From Models to Rasterization

Application → Geometry → Rasterization

3D Model →

meshing

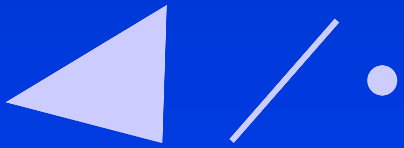
decimation

collision detection

animation

…

→ Rendering primitives

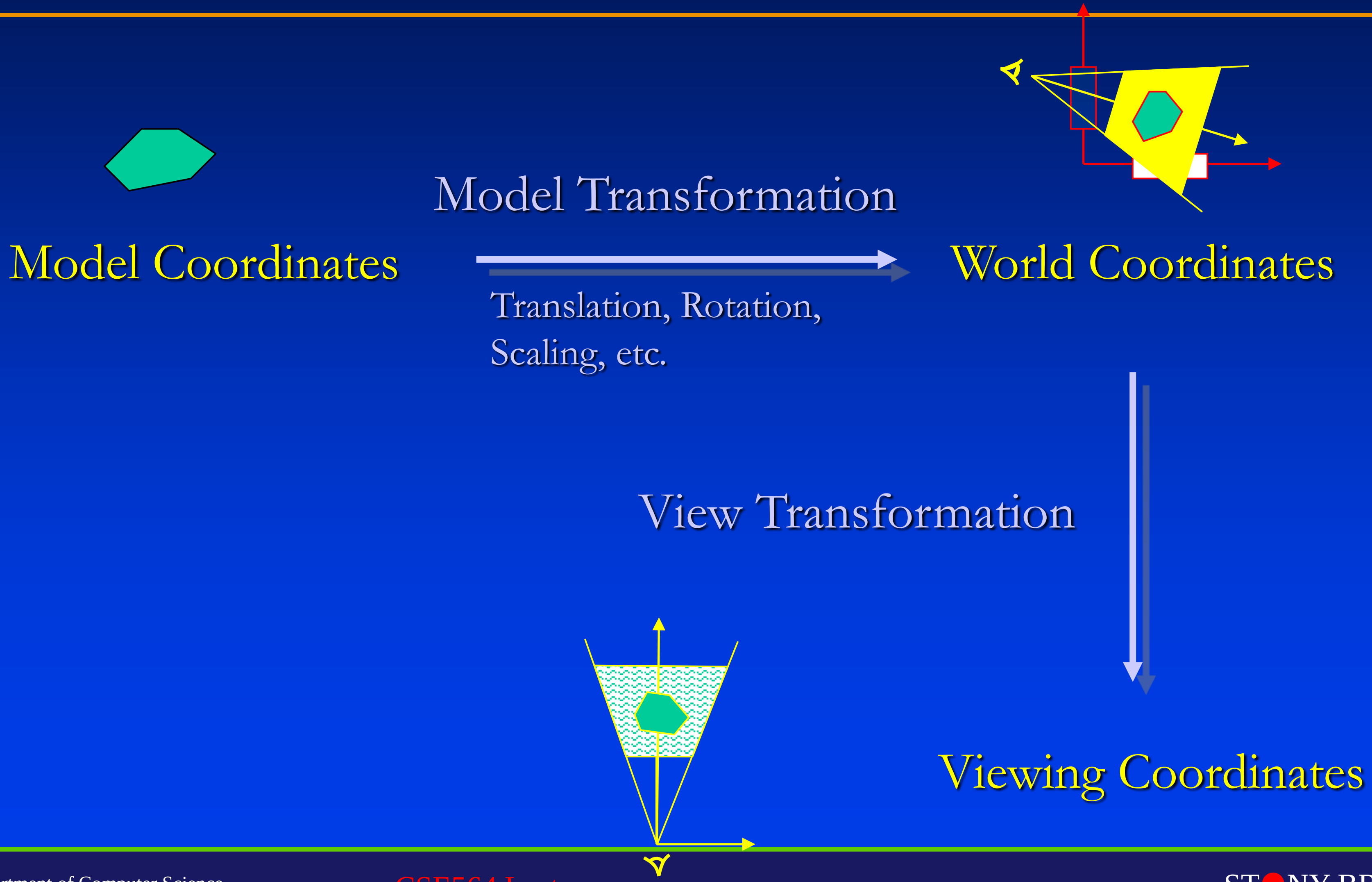


Software-based processing / modifications

Geometry

Transformations → Lighting → Projection → Clipping

Geometry: Transformations



Model Coordinates

Model Transformation

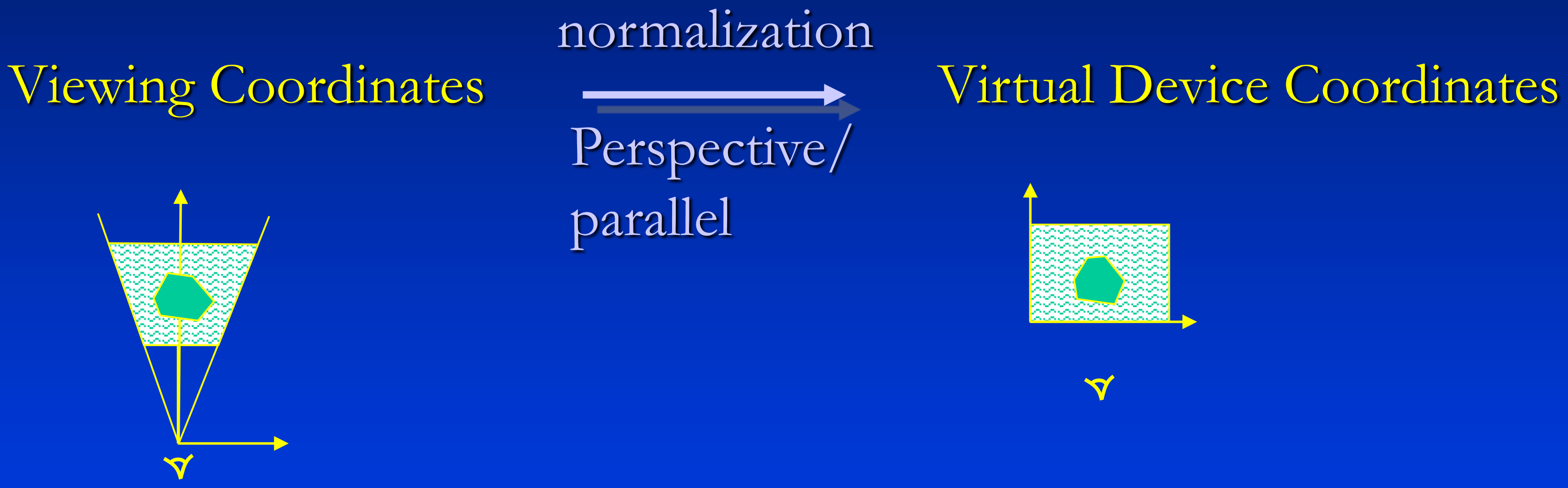
Translation, Rotation, Scaling, etc.

World Coordinates

View Transformation

Viewing Coordinates

Geometry: Projection



Viewing Coordinates

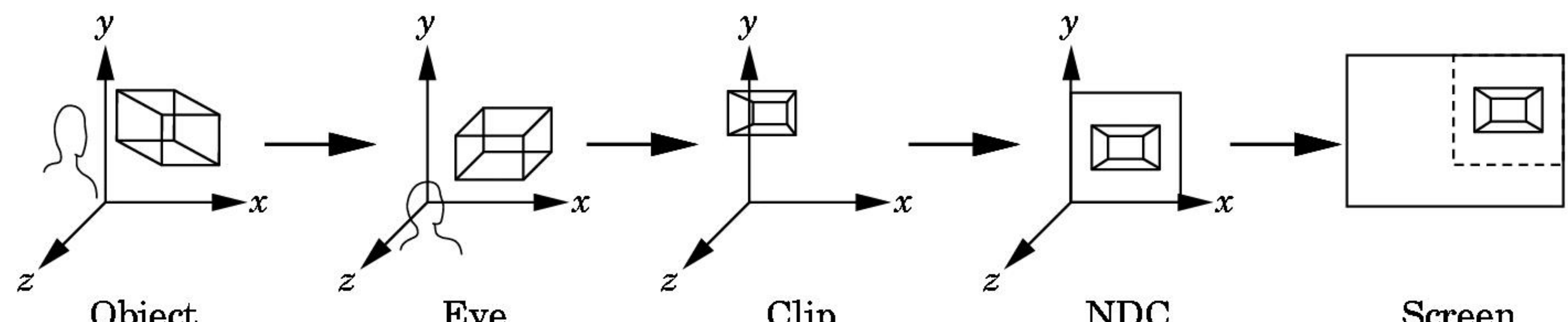
normalization

Perspective/ parallel

Virtual Device Coordinates

Geometric Transformations

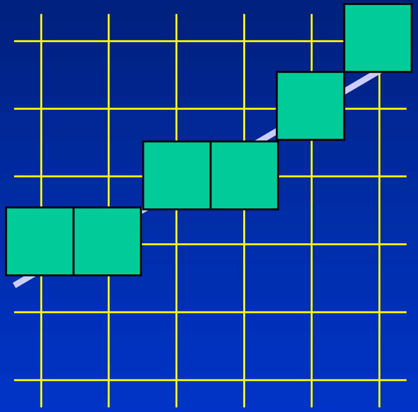
- Five coordinate systems of interest:
  - Object coordinates
  - Eye (world) coordinates [after modeling transform, viewer at the origin]
  - Clip coordinates [after projection]
  - Normalized device coordinates [after ÷w]
  - Window (screen) coordinates [scale to screensize]



Rasterization

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

Screen = matrix

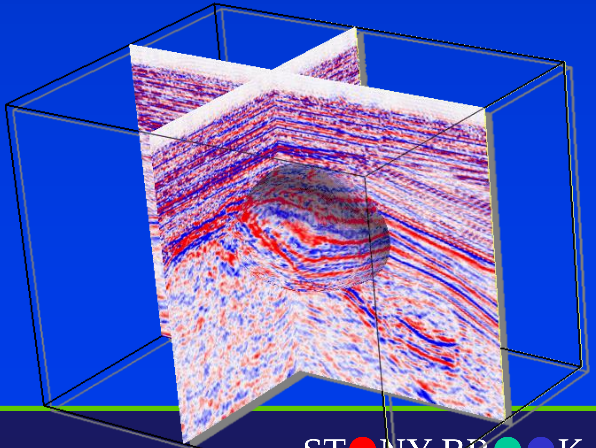


Scan conversion of lines:
- naive version
- Bresenham algorithm

Scan conversion of polygons

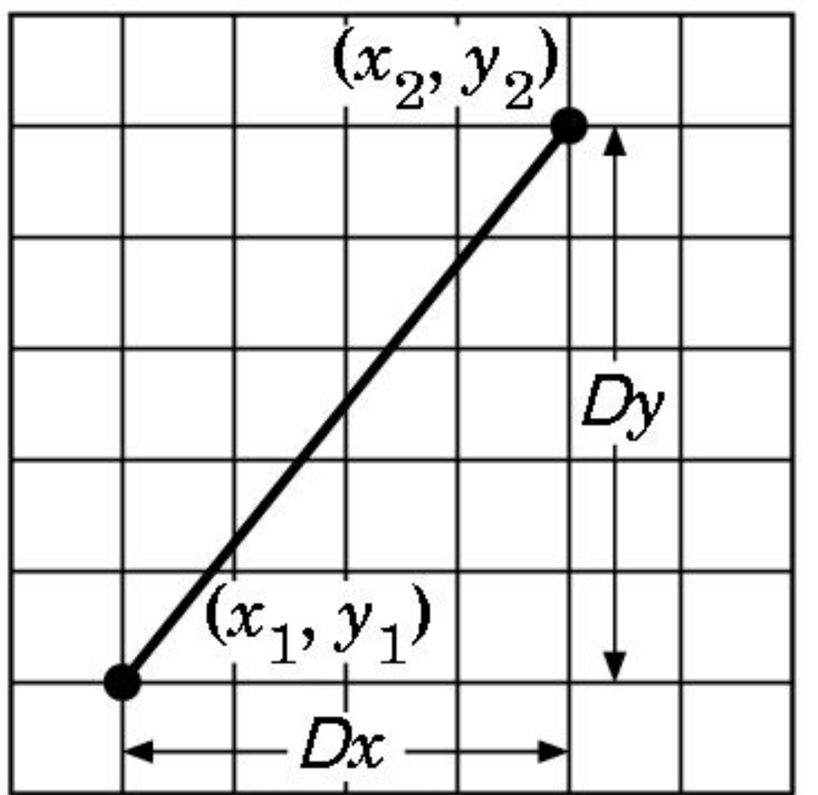
Aliasing / antialiasing

Texturing



Rendering Line Segments (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)?
- Use the equation
  $y = mx + h$ (explicit)
- What about horizontal vs. vertical lines?

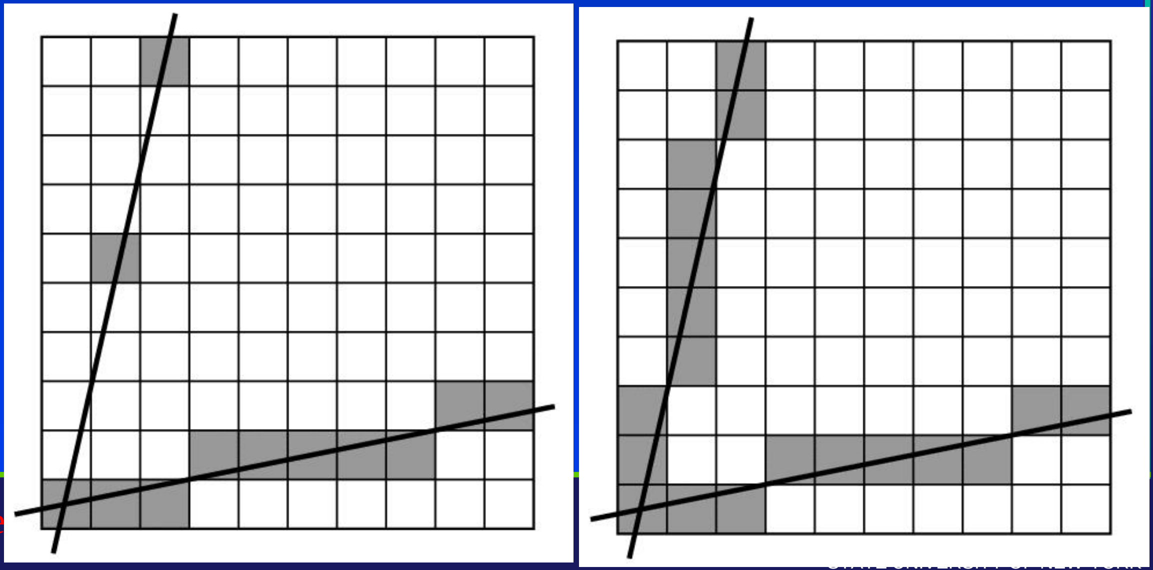


DDA Algorithm

- DDA: Digital Differential Analyzer (DDA)

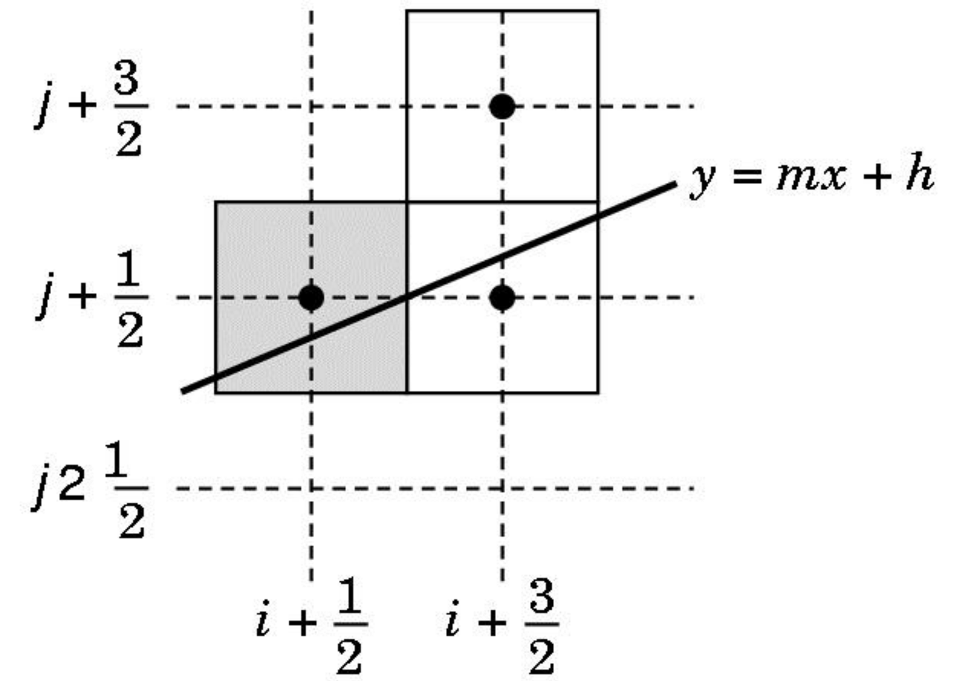
```
for ($x=x_1$; $x<=x_2$; x++)
        y += m;
        draw_pixel(x, y, color)
```

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



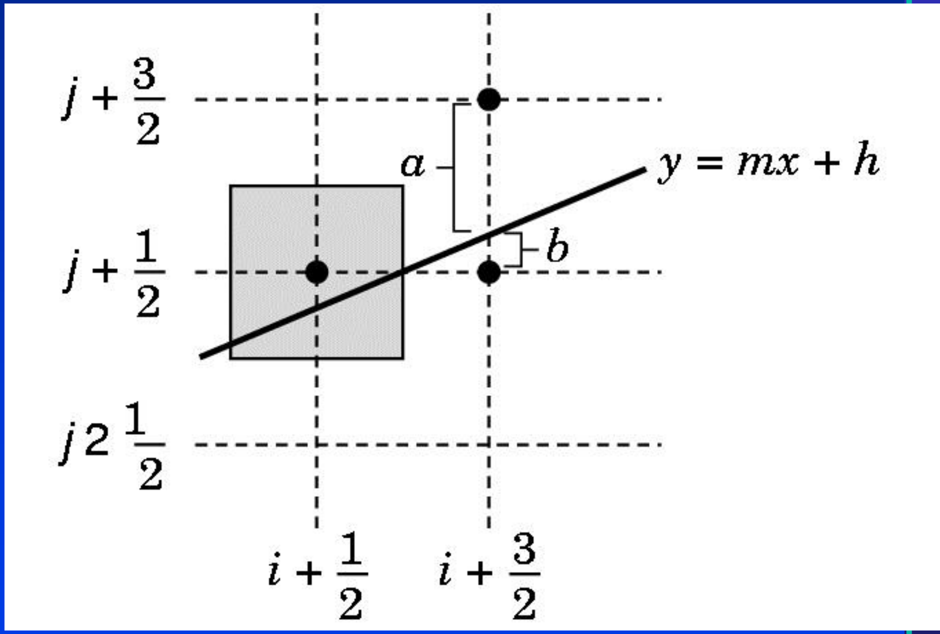
Bresenham’s Algorithm

- 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?



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

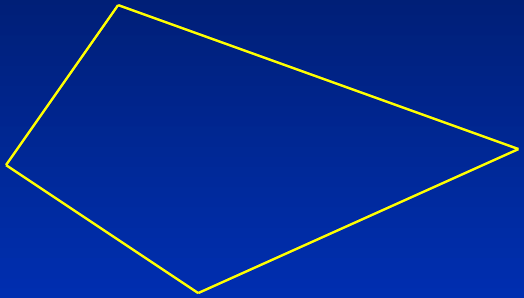
- Set up loop computing *d* at x_1, y_1

```
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)

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



Convex

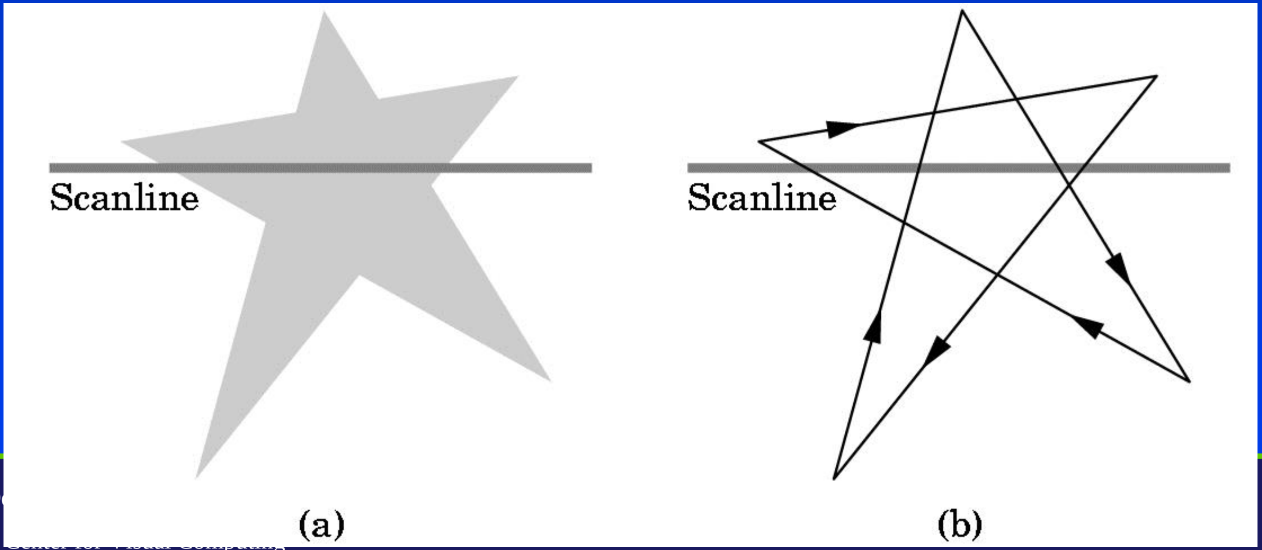
Not Convex

Convex

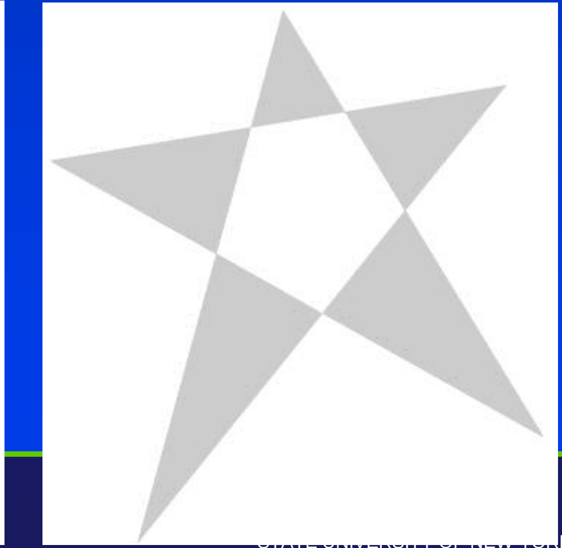
- A polygon is convex if…
  - A line segment connecting any two points on the polygon is contained in the polygon.
  - If you can wrap a rubber band around the polygon and touch all of the sides, the polygon is convex

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?

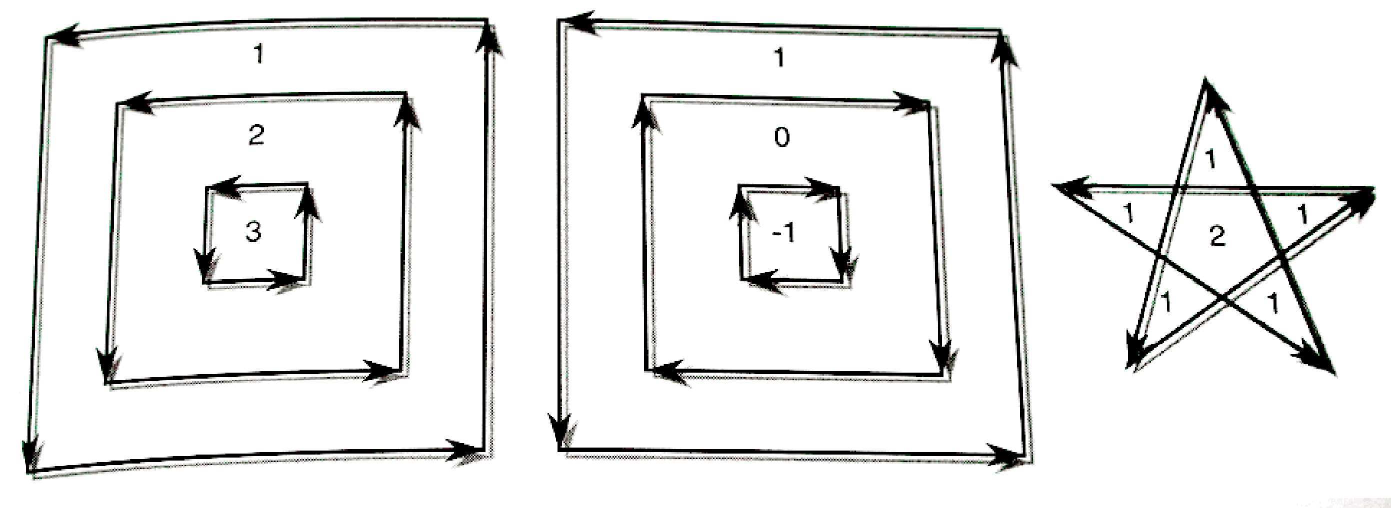


(a) (b)



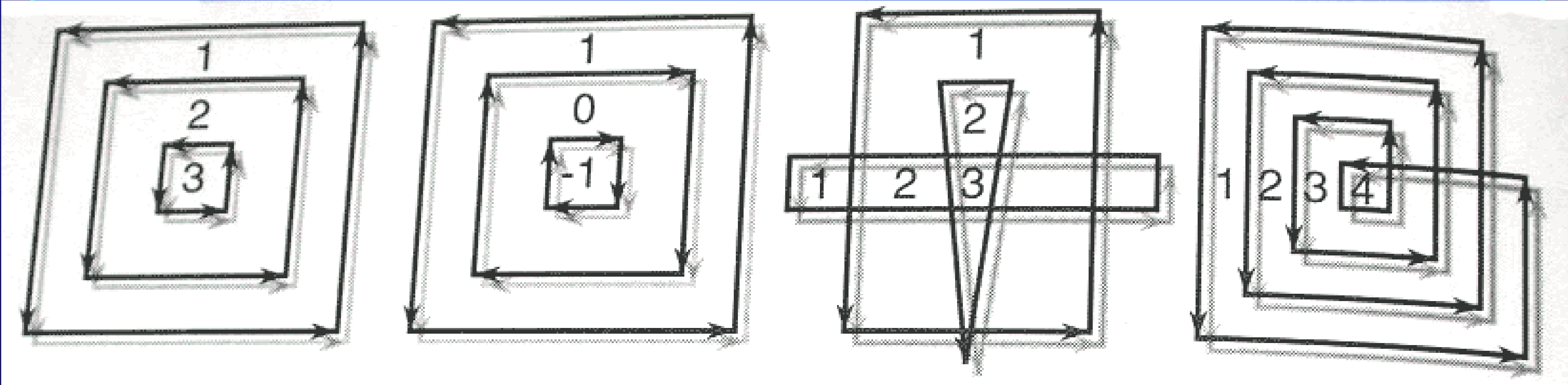
Winding Test

- 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?

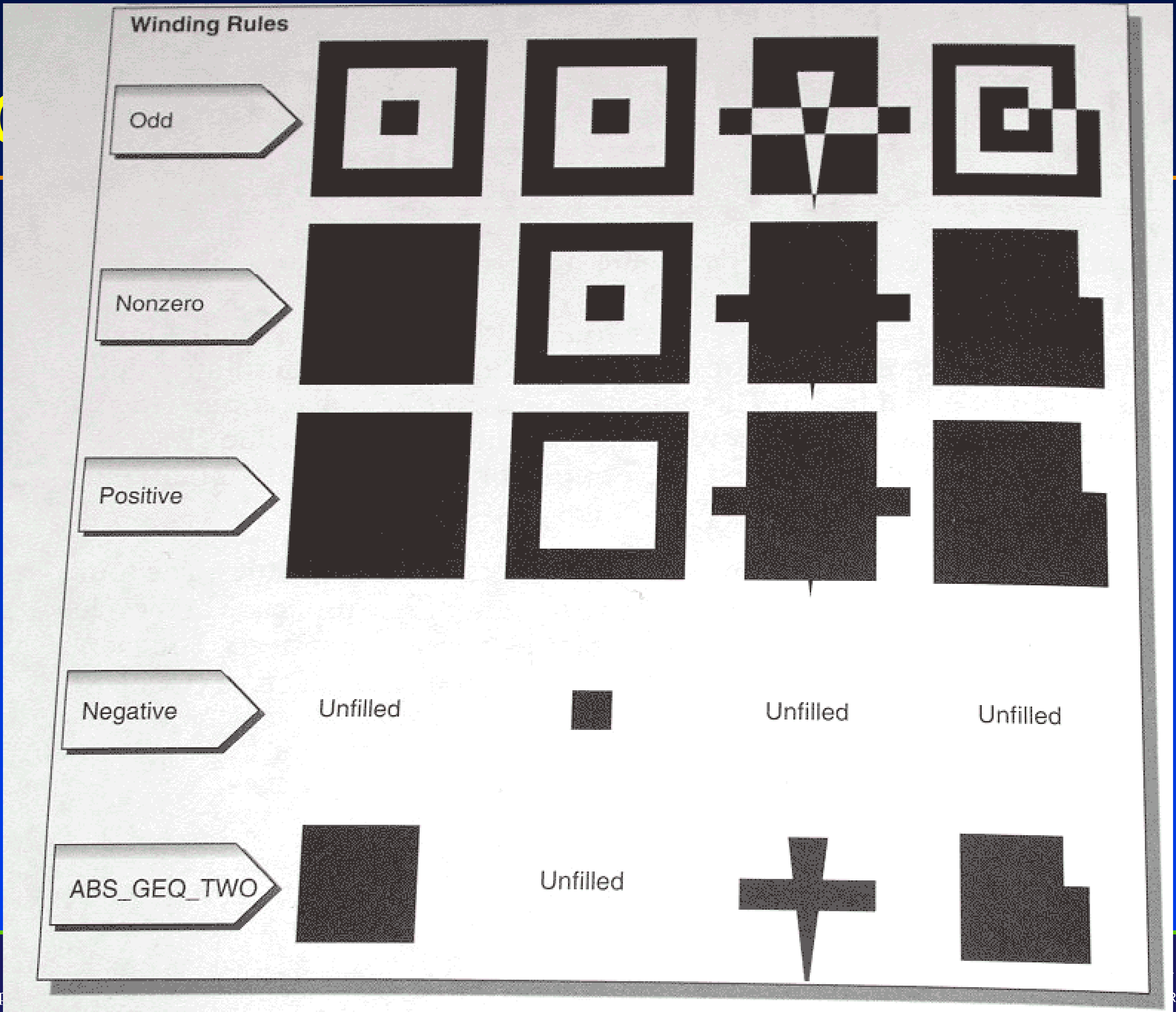


OpenGL and Concave polygons

- 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



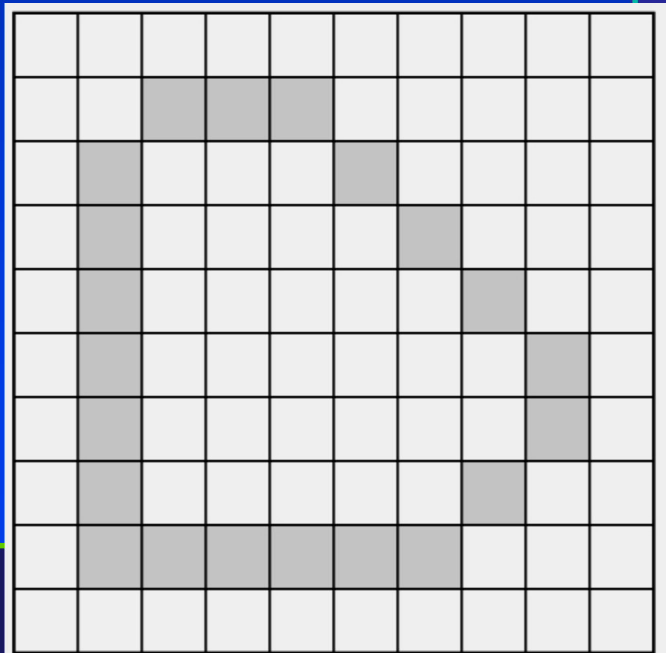
Win


Winding Rules
Odd
Nonzero
Positive
Negative
Unfilled
Unfilled
Unfilled
ABS_GEQ_TWO
Unfilled

Scan-Converting a Polygon

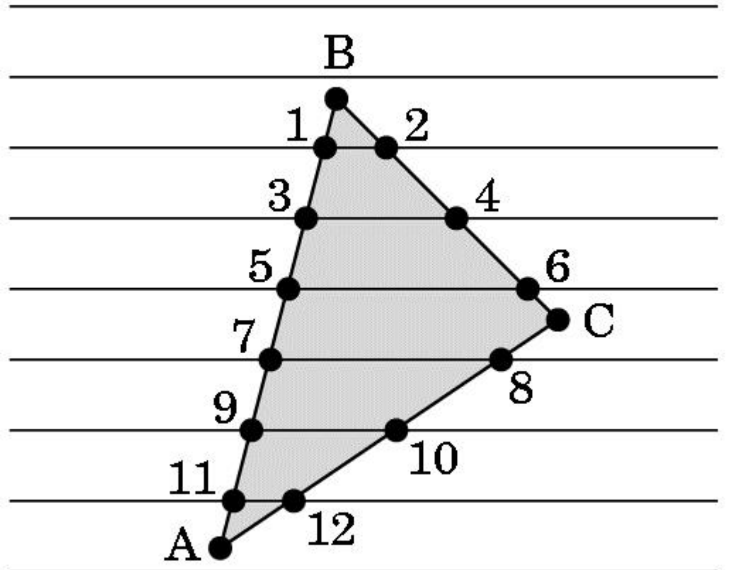
- General approach: any ideas?
- One idea: *flood fill*
  - – Draw polygon edges
  - – Pick a point (x,y) inside and **flood fill** with DFS

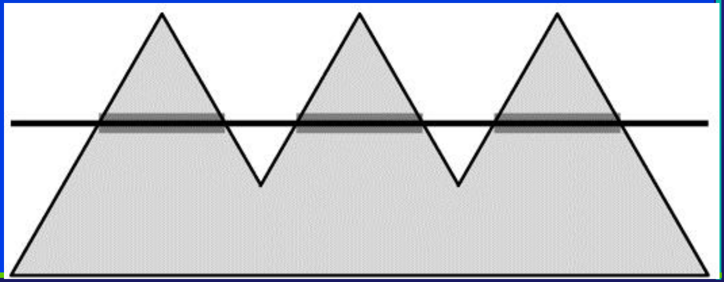
```
flood_fill(x,y) {
  if (read_pixel(x,y)==white) {
     write_pixel(x,y,black);
     flood_fill(x-1,y);
     flood_fill(x+1,y);
     flood_fill(x,y-1);
     flood_fill(x,y+1);
```



Scan-Line Approach

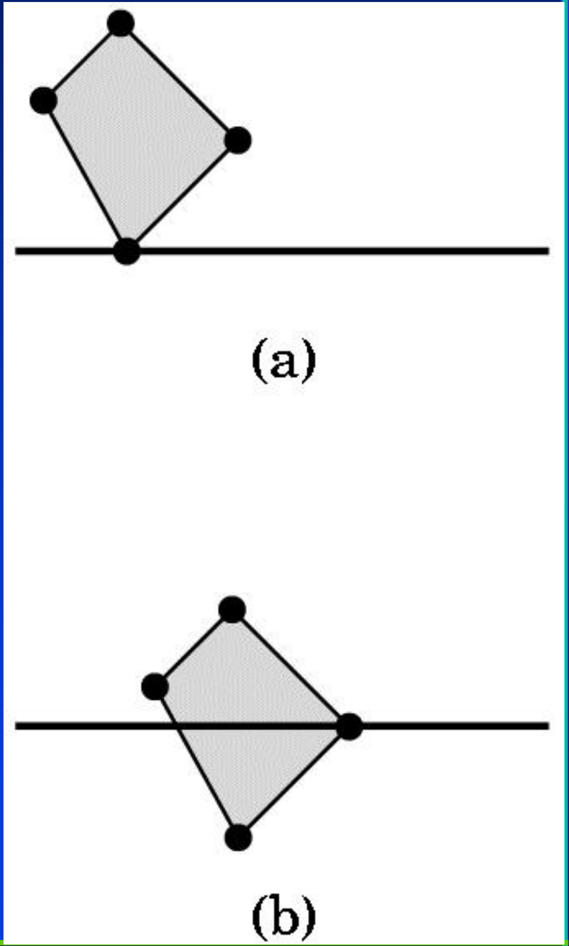
- More efficient way: use a scan-line rasterization algorithm
- For each y value, compute x intersections. Fill according to winding rule
- How to compute intersection points?
- How to handle shading?
- Some hardware can handle multiple scanlines in parallel





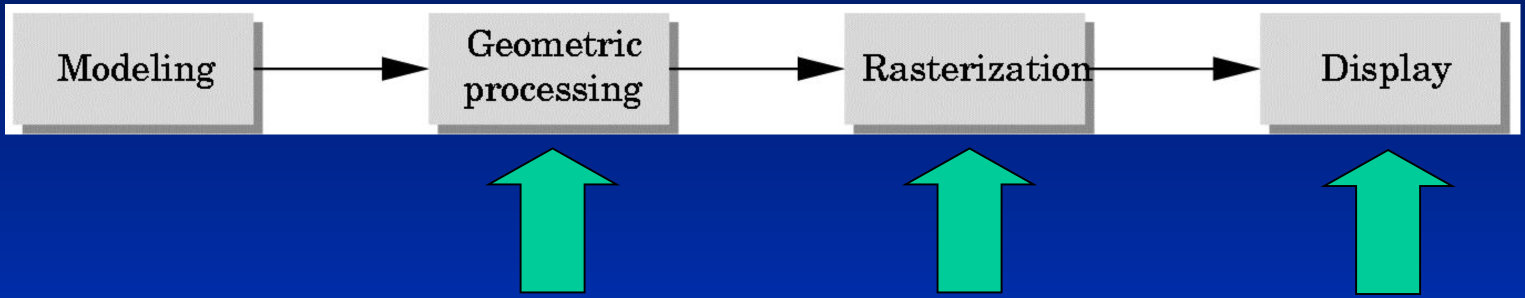
Singularities (Special Cases)

- 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



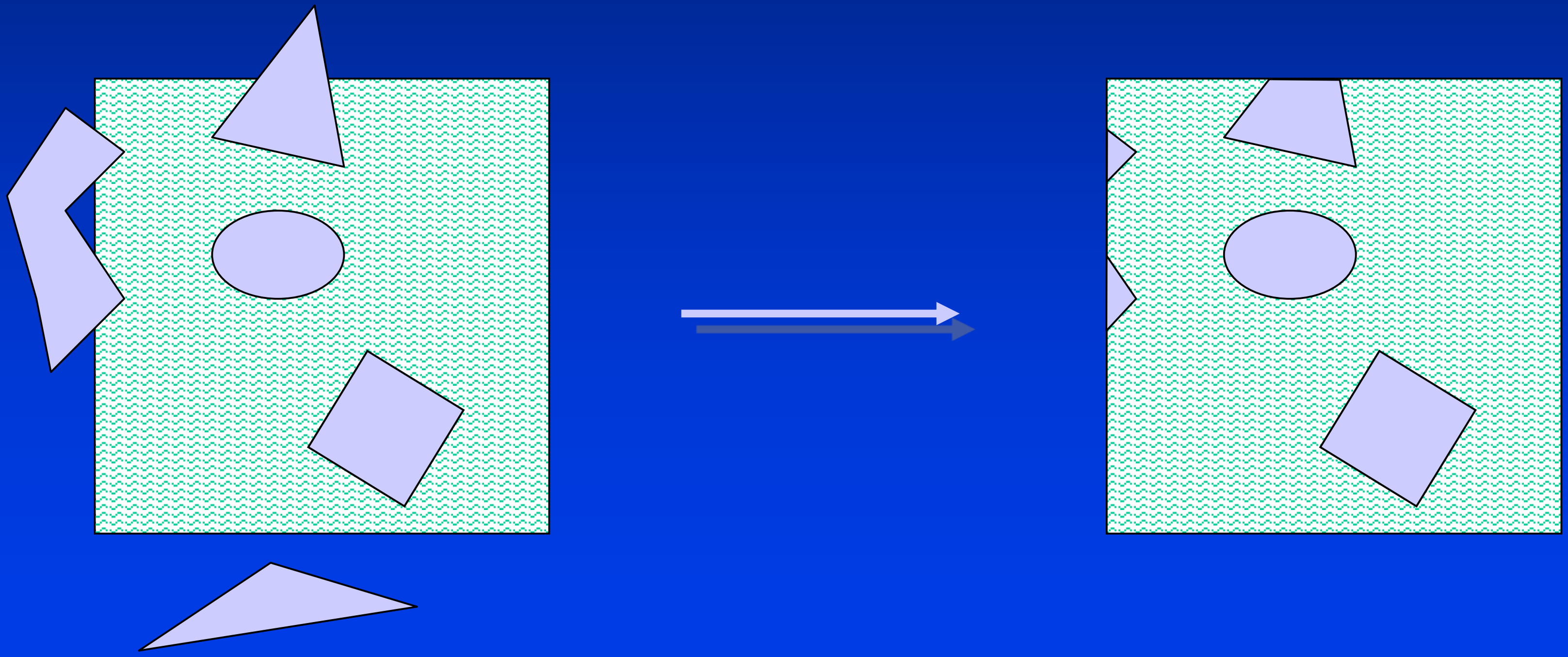
Computer Graphics: Geometric Clipping

Rendering Pipeline

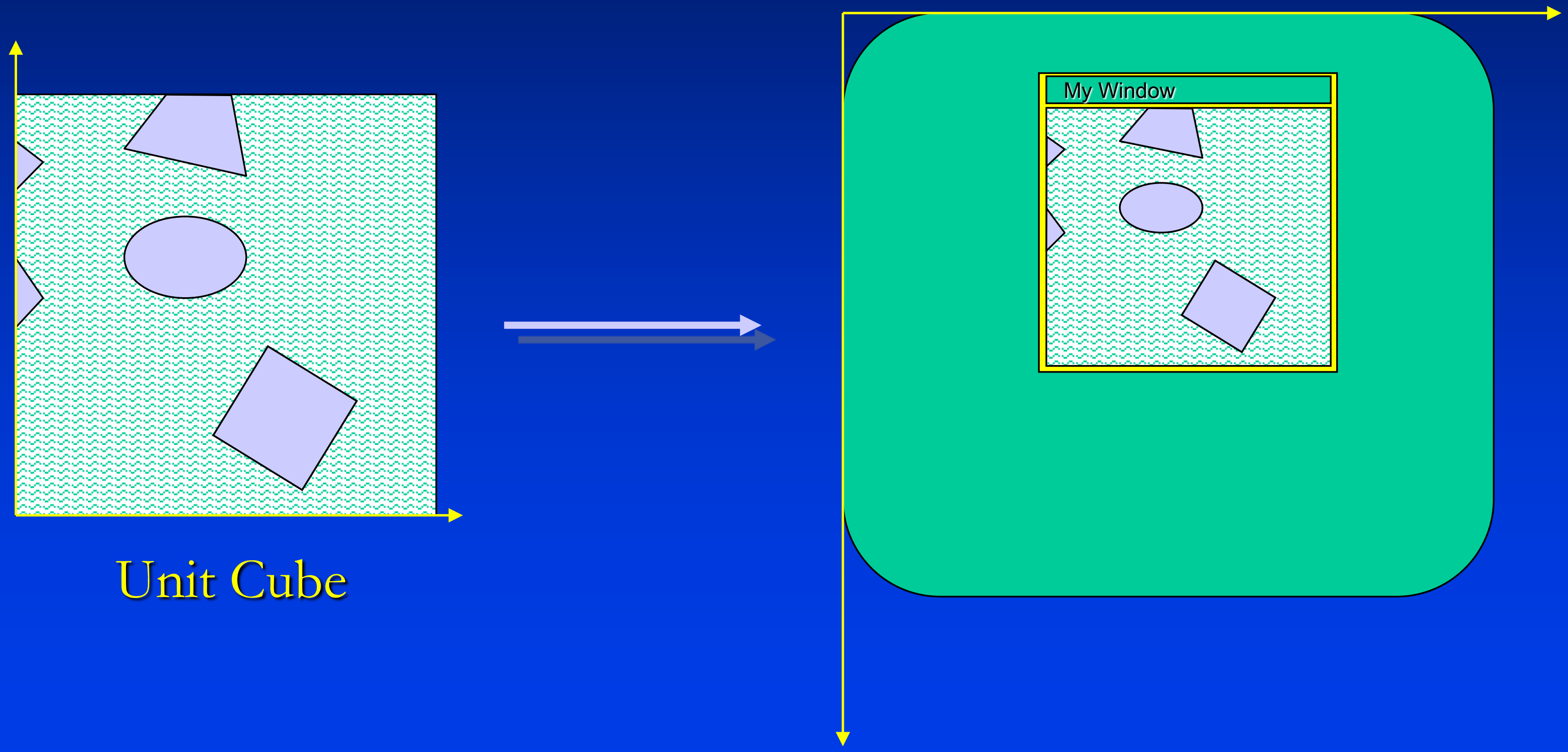


- Geometric processing: normalization, clipping, hidden surface removal, lighting, projection (*front end*)
- Rasterization or scan conversion, including texture mapping (*back end*)
- Fragment processing and display

Geometry: Clipping



Geometry: Device Coordinates



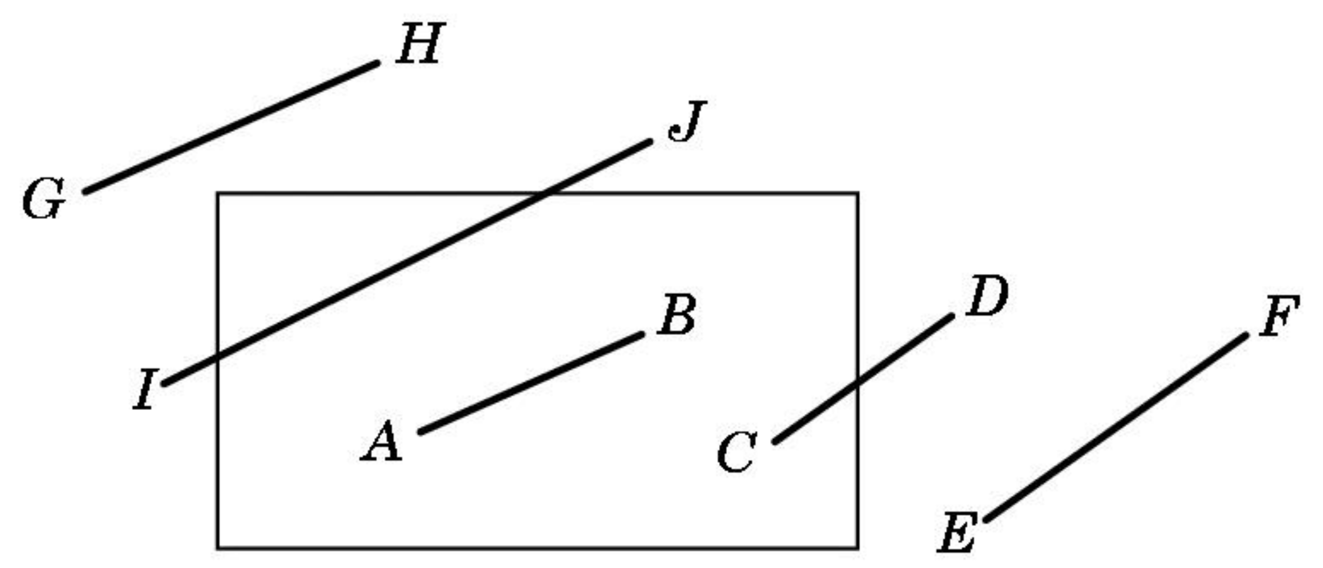
Unit Cube

How Do We Define a Window?

- Window
- Viewport

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



The Fundamental Operation

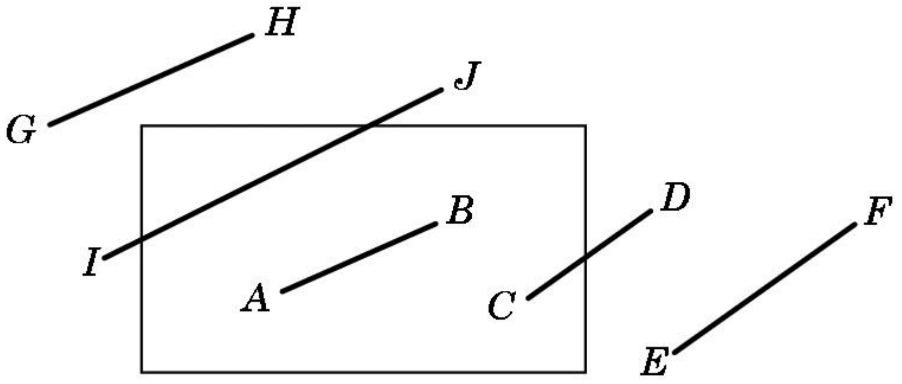
- In geometric clipping, the most fundamental operation is how to compute line-line intersection: (1) whether two lines are intersecting or NOT; (2) if they Do intersect, can you please find such intersection point(s)?
- Equations for a line: (1) explicit representation; (2) implicit representation;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

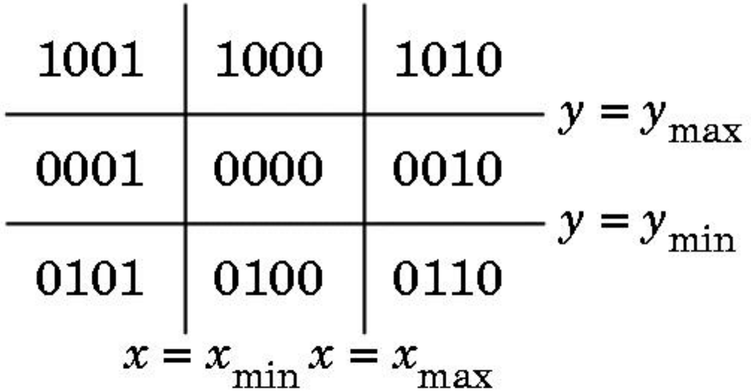
Clipping a Line Segment Against x_{min}

- 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 endpoints 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 $\text{tbrl}_1=\text{tbrl}_2=0$, simply **accept!**
  - – 2. If one is zero, one nonzero, compute an intercept. If necessary compute another intercept. Then **accept.**
  - – 3. If tbrl_1 & $\text{tbrl}_2 \neq 0$. If both outcodes are nonzero and the bitwise AND is nonzero, two endpoints lie on same outside side. **Simply reject!**
  - – 3. If tbrl_1 & $\text{rbrl}_2 = 0$. 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)

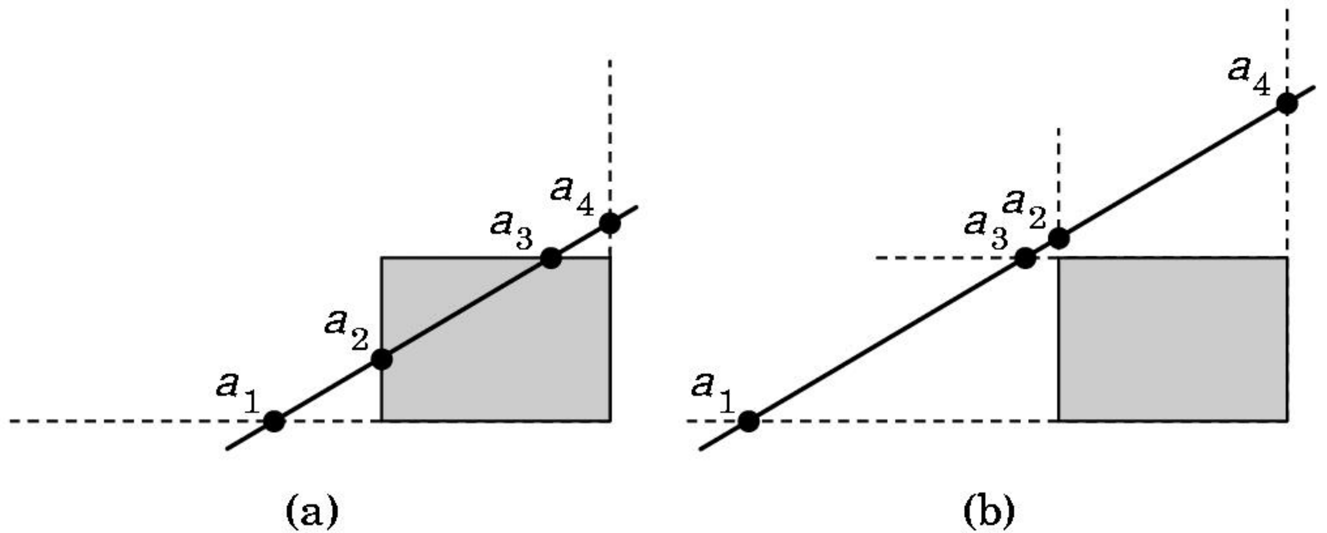
- In many cases, a few integer comparisons and Boolean operations suffice for simple reject or simple accept.
- This algorithm works best when there are many line segments, and most are clipped away
- But note that the $y=mx+h$ form of equation for a line doesn't work for vertical lines (this is actually 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 $0<a_1<a_3<a_2<a_4<1$?

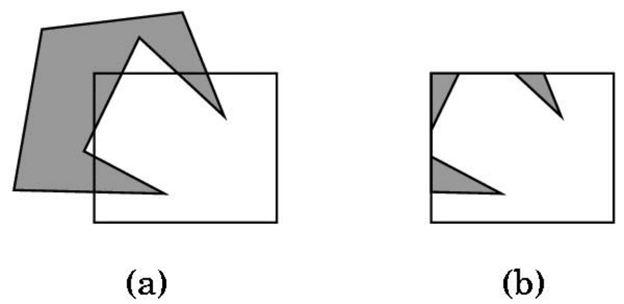


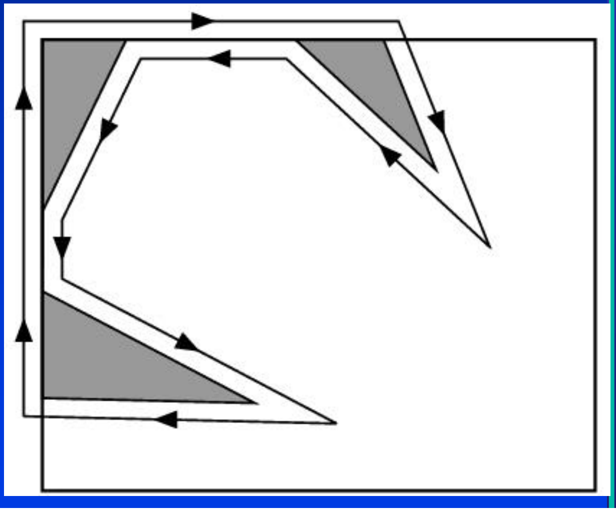
Computing Intersection Parameters

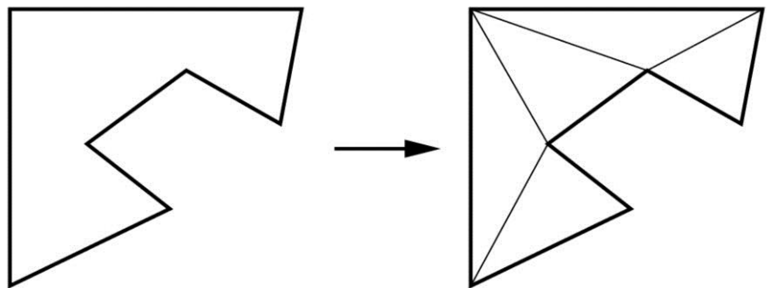
- Line-line intersection computation can be very costly.
- Hold off on computing parameters as long as possibly (lazy computation); many lines can be rejected early
- Could compute $a=(y_{max}-y_1)/(y_2-y_1)$
- Can rewrite $a\ (y_2-y_1) = (y_{max}-y_1)$
- Perform work in integer operations by comparing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

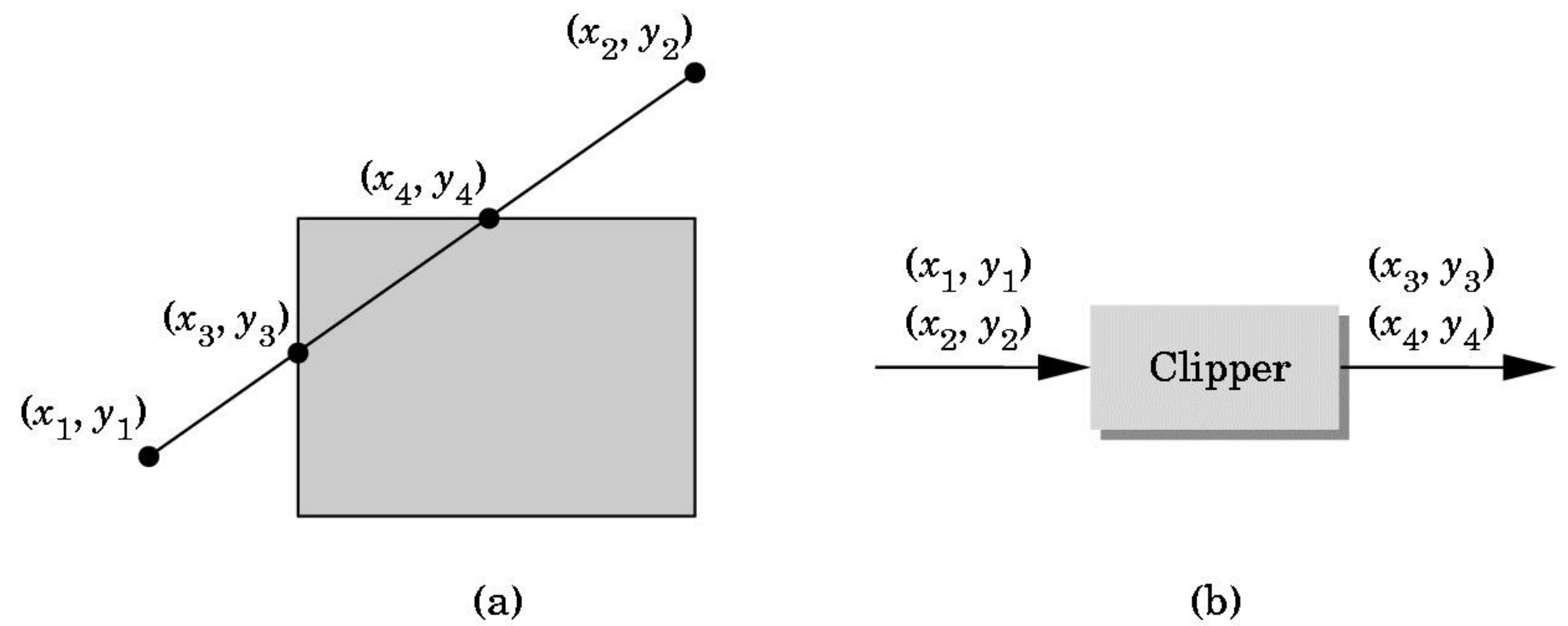






Sutherland-Hodgeman Polygon Clipping

- 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

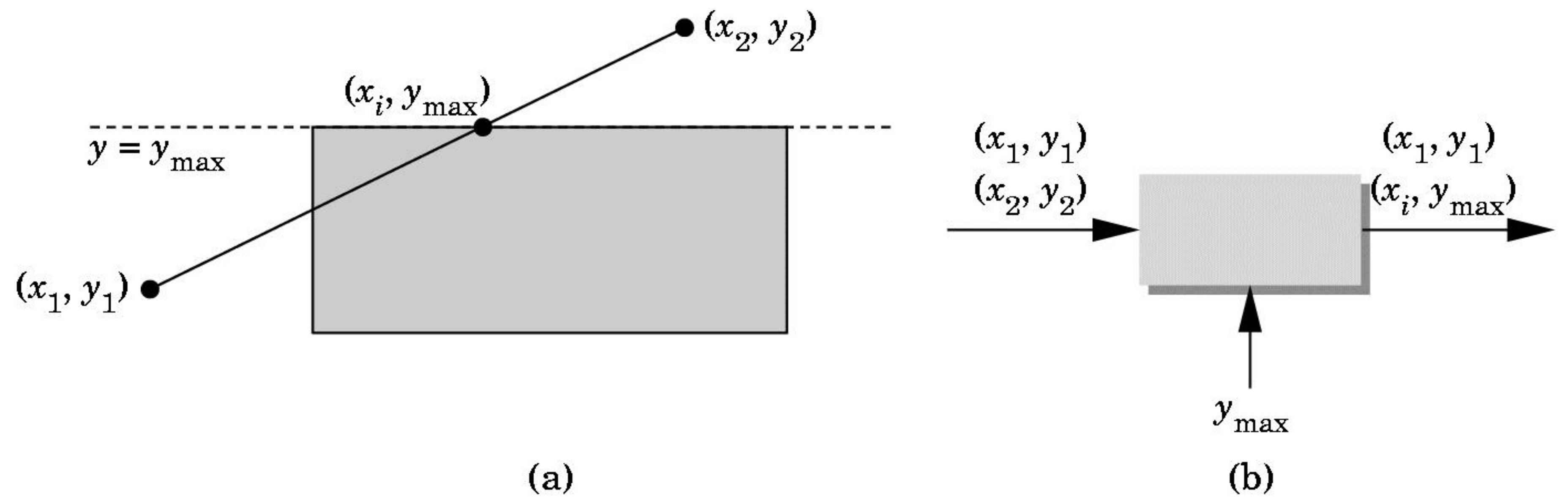


(a)

(b)

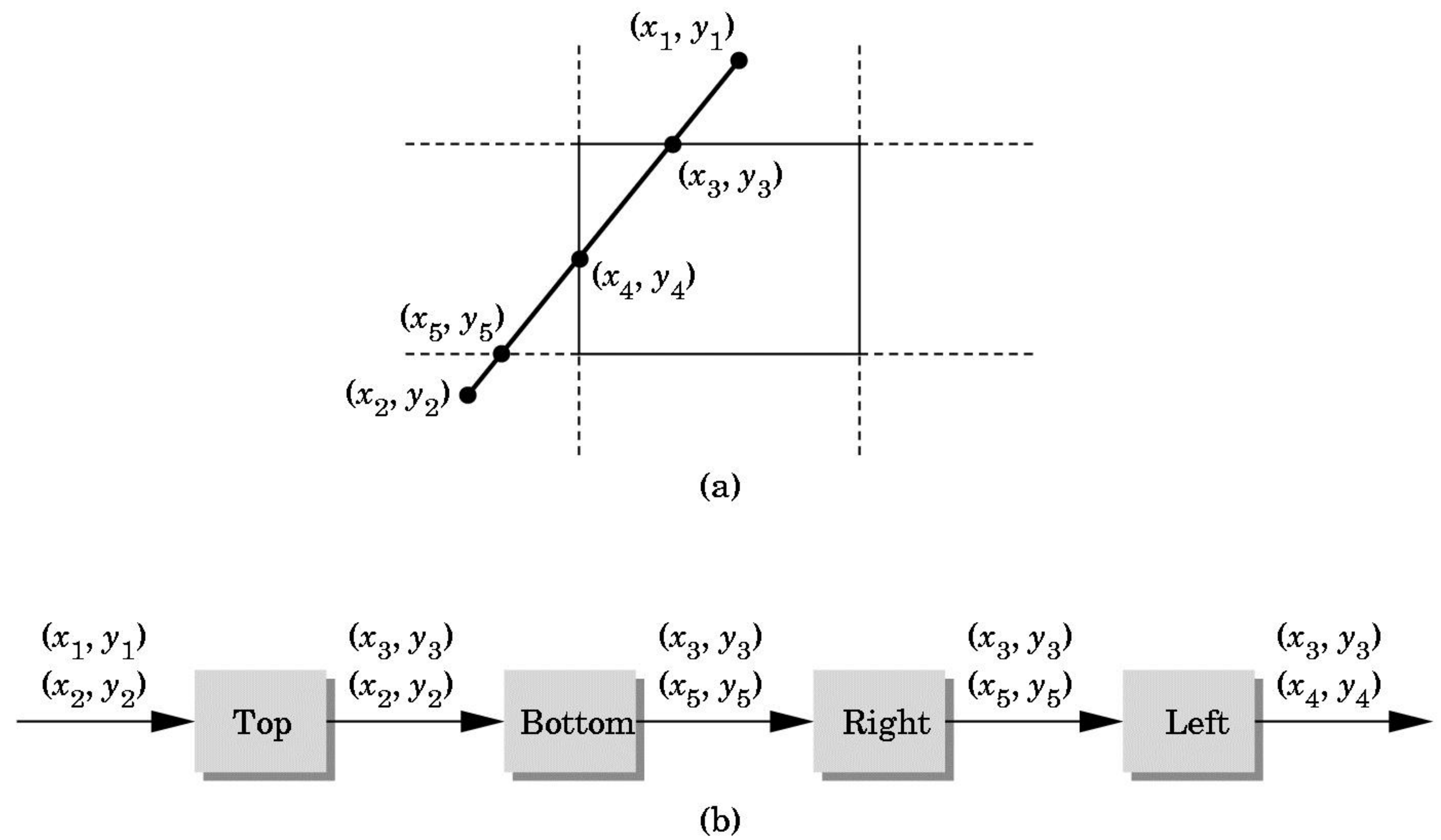
Clip Against Each Boundary

- 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

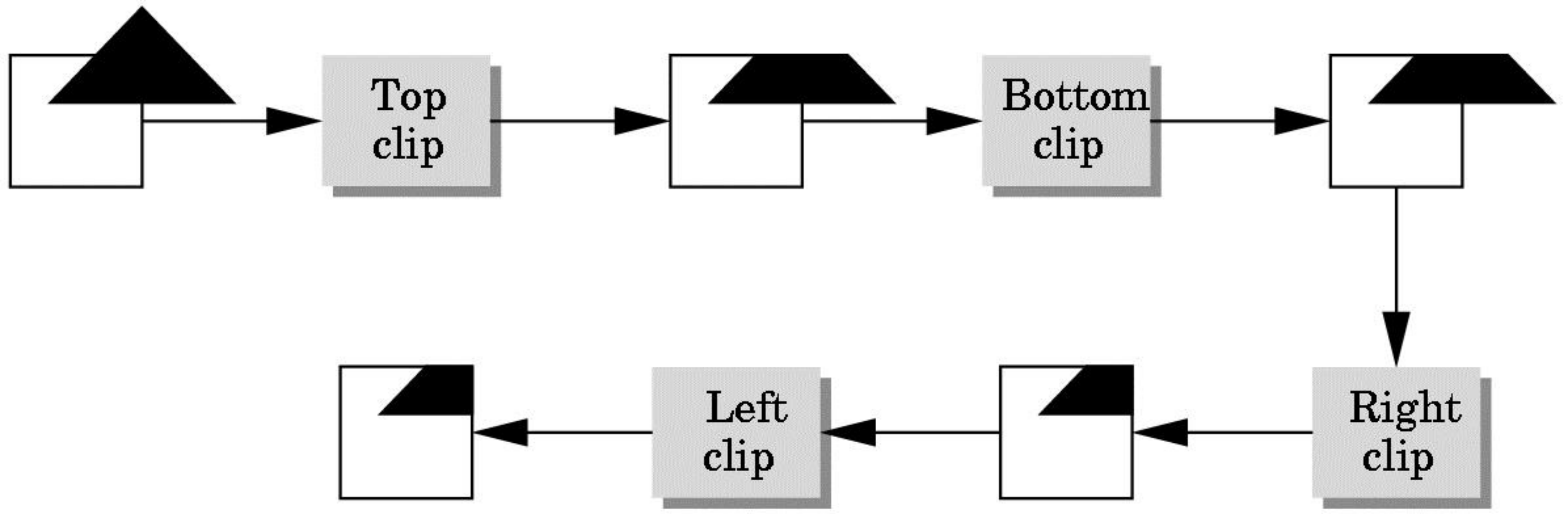


(a)

(b)

Clipping in Hardware

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

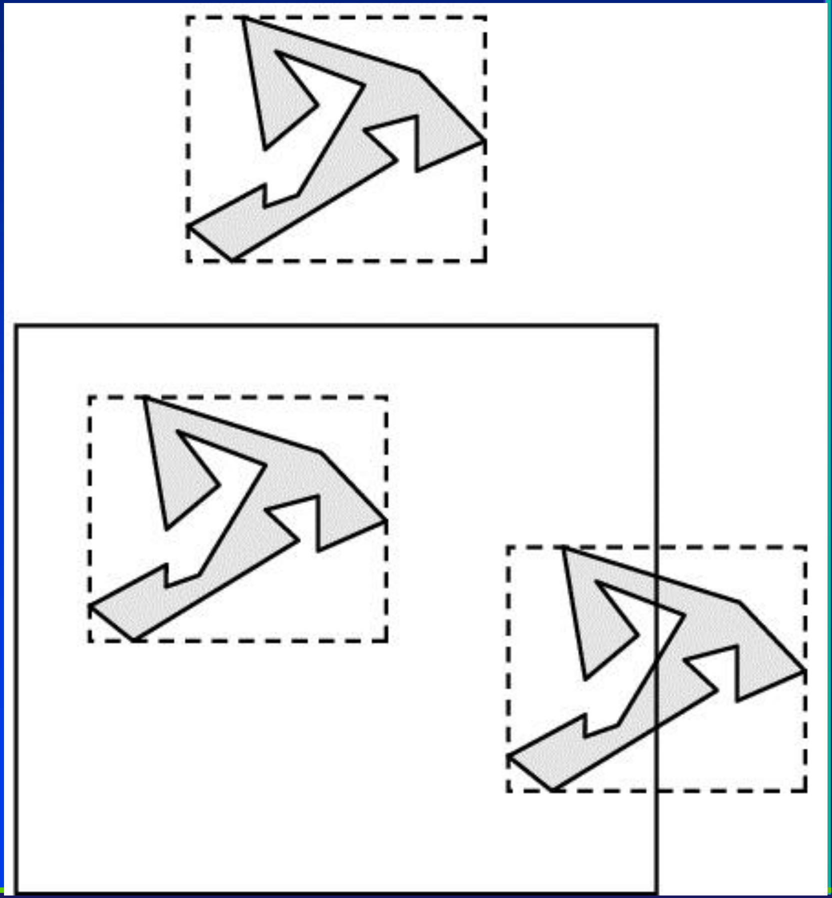


Clipping complicated objects

- 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?

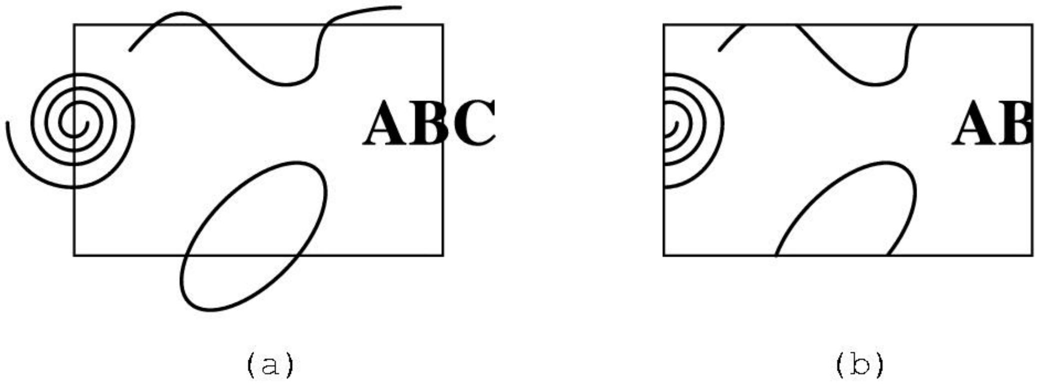
Clipping Other Primitives

- 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

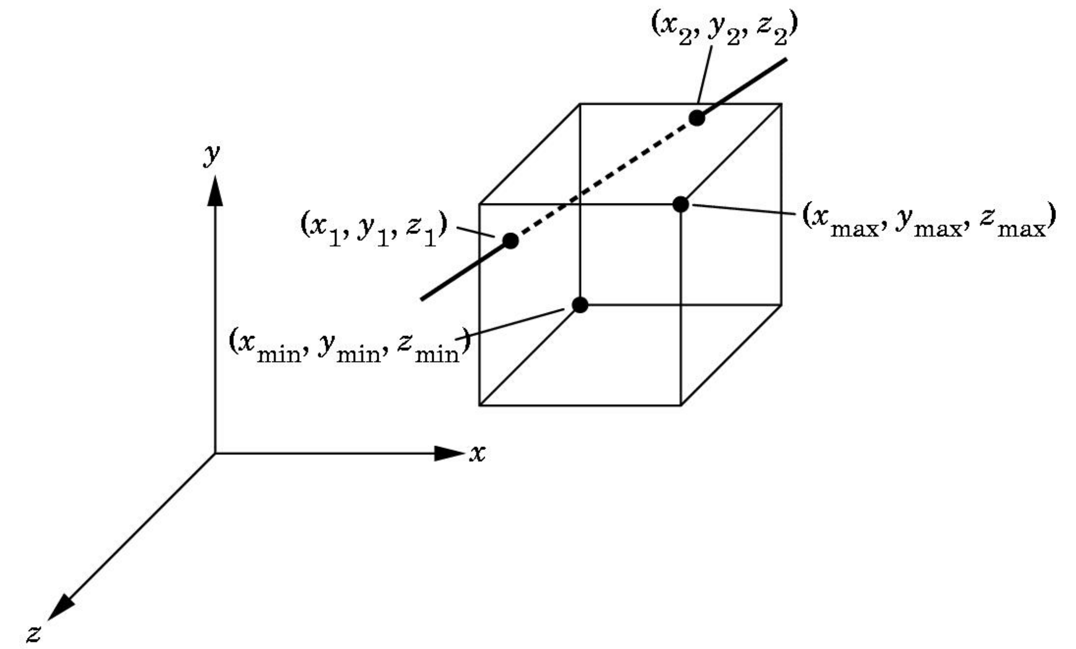
- Some shapes are so complex that they are difficult to clip analytically



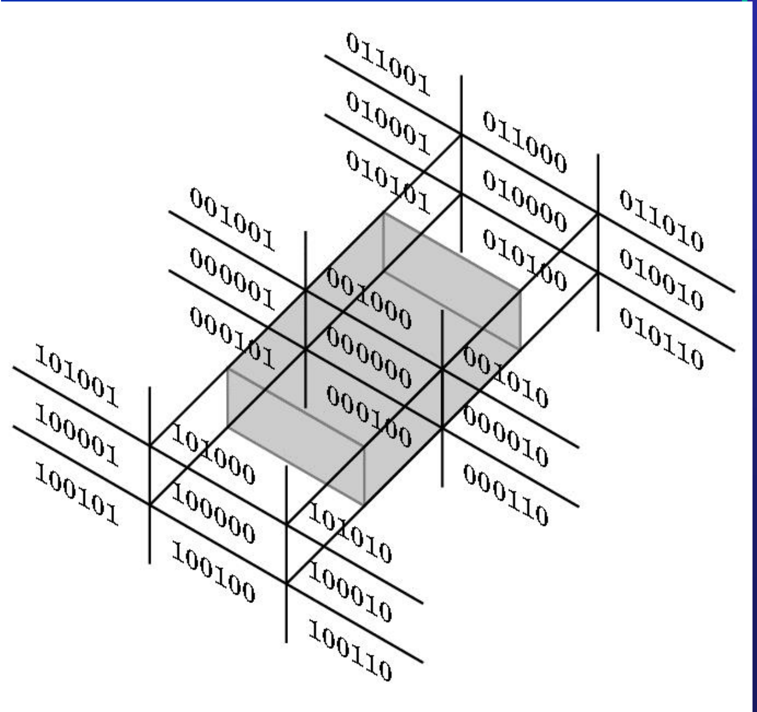
- Can approximate with line segments
- Can allow the clipping to occur in the frame buffer (pixels outside the screen rectangle aren’t drawn)
- Called “*scissoring*”
- How does performance compare with others?

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

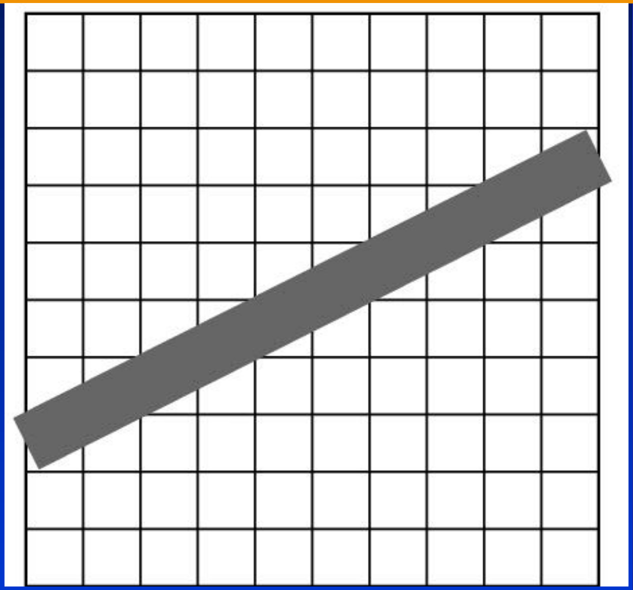
- RAM DAC converts frame buffer to video signal
- Other considerations:
  - Color correction
  - Antialiasing

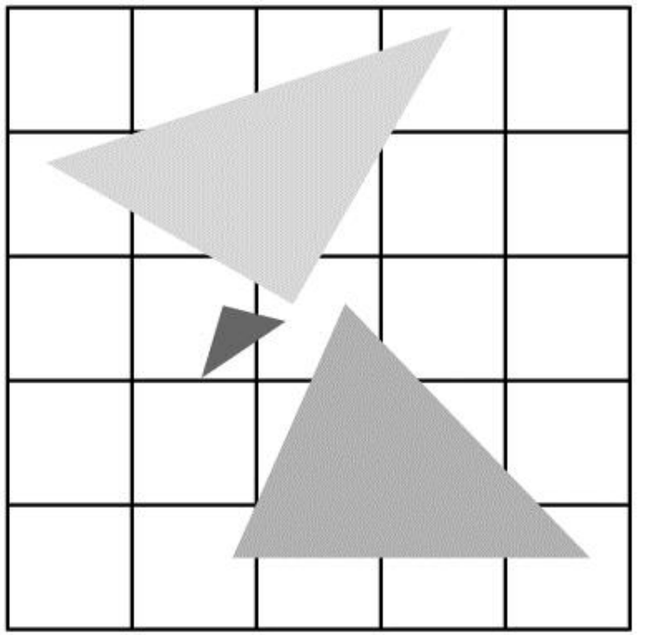
Implementation Strategies

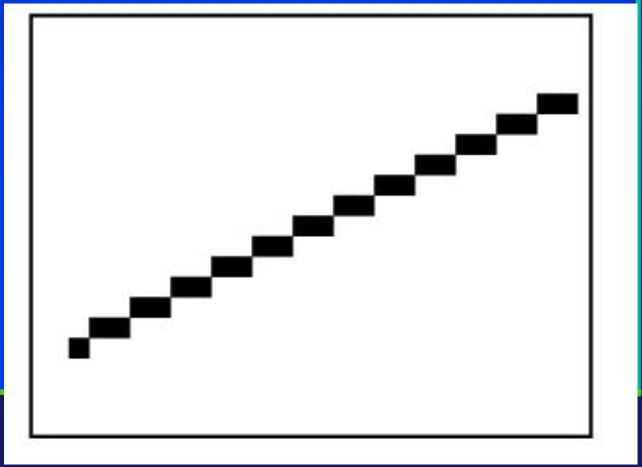
- Major approaches:
  - Object-oriented approach (pipeline renderers like OpenGL)
    - For each primitive, convert to pixels
    - Hidden-surface removal happens at the end
  - Image-oriented approach (e.g. ray tracing)
    - For each pixel, figure out what color to make it
    - Hidden-surface removal 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

Aliasing

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

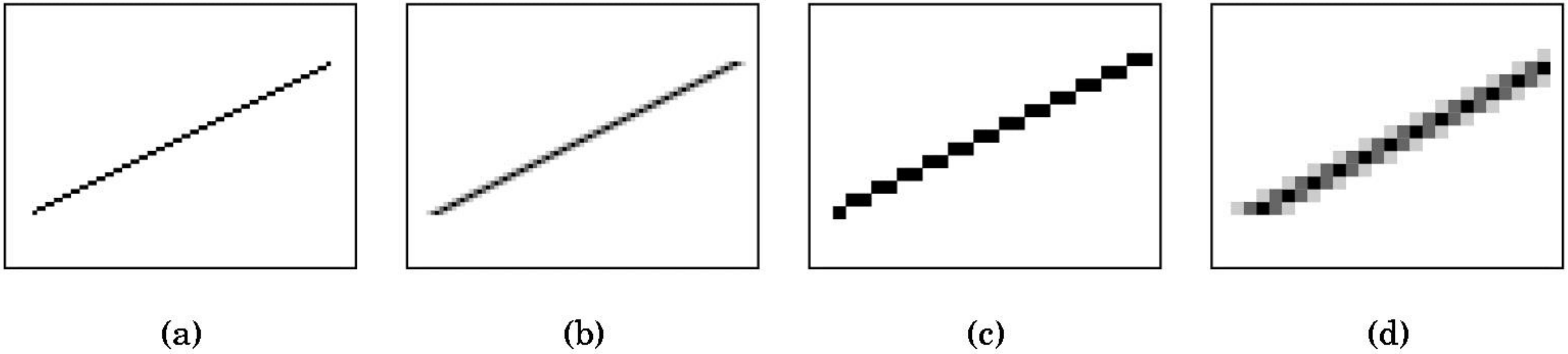






Anti-Aliasing

- 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



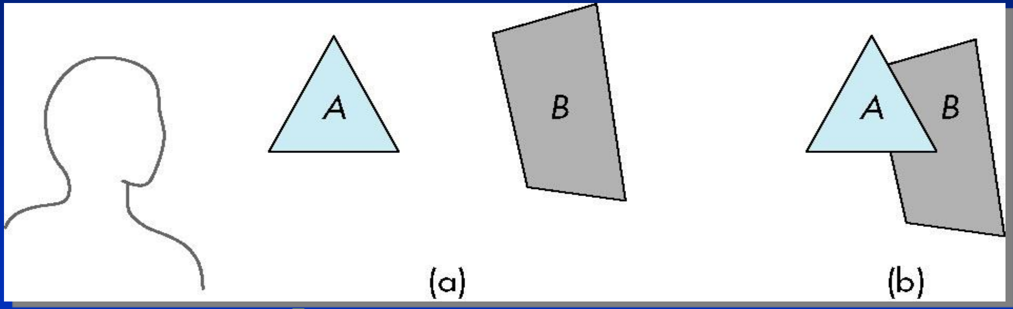
Hidden Surface Removal

- 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

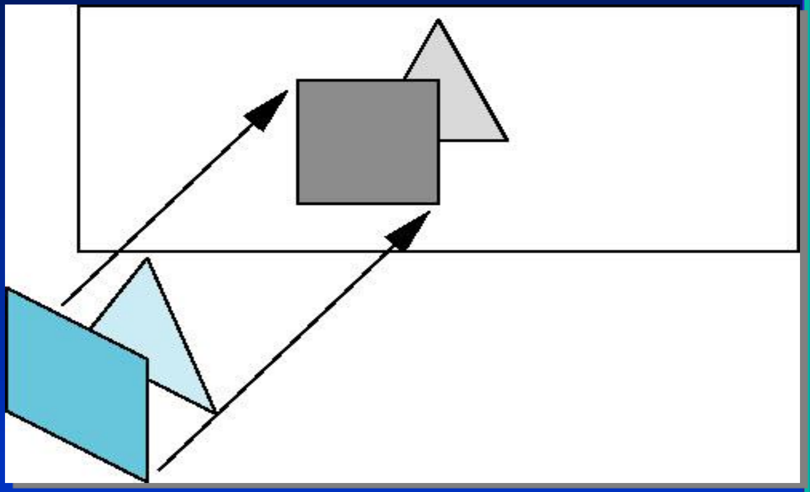
- The *painter's algorithm:* draw from back to front

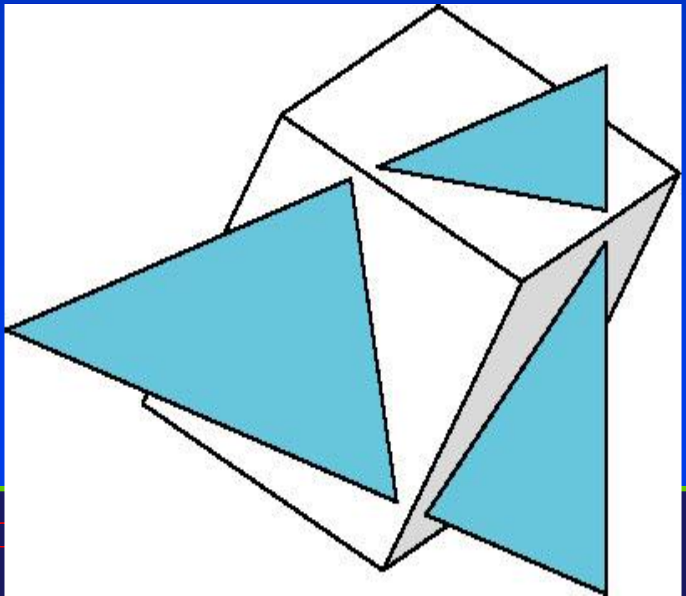


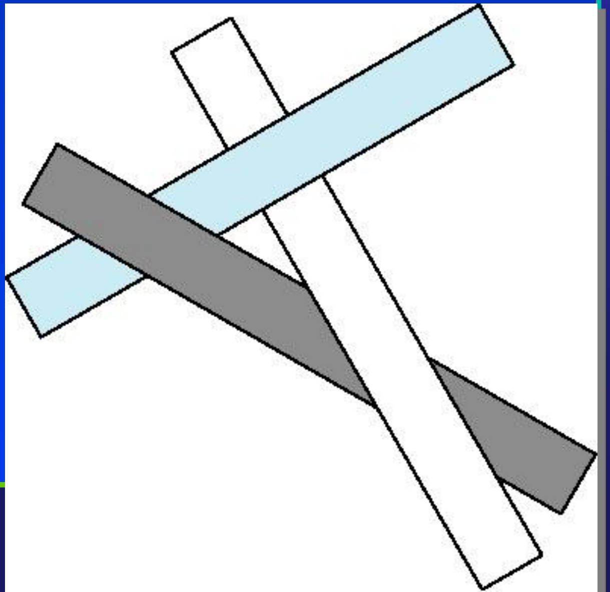
- Depth-sort hidden surface removal:
  - sort display list by z-coordinate from back to front
  - render/display
- Drawbacks
  - it takes some time (especially with bubble sort!)
  - it doesn't work

Depth-Sort Difficulties

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

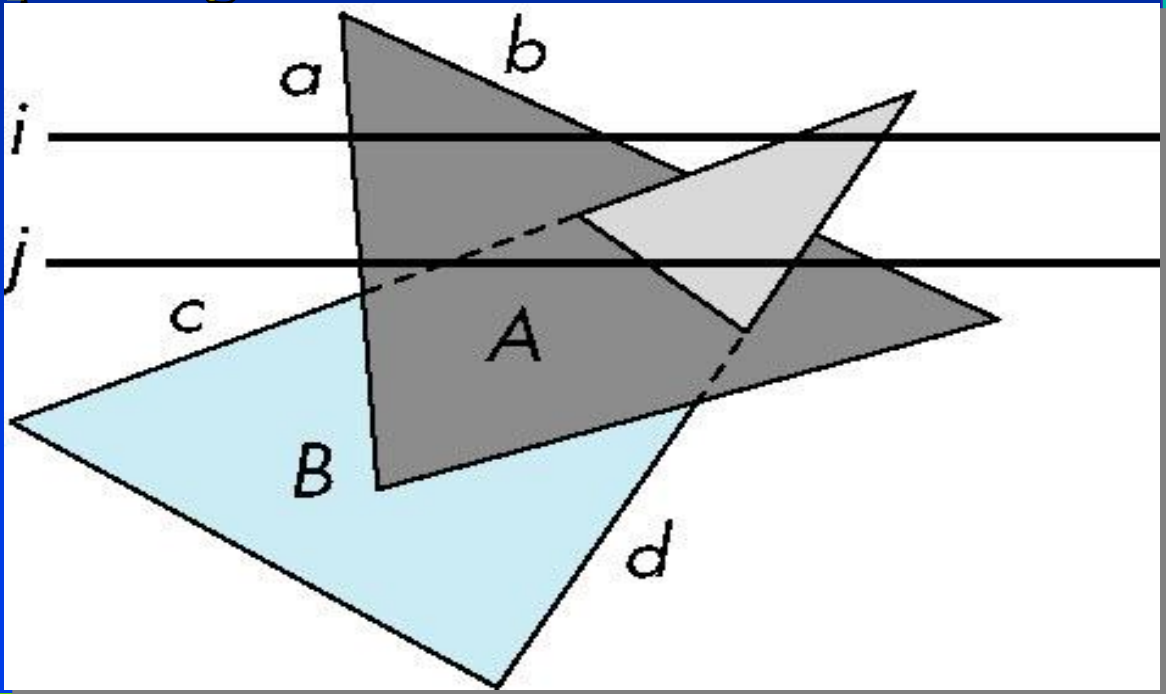






Scan-line Algorithm

- 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

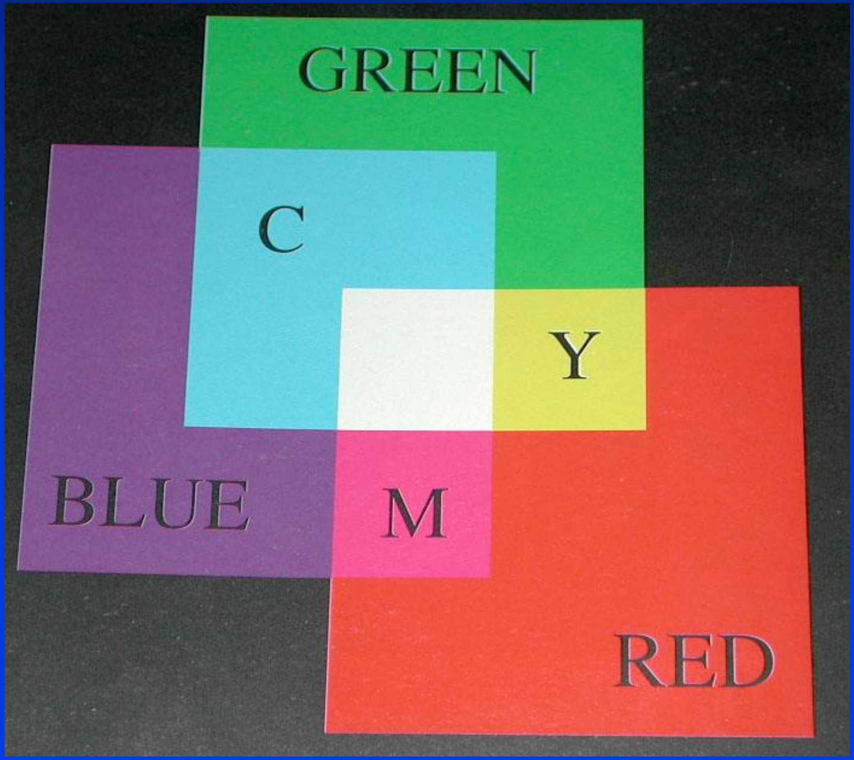
Temporal Aliasing

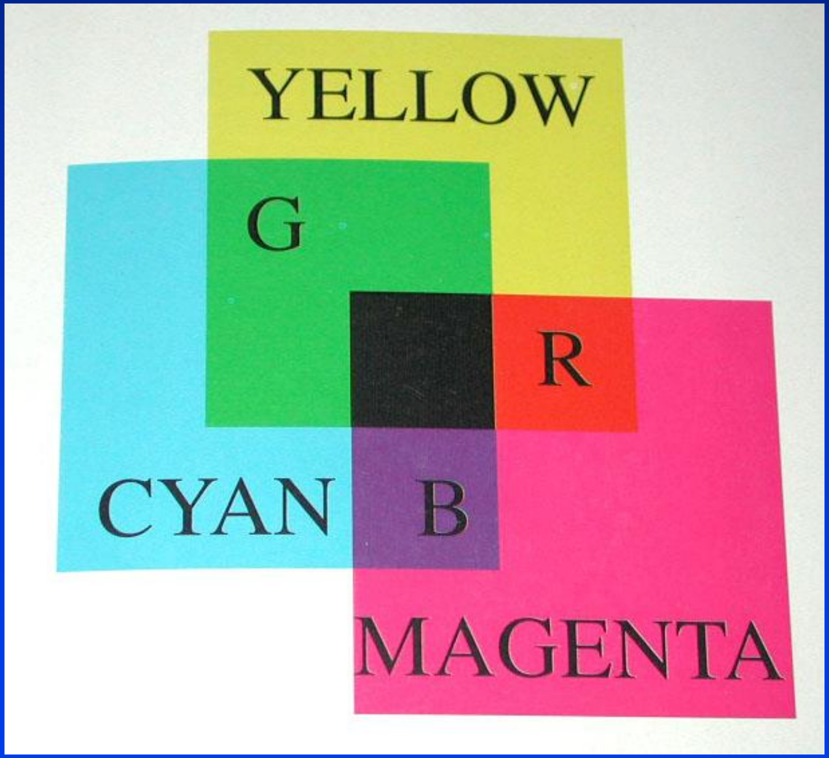
- Need *motion blur* for motion that doesn't flicker at slow frame rates
- Common approach: *temporal supersampling*
  - render images at several times within frame time interval
  - average results

Display Considerations

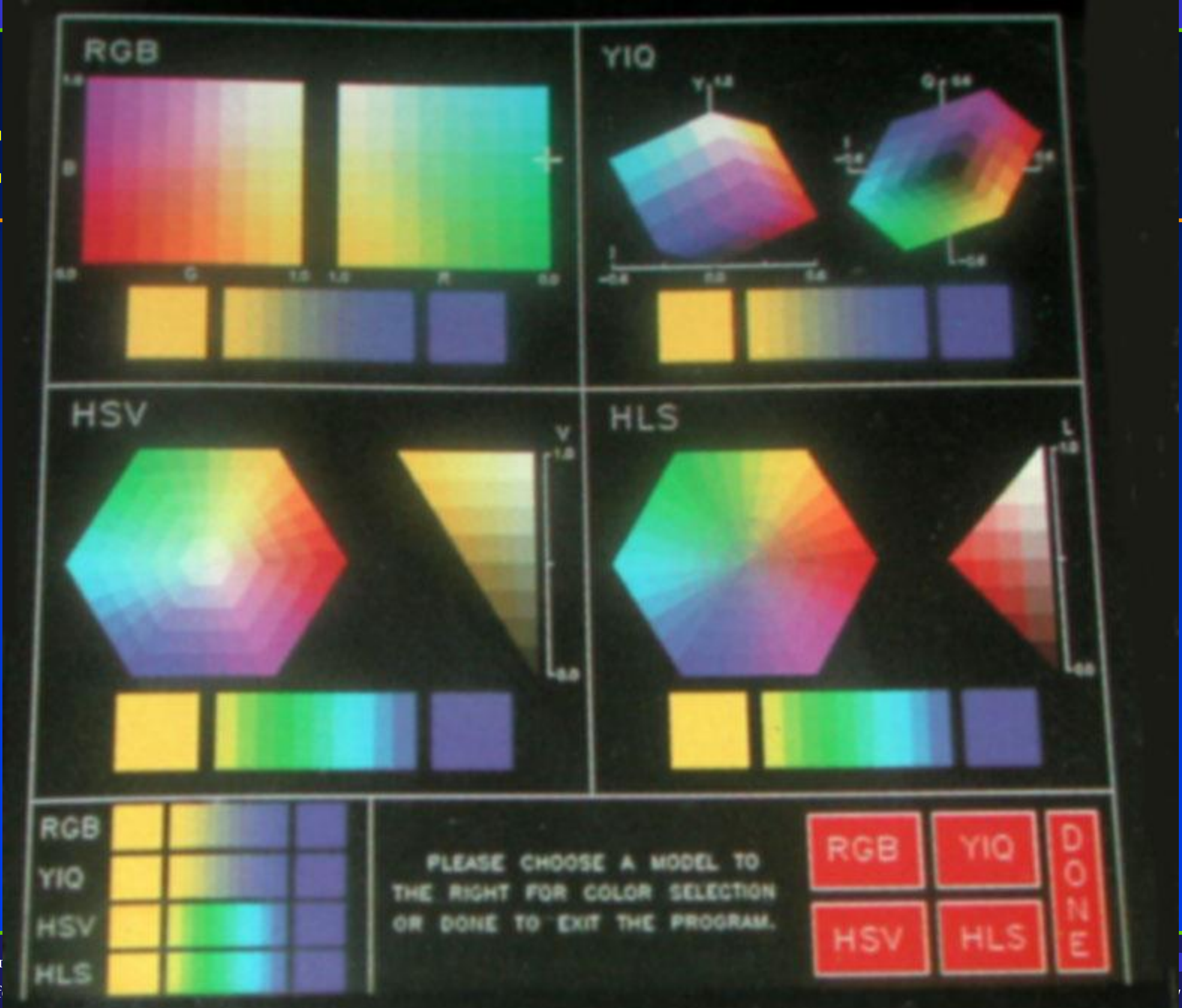
- Color systems
- Color quantization
- Gamma correction
- Dithering and Halftoning

Additive and Subtractive Color





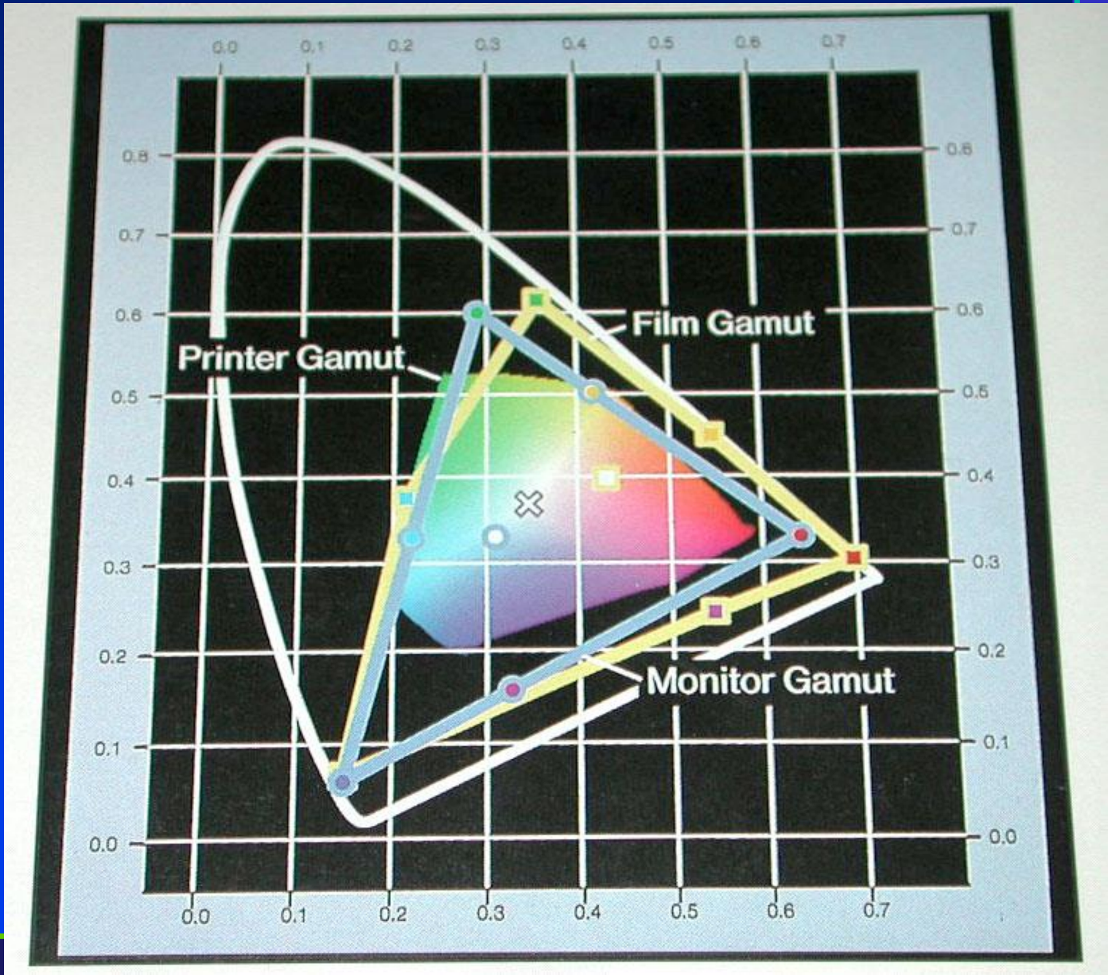
C



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

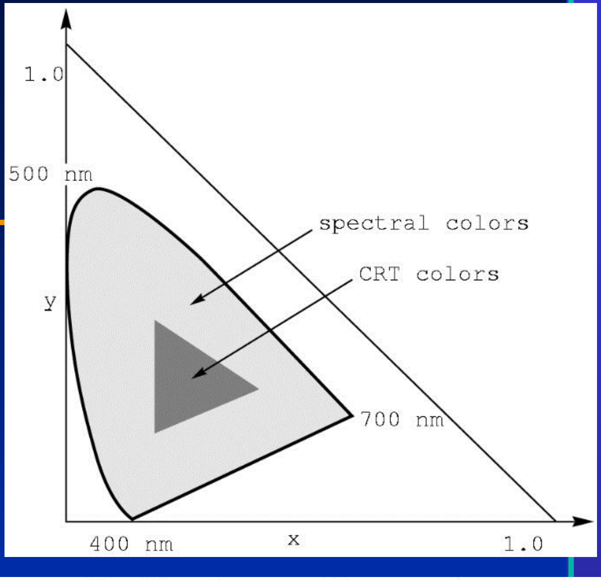
Color Systems

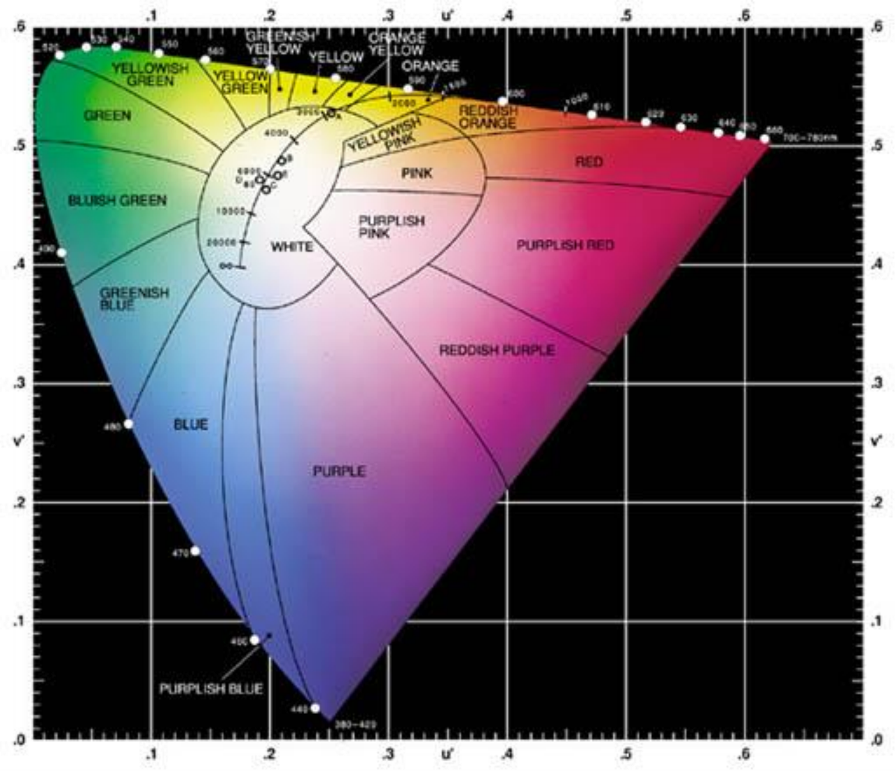
- RGB
- YIQ
- CMYK
- HSV, HLS
- Chromaticity
- Color gamut



Chromaticity

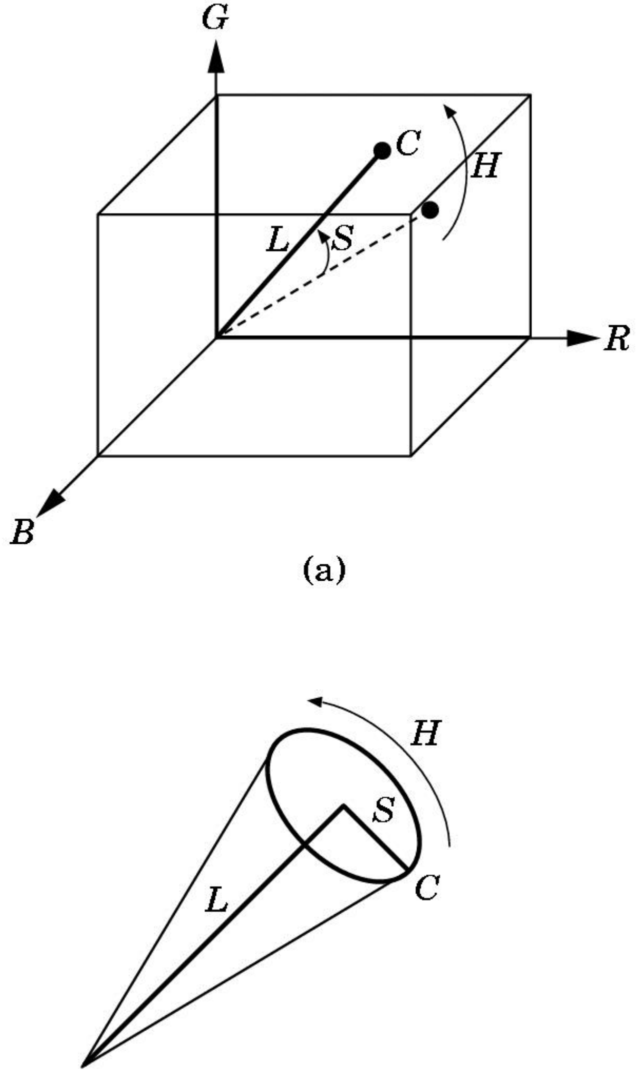
- Tri-stimulus values: R, G, B values that we know of
- Color researchers often prefer chromaticity coordinates:
  - t1 = T1 / (T1 + T2 + T3)
  - t2 = T2 / (T1 + T2 + T3)
  - t3 = T3 / (T1 + T2 + T3)
- Thus, t1+t2+t3 = 1.0.
- Use t1 and t2; t3 can be computed as 1-t1-t2
- Chromaticity diagram uses this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





HLS

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



(a)

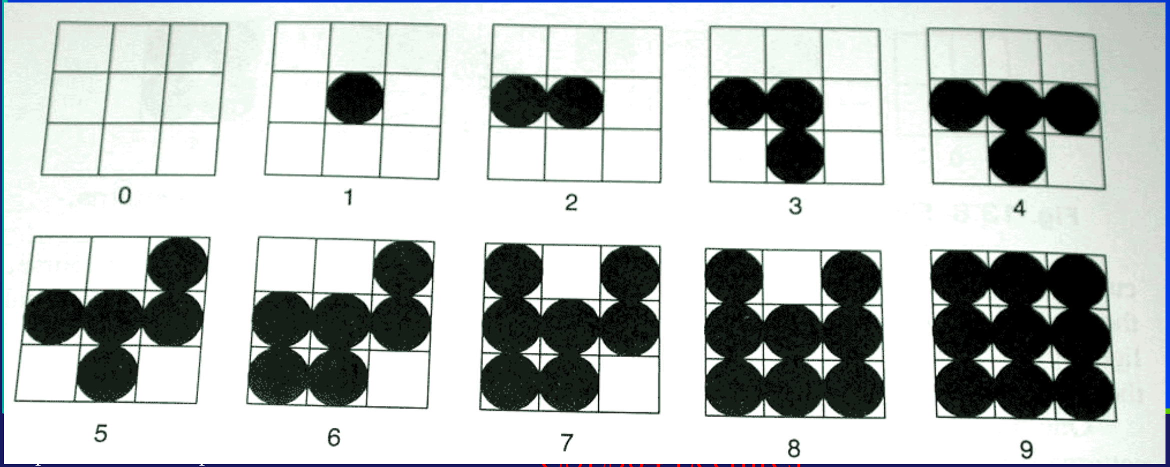
(b)

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?]

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)

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

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 alg)
  - Plan B: Divide the image into n regions with a pipeline for each region; send needed objects to each pipeline