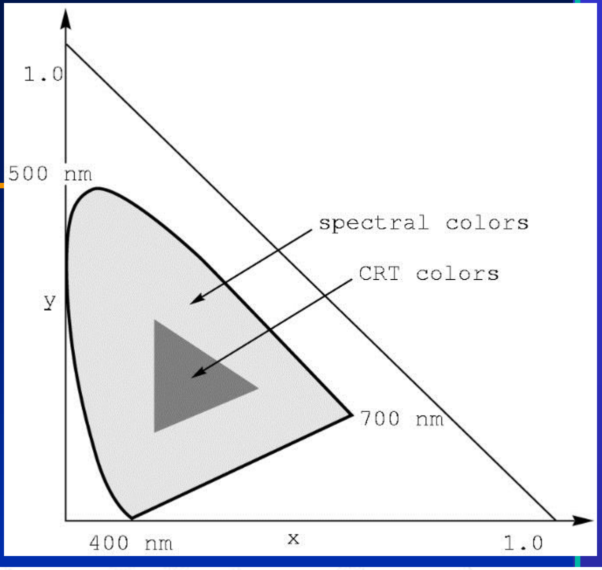
- Color systems
- Color quantization
- Gamma correction
- Dithering and Halftoning

Color Systems

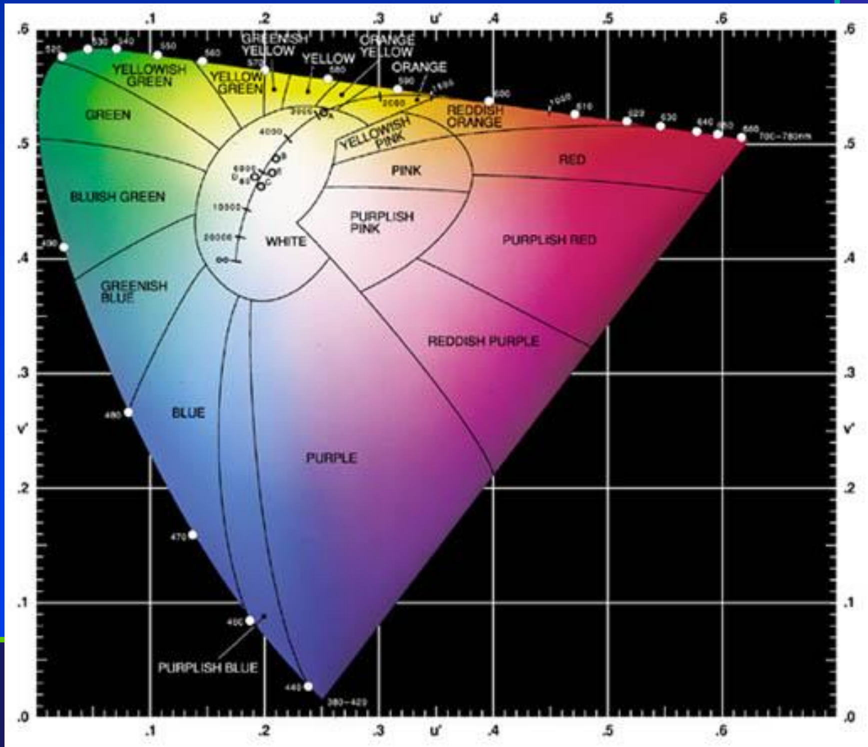
- RGB
- YIQ
- CMYK
- HSV, HLS
- Chromaticity
- Color gamut



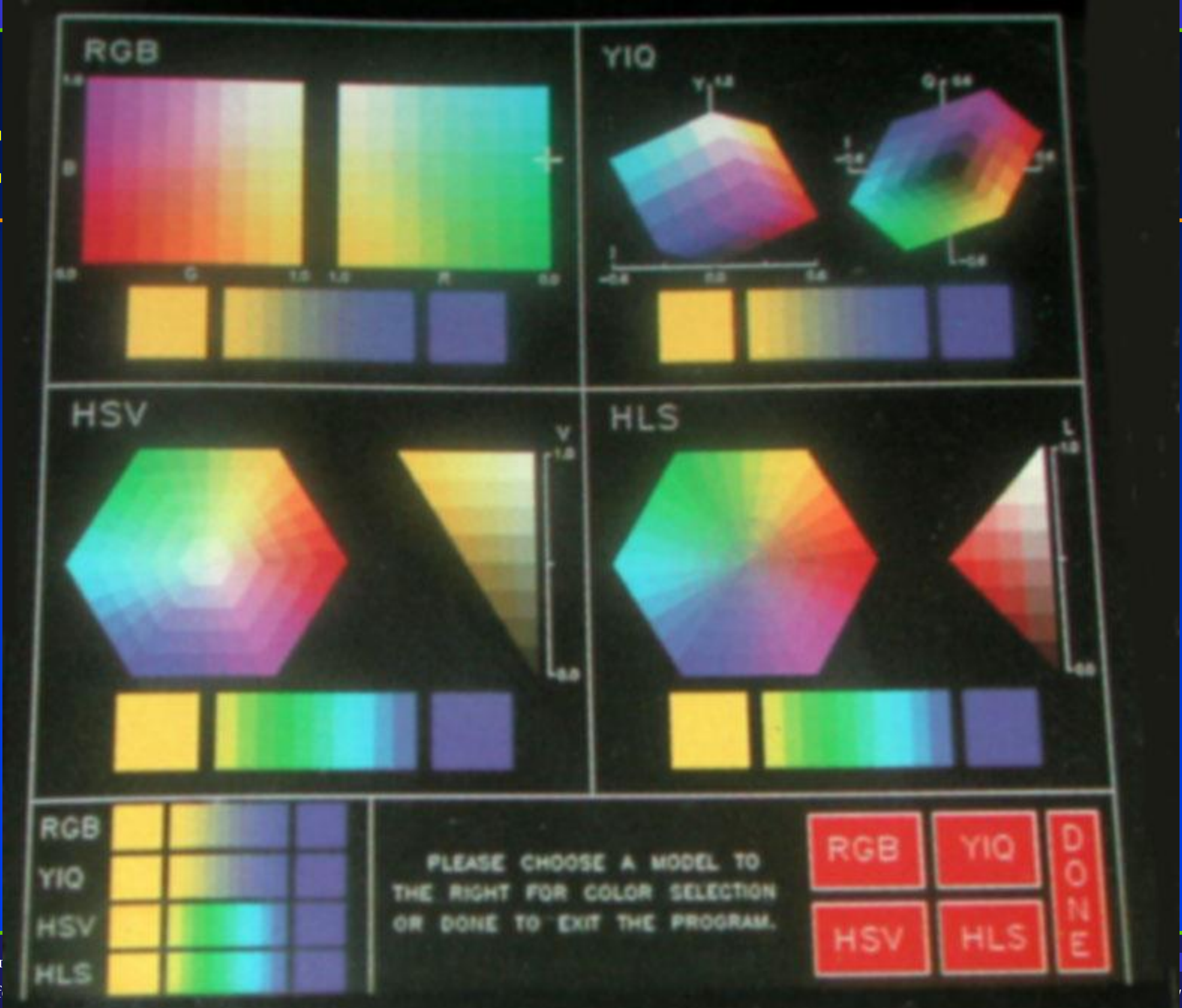
Chromaticity



- Tristimulus values: R, G, B values that we know of
- Color researchers often prefer chromaticity coordinates:
  - – $t1 = T1 / (T1 + T2 + T3)$
  - – $t2 = T2 / (T1 + T2 + T3)$
  - – $t3 = T3 / (T1 + T2 + T3)$
- Thus, $t1+t2+t3 = 1.0$.
- Use t1 and t2; t3 can be computed as 1-t1-t2
- Chromaticity diagram uses this approach for theoretical XYZ color system, where Y is luminance

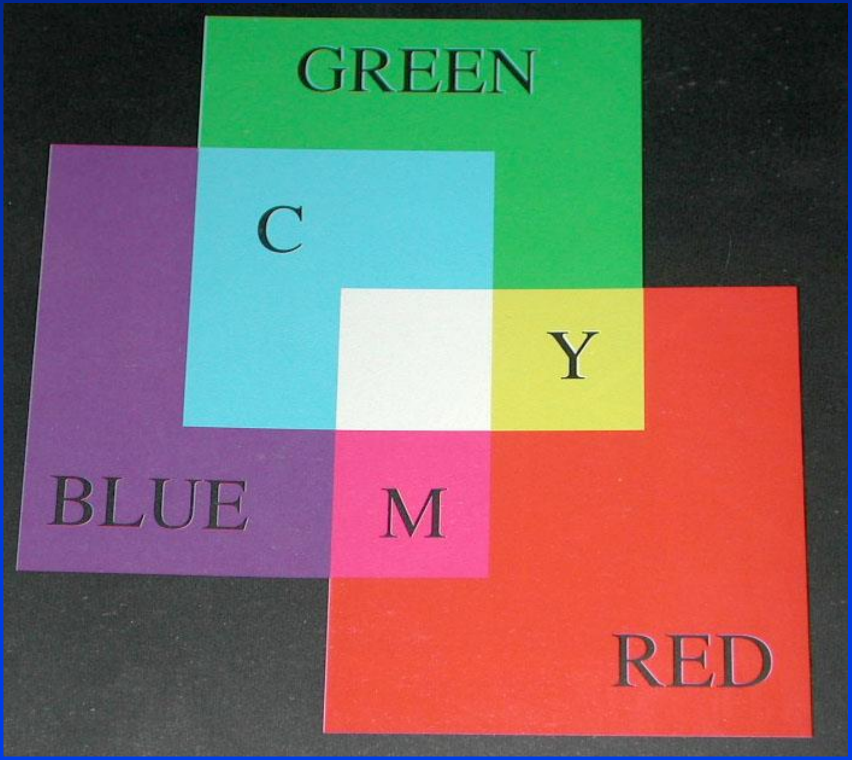


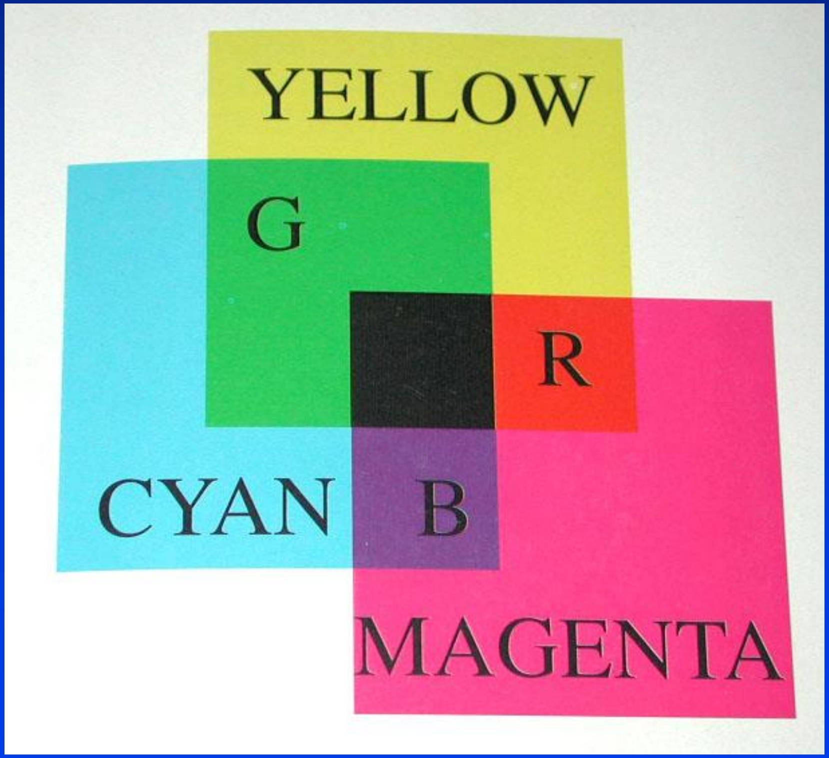
C



PLEASE CHOOSE A MODEL TO THE RIGHT FOR COLOR SELECTION OR DONE TO EXIT THE PROGRAM.

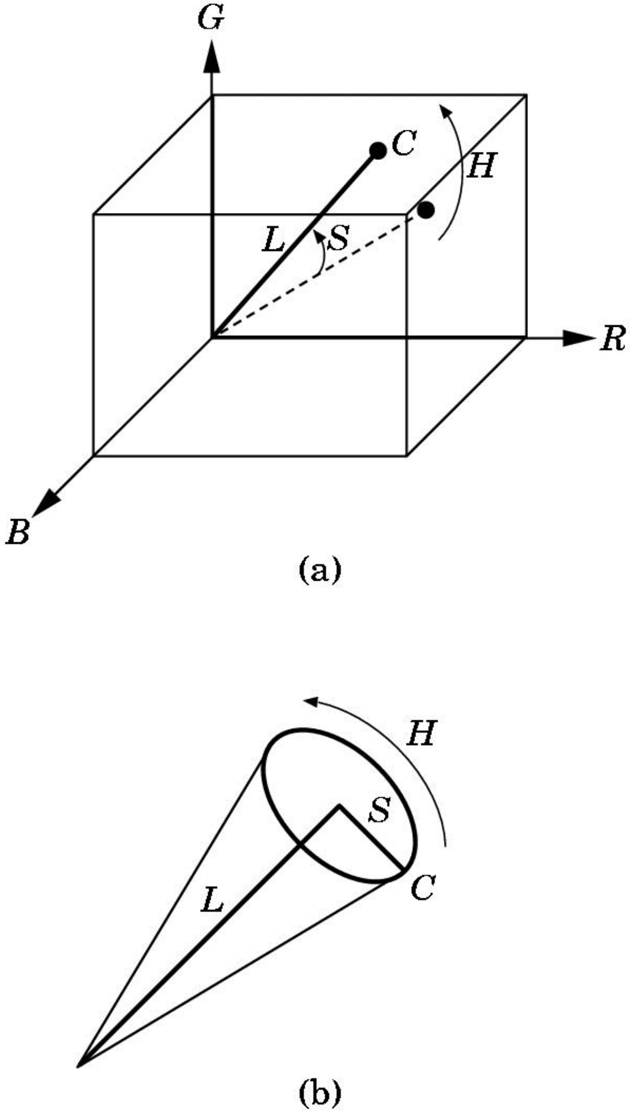
Additive and Subtractive Color





HLS

- Hue: “direction” of color: red, green, purple, etc.
- Saturation: intensity. E.g. red vs. pink
- Lightness: how bright

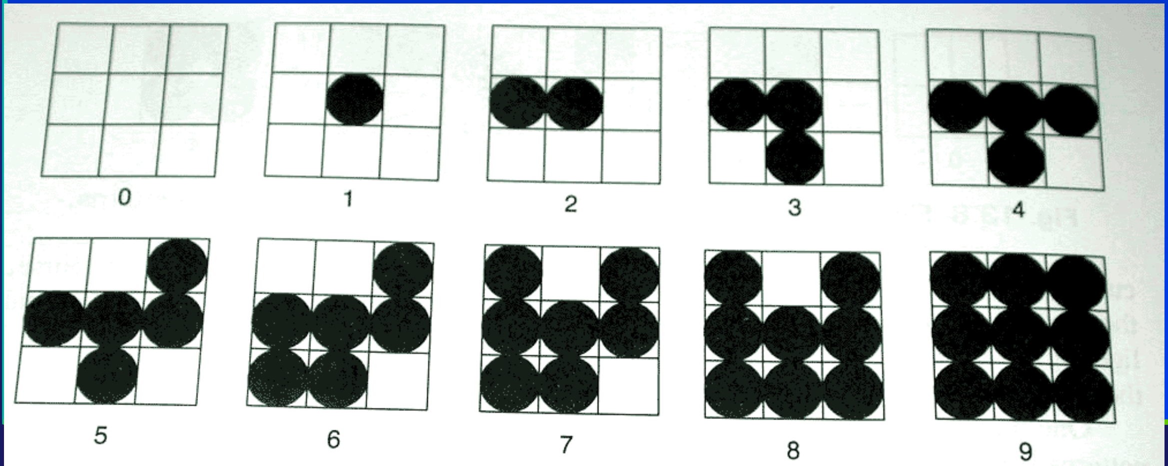


(a)

(b)

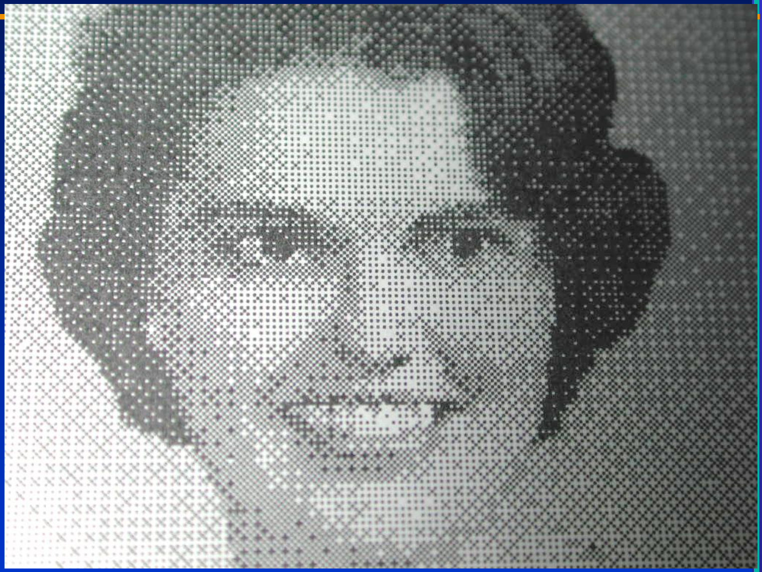
Dithering

- *Dithering* (patterns of b/w or colored dots) used for computer screens
- OpenGL can dither
- But, patterns can be visible and bothersome. A better approach?



Floyd-Steinberg Error Diffusion Dither

- Spread out "error term"
  - 7/16 right
  - 3/16 below left
  - 5/16 below
  - 1/16 below right
- Note that you can also do this for color images (dither a color image onto a fixed 256-color palette)



Halftoning

- How do you render a colored image when colors can only be **on** or **off** (e.g. inks, for print)?
- *Halftoning:* dots of varying sizes
- [But what if only fixed-sized pixels are available?]

Color Quantization

- Color quantization: modifying a full-color image to render with a 256-color palette
- For a fixed palette (e.g. web-safe colors), can use closest available color, possibly with error-diffusion dither
- Algorithm for selecting an adaptive palette?
  - E.g. Heckbert Median-cut algorithm
    - Make a 3-D color histogram
    - Recursively cut the color cube in half at a median
    - Use average color from each resulting box

Implementation Strategies

- Major approaches:
  - Object-oriented approach (pipeline renderers like OpenGL)
    - For each primitive, convert to pixels
    - Hidden-surface removal happens at the end
  - Image-oriented approach (e.g. ray tracing)
    - For each pixel, figure out what color to make it
    - Hidden-surface removal happens early
- Considerations on object-oriented approach
  - Memory requirements were a serious problem with the object-oriented approach until recently
  - Object-oriented approach has a hard time with interactions between objects
  - The simple, repetitive processing allows hardware speed: e.g. a 4x4 matrix multiply in one instruction
  - Memory bandwidth not a problem on a single chip

Hardware Implementations

- Pipeline architecture for speed
  (but what about latency?)
- Originally, whole pipeline on CPU
- Later, back-end on graphics card
- Now, whole pipeline on graphics card
- What's next?

Future Architectures?

- 10+ years ago, fastest performance of 1M polygons per second cost millions
  - Performance limited by memory bandwidth
  - Main component of price was lots of memory chips
  - Now a single graphics chip is faster (memory bandwidth on a chip is much greater)
- Fastest performance today achieved with several parallel commodity graphics chips (*Playstation farm?*)
  - Plan A: send 1/n of the objects to each of the n pipelines; merge resulting images (with something like z-buffer algorithm)
  - Plan B: divide the image into n regions with a pipeline for each region; send needed objects to each pipeline