# Rasterization

MIT EECS 6.837
Frédo Durand and Barb Cutler

# Last time?

- Point and segment Clipping
- Planes as homogenous vectors (duality)
- In homogeneous coordinates before division
- Outcodes for efficient rejection
- Notion of convexity
- Polygon clipping via walking
- Line rasterization, incremental computation

# High-level concepts for 6.837

- Linearity
- Homogeneous coordinates
- Convexity
- Discrete vs. continuous
- Incremental computation

- Trying things on simple examples

# Scan Conversion (Rasterization)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

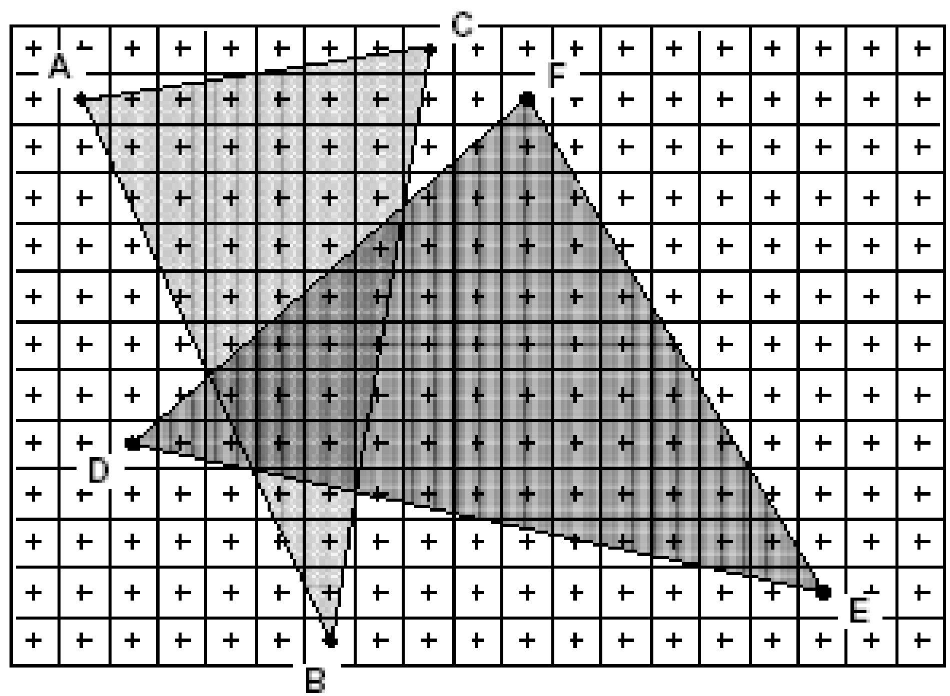
Clipping

Projection (to Screen Space)

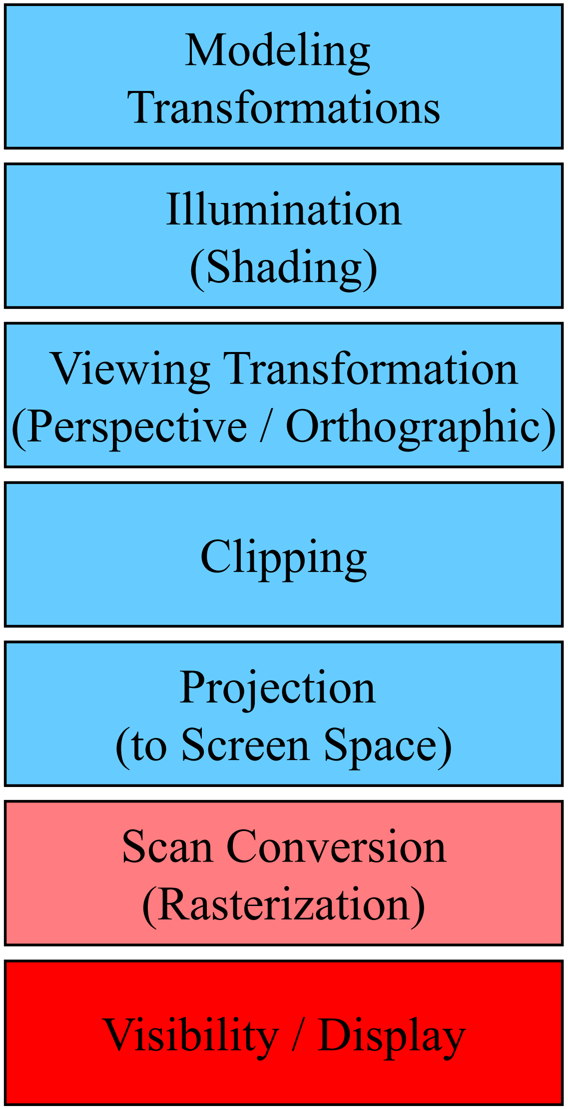
**Scan Conversion (Rasterization)**

Visibility / Display

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)



# Visibility / Display

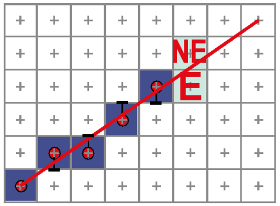



- Each pixel remembers the closest object (depth buffer)

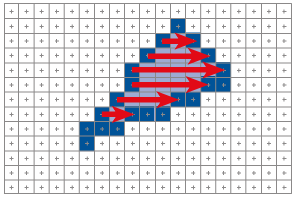
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

# Today

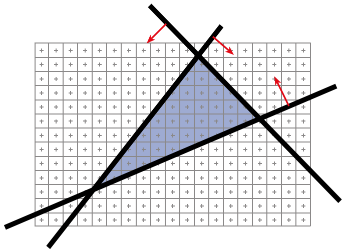
- Line scan-conversion




- Polygon scan conversion

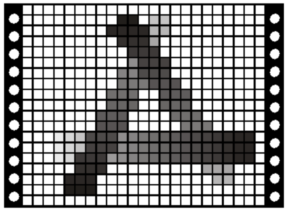


  - smart



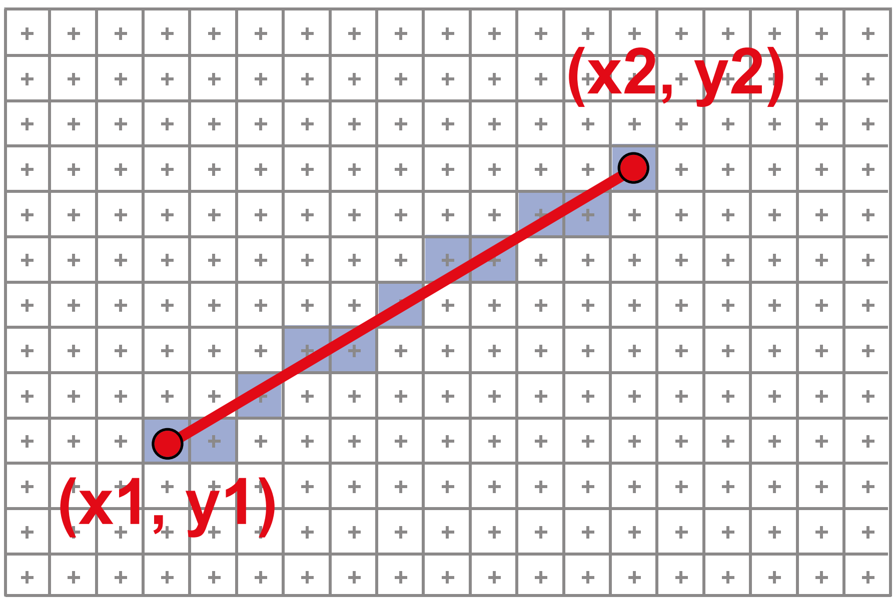
  - back to brute force

- Visibility



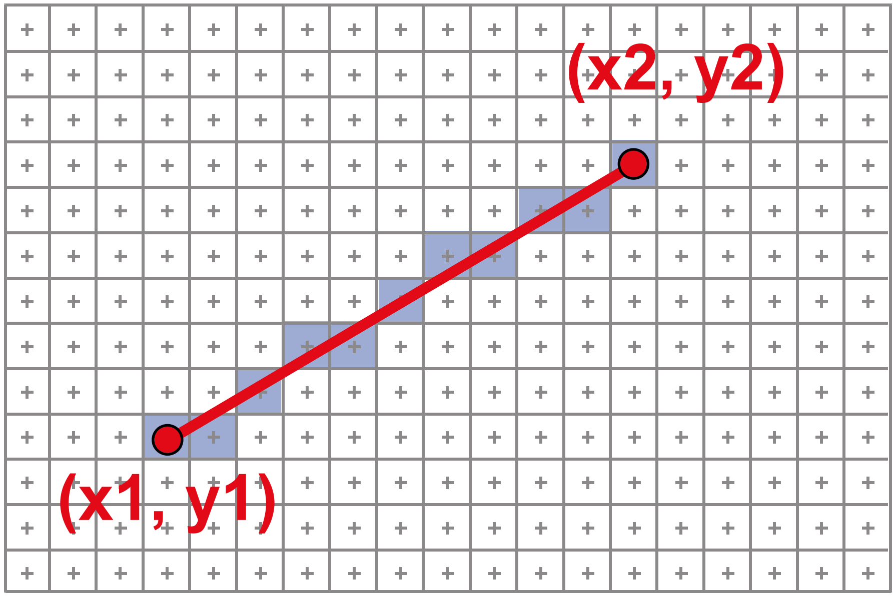
# Scan Converting 2D Line Segments

- Given:
  - Segment endpoints (integers x1, y1; x2, y2)
- Identify:
  - Set of pixels (x, y) to display for segment



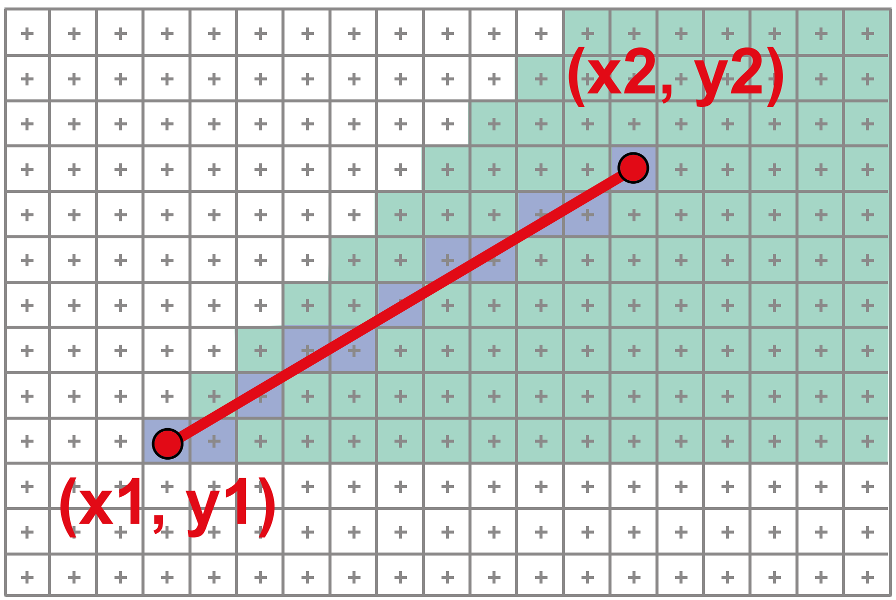
# Line Rasterization Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed



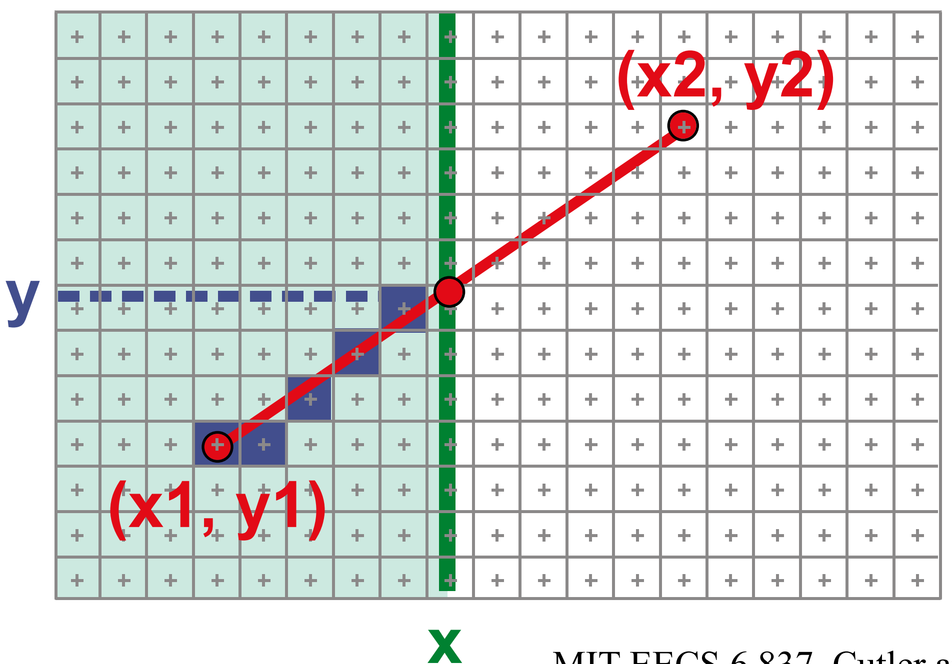
# Algorithm Design Choices

- Assume:
  - $m = dy/dx, \ \ 0 < m < 1$
- Exactly one pixel per column
  - fewer → disconnected, more → too thick



# Naive Line Rasterization Algorithm

- Simply compute y as a function of x
  - Conceptually: move vertical scan line from x1 to x2
  - What is the expression of y as function of x?
  - Set pixel (x, round (y(x)))




$$y = y1 + \frac{x - x1}{x2 - x1}(y2 - y1)$$

$$= y1 + m(x - x1)$$

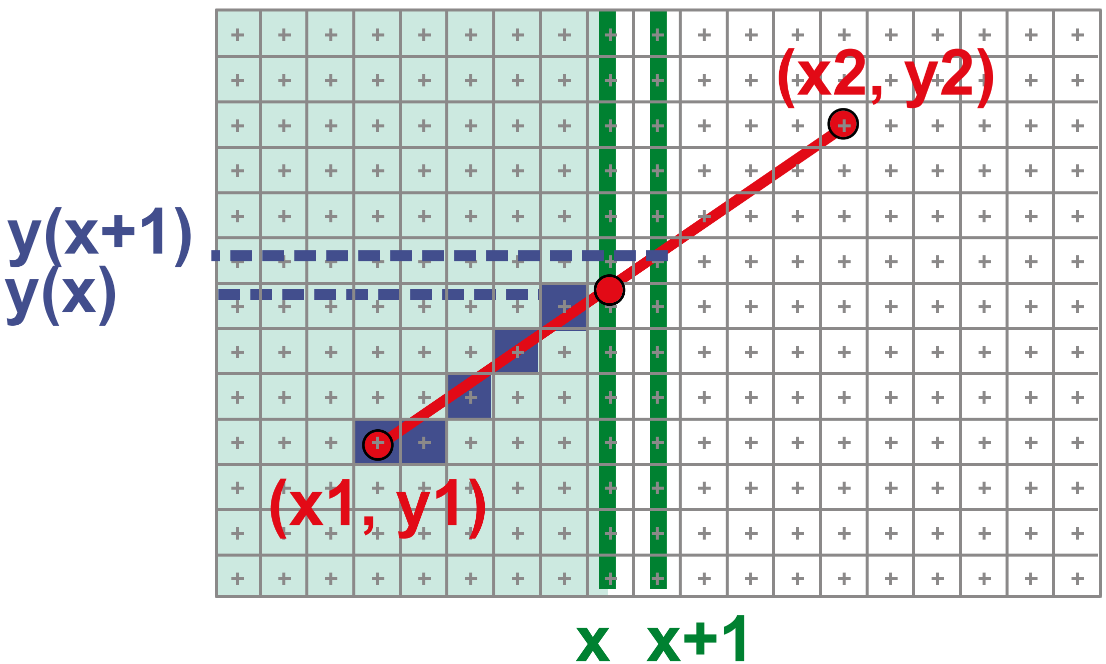
$$m = \frac{dy}{dx}$$

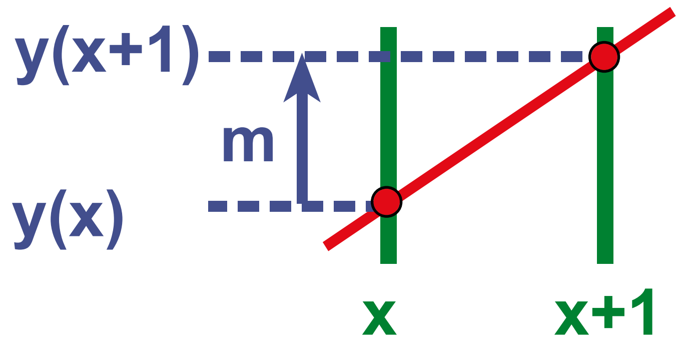
# Efficiency

- Computing y value is expensive

$$y = y1 + m(x - x1)$$

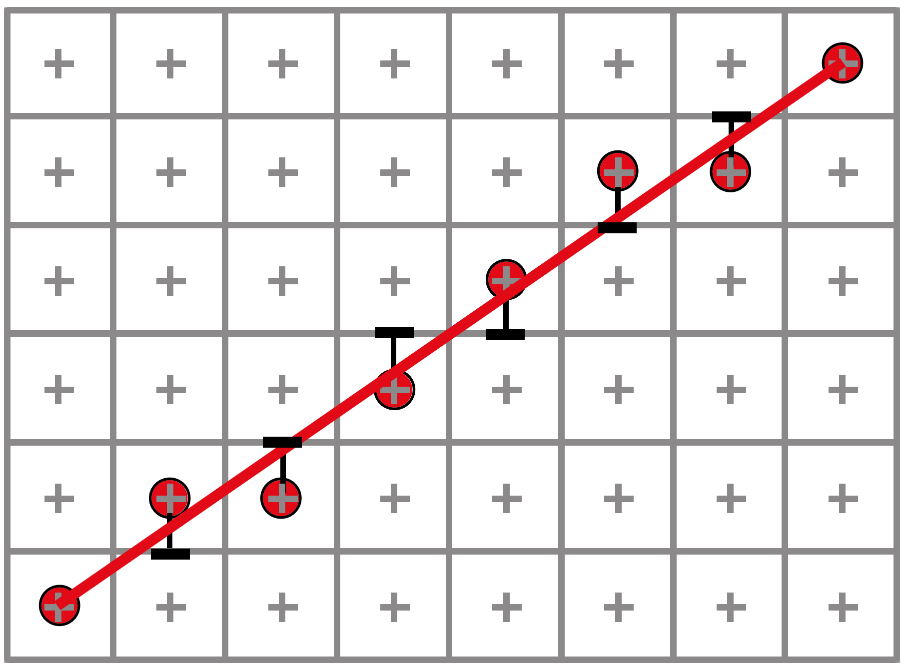
- Observe: $y \mathrel{+}= m$ at each $x$ step ($m = dy/dx$)



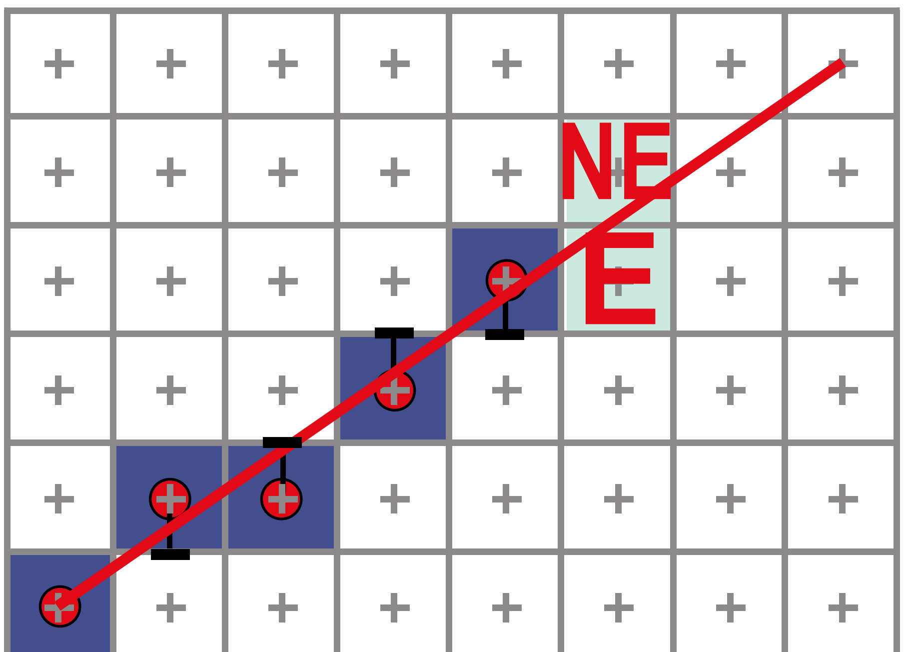
# Bresenham's Algorithm (DDA)

- Select pixel vertically closest to line segment
  - intuitive, efficient,
    pixel center always within 0.5 vertically
- Same answer as naive approach



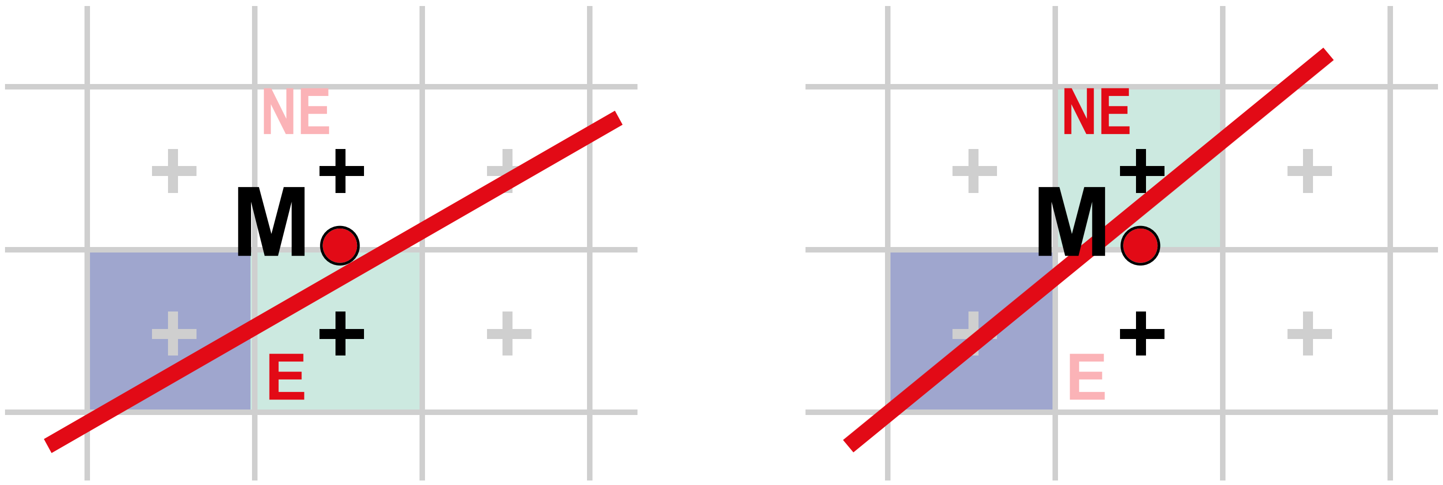
# Bresenham's Algorithm (DDA)

- Observation:
  - If we're at pixel P ($x_p$, $y_p$), the next pixel must be either E ($x_p$+1, $y_p$) or NE ($x_p$+*1*, $y_p$+1)
  - Why?



# Bresenham Step

- Which pixel to choose: E or NE?
  - Choose E if segment passes below or through middle point M
  - Choose NE if segment passes above M

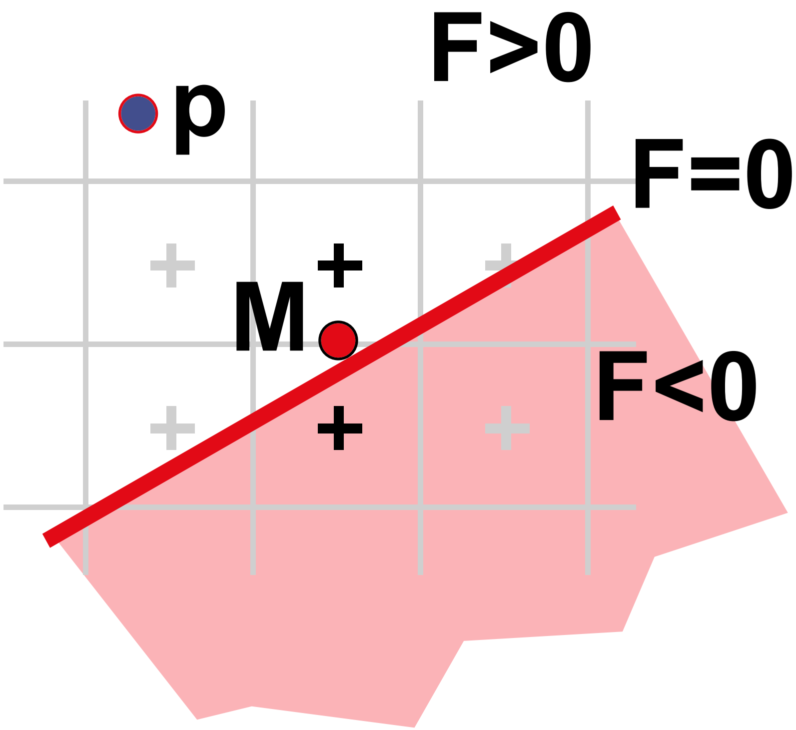


# Bresenham Step

- Use *decision function* D to identify points underlying line L:

$D(x, y) = y\text{-}mx\text{-}b$

  - positive above L
  - zero on L
  - negative below L

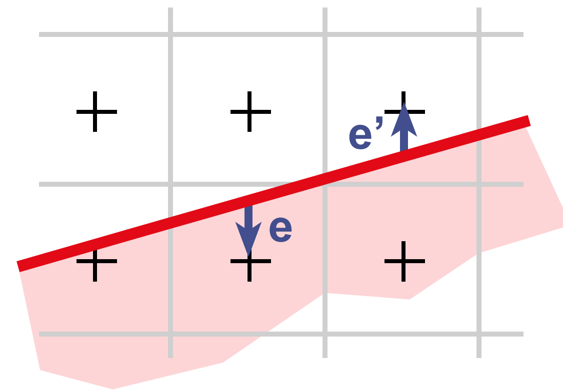



$D(p_x, p_y) =$ vertical distance from point to line

# Bresenham's Algorithm (DDA)

- Decision Function:

  $D(x, y) = y - mx - b$




- Initialize:

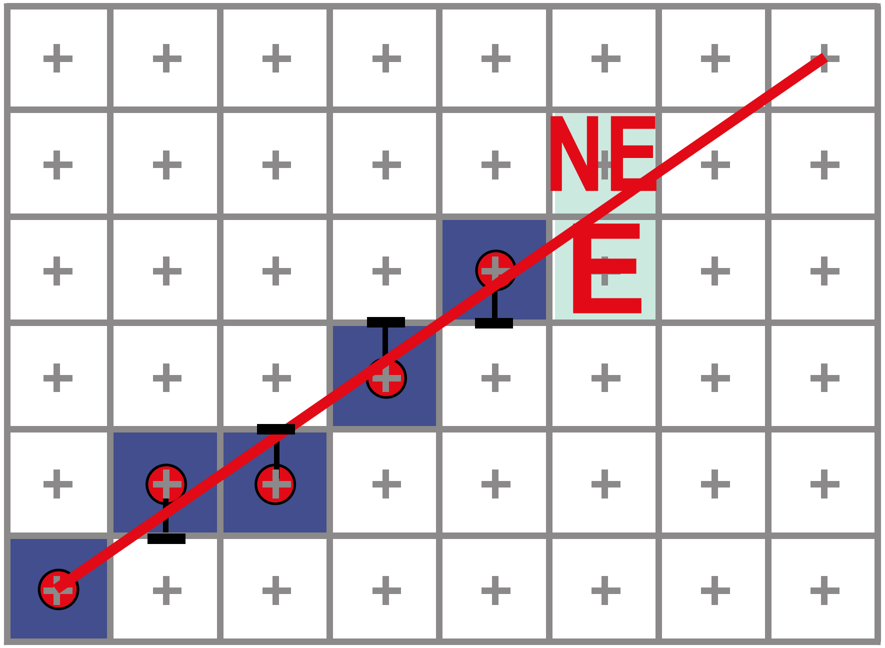
  error term $e = -D(x,y)$

- On each iteration:

  update $x$: $x' = x+1$
  update $e$: $e' = e + m$
  if ($e \leq 0.5$): $y' = y$ (choose pixel E)
  if (e > 0.5): y' = y + (choose pixel NE) e' = e - 1

# Summary of Bresenham

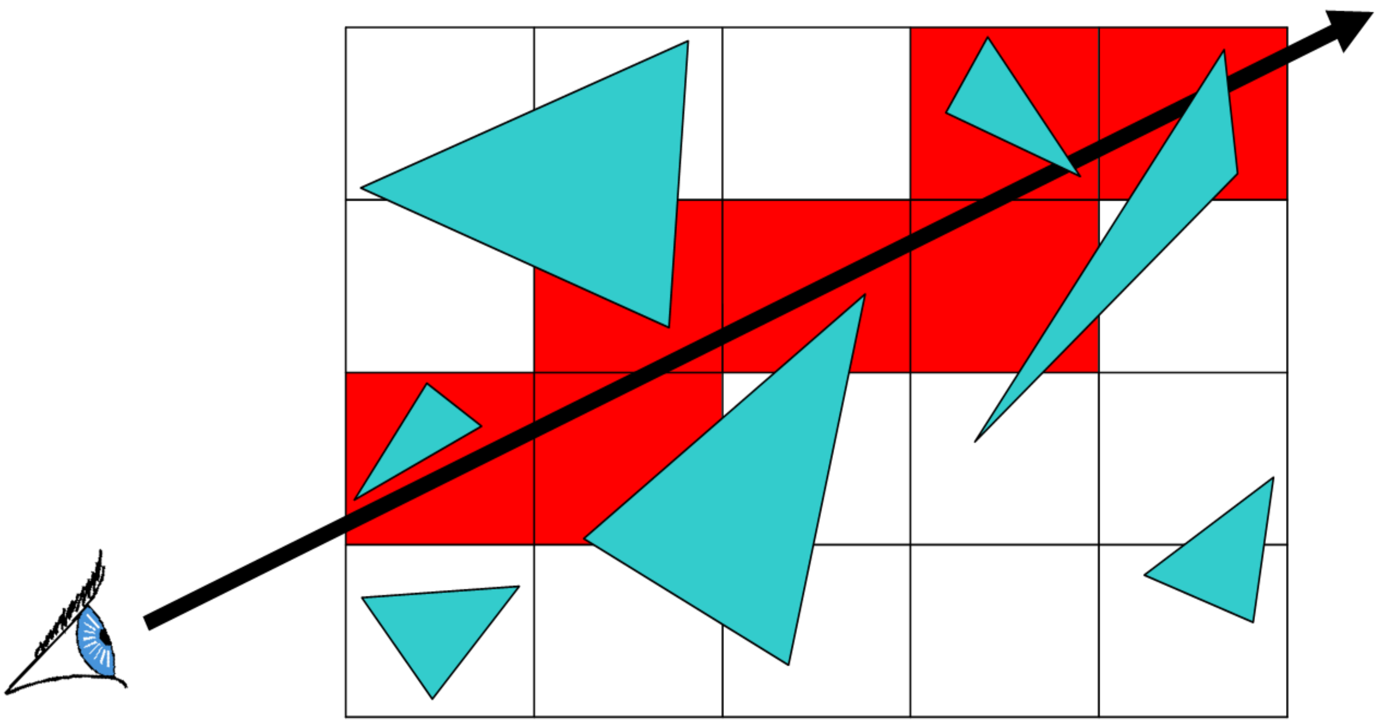
- initialize $x$, $y$, $e$
- for ($x$ = x1; $x \le$ x2; $x$++)
  - plot ($x$,$y$)
  - update $x$, $y$, $e$



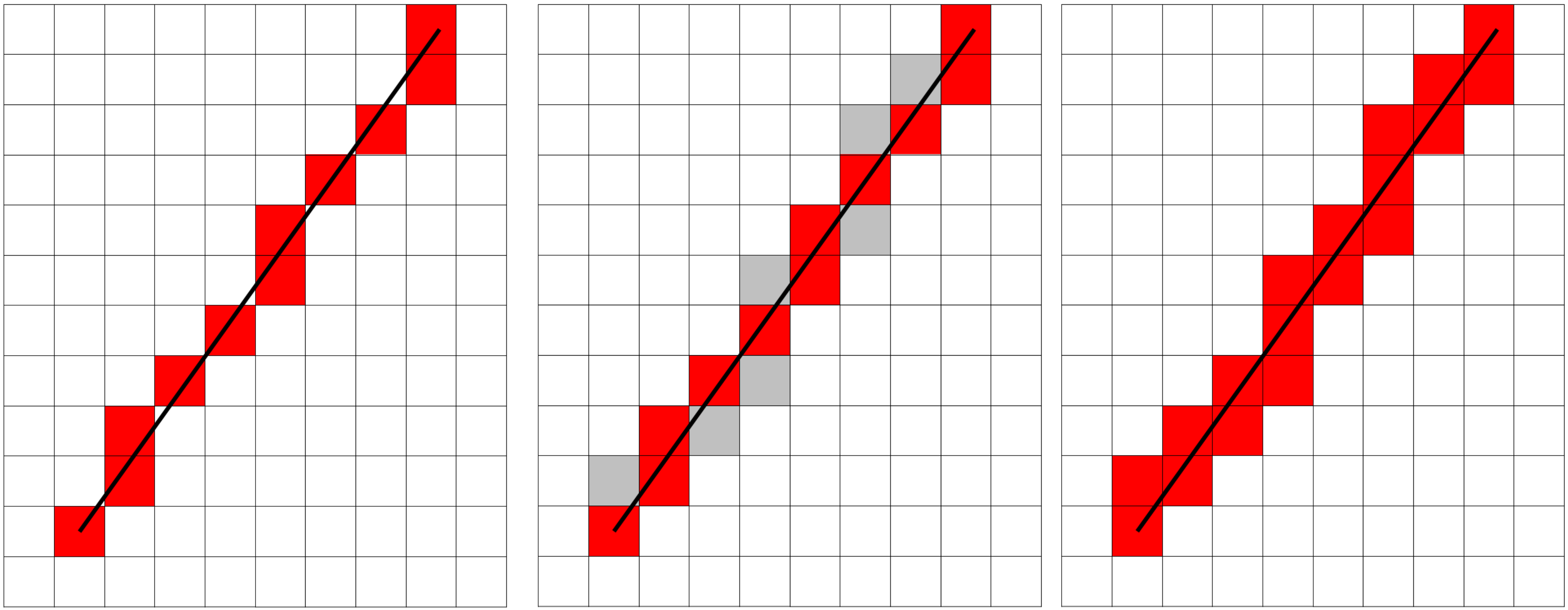

- Generalize to handle all eight octants using symmetry
- Can be modified to use only integer arithmetic

# Line Rasterization

- We will use it for ray-casting acceleration
- March a ray through a grid



# Line Rasterization vs. Grid Marching



Line Rasterization:

Best discrete approximation of the line

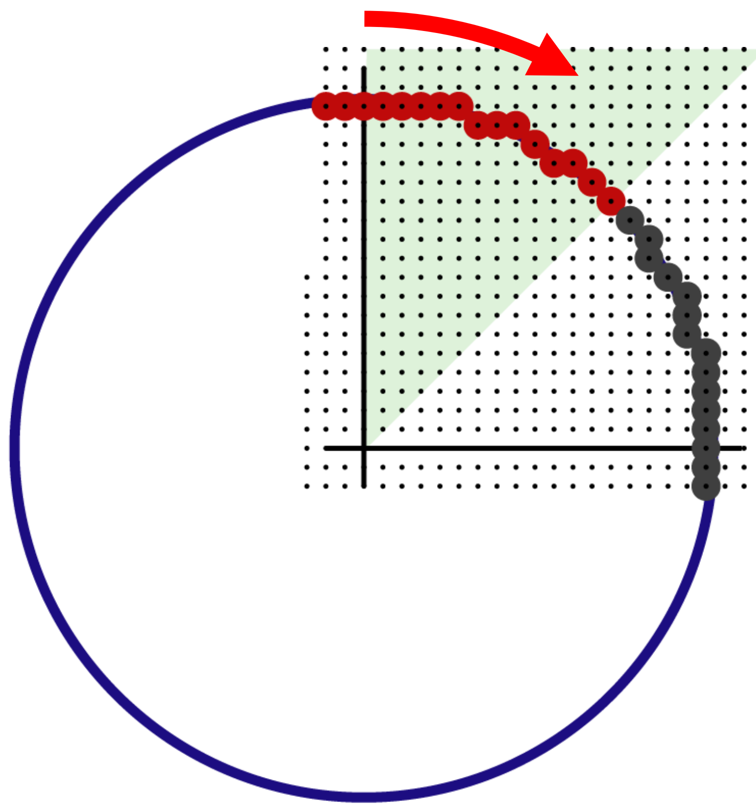
Ray Acceleration:

Must examine every cell the line touches

# Questions?

# Circle Rasterization

- Generate pixels for 2nd octant only
- Slope progresses from 0 $\rightarrow$ –1
- Analog of Bresenham Segment Algorithm



# Circle Rasterization

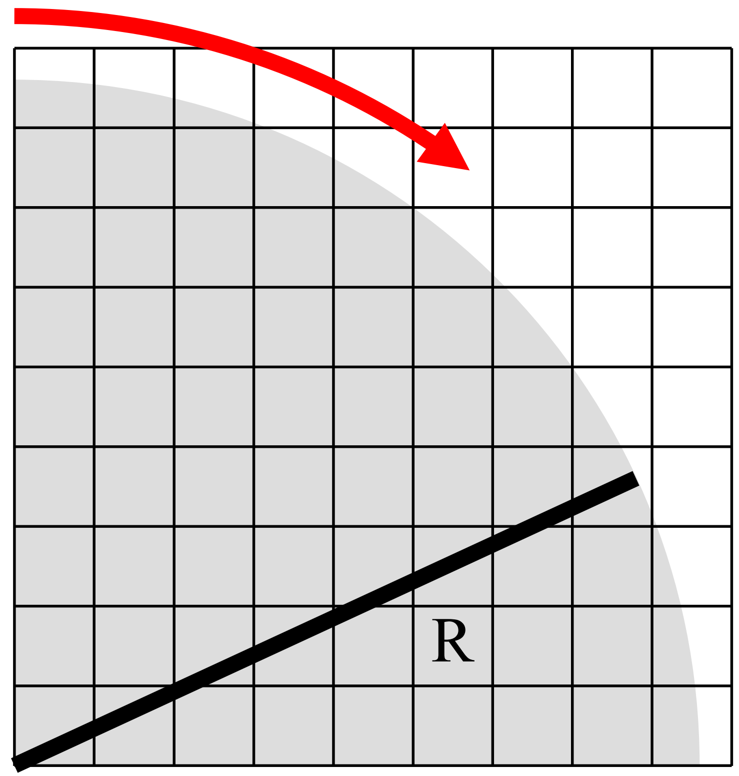
- Decision Function:

  $D(x, y) = \quad x^2 + y^2 - R^2$

- Initialize:

  error term $e = -D(x,y)$

- On each iteration:




  update x: $\quad x' = x + 1$
  update e: $\quad e' = e + 2x + 1$
  if $(e \geq 0.5)$: $\quad y' = y$ (choose pixel E)
  if $(e < 0.5)$: $\quad y' = y - 1$ (choose pixel SE), $\quad e' = e + 1$

# Philosophically

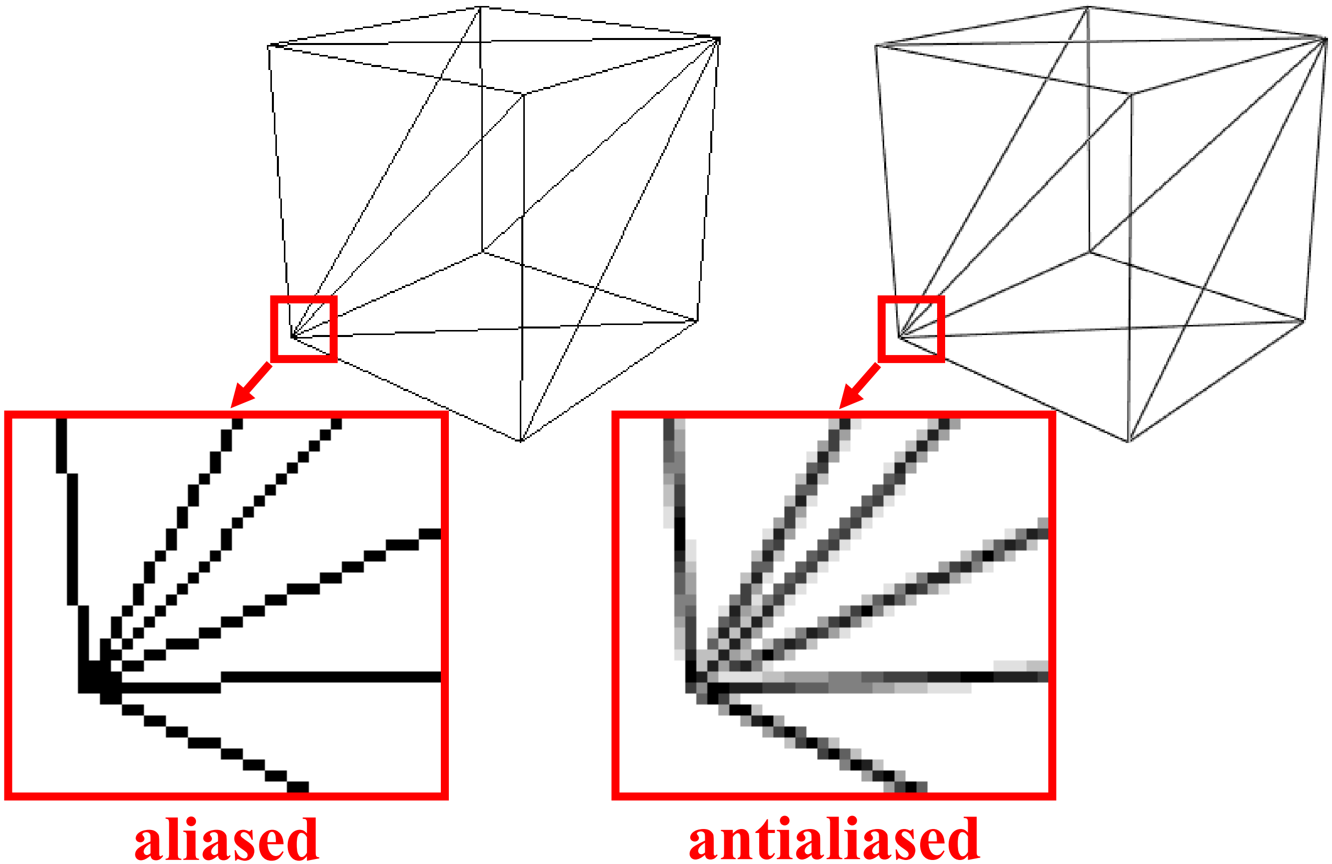
Discrete differential analyzer (DDA):

- Perform incremental computation
- Work on derivative rather than function
- Gain one order for polynomial
  - Line becomes constant derivative
  - Circle becomes linear derivative

# Questions?

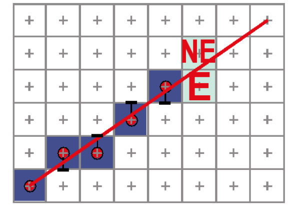
# Antialiased Line Rasterization

- Use gray scales to avoid jaggies
- Will be studied later in the course

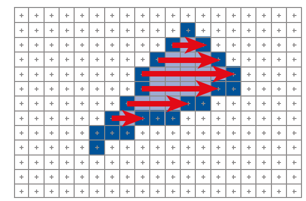


# Today

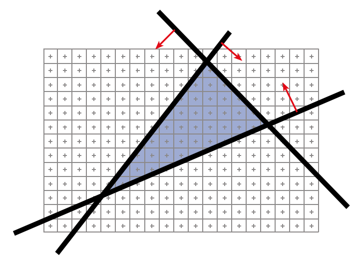
- Line scan-conversion




- Polygon scan conversion

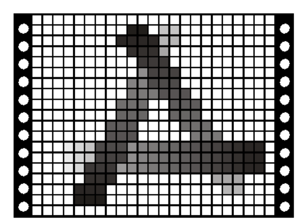


  - smart



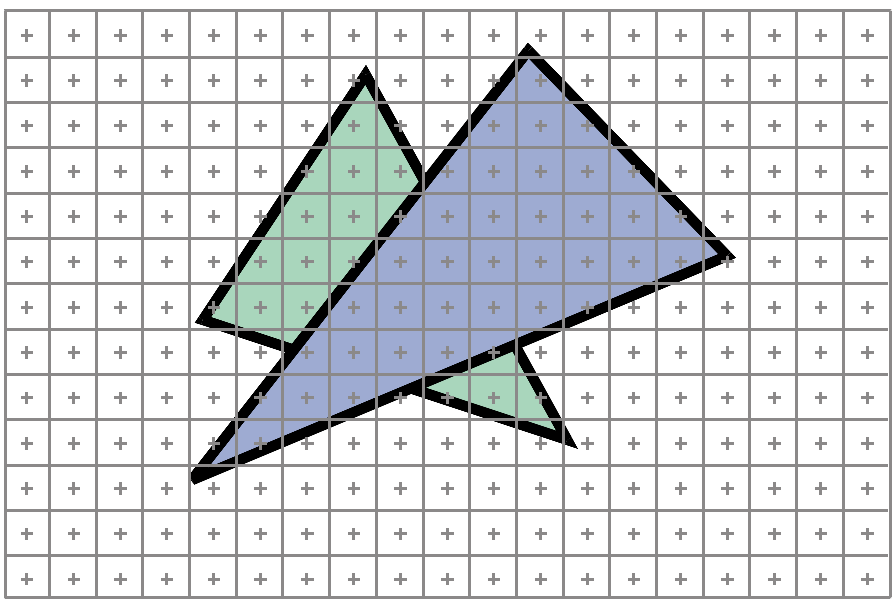
  - back to brute force

- Visibility



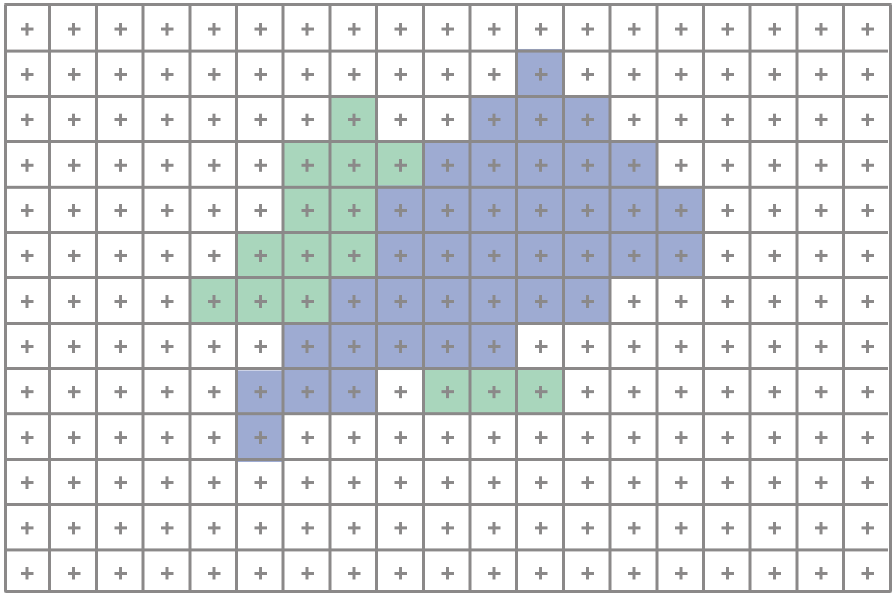
# 2D Scan Conversion

- Geometric primitive
  - 2D: point, line, polygon, circle...
  - 3D: point, line, polyhedron, sphere...
- Primitives are continuous; screen is discrete



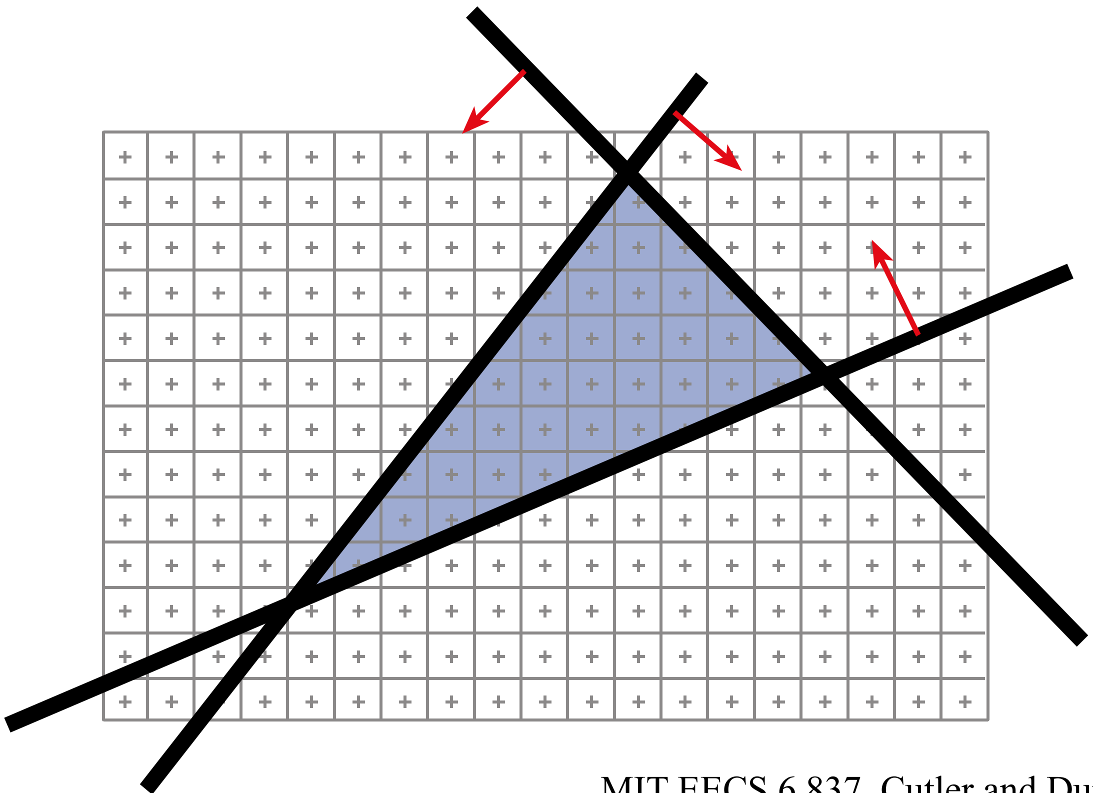
# 2D Scan Conversion

- Solution: compute discrete approximation
- Scan Conversion:
  algorithms for efficient generation of the samples comprising this approximation



# Brute force solution for triangles

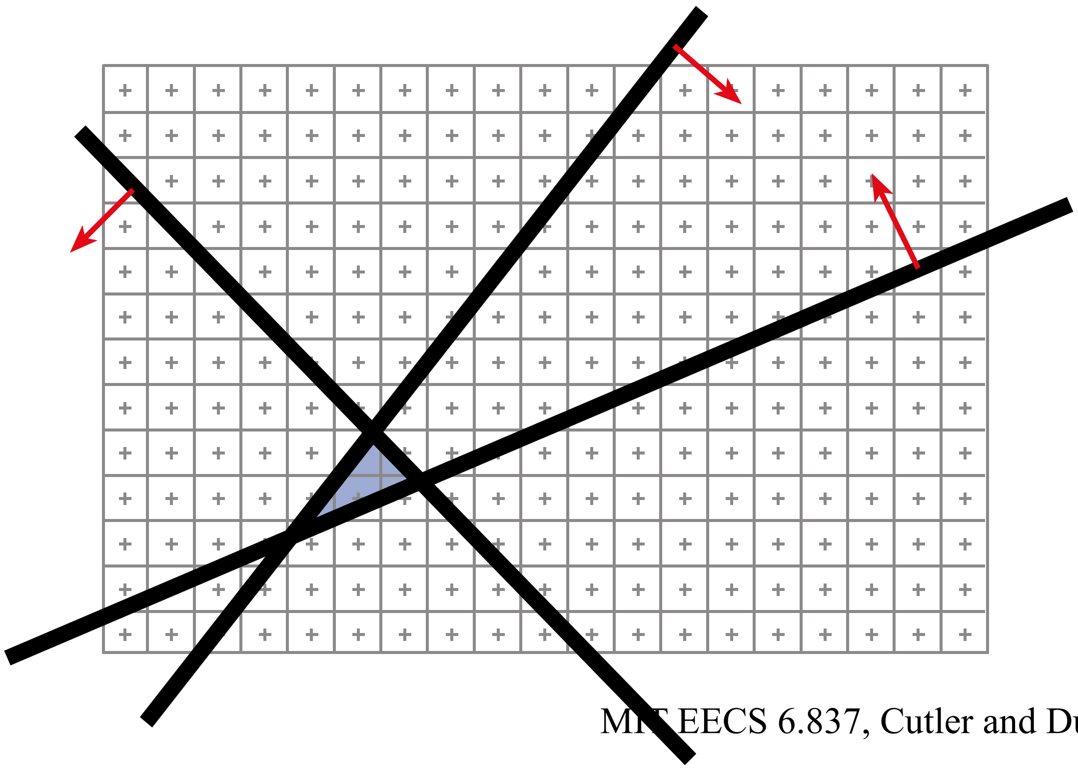
- For each pixel
  - Compute line equations at pixel center
  - "clip" against the triangle



Problem?

# Brute force solution for triangles

- For each pixel
  - Compute line equations at pixel center
  - "clip" against the triangle

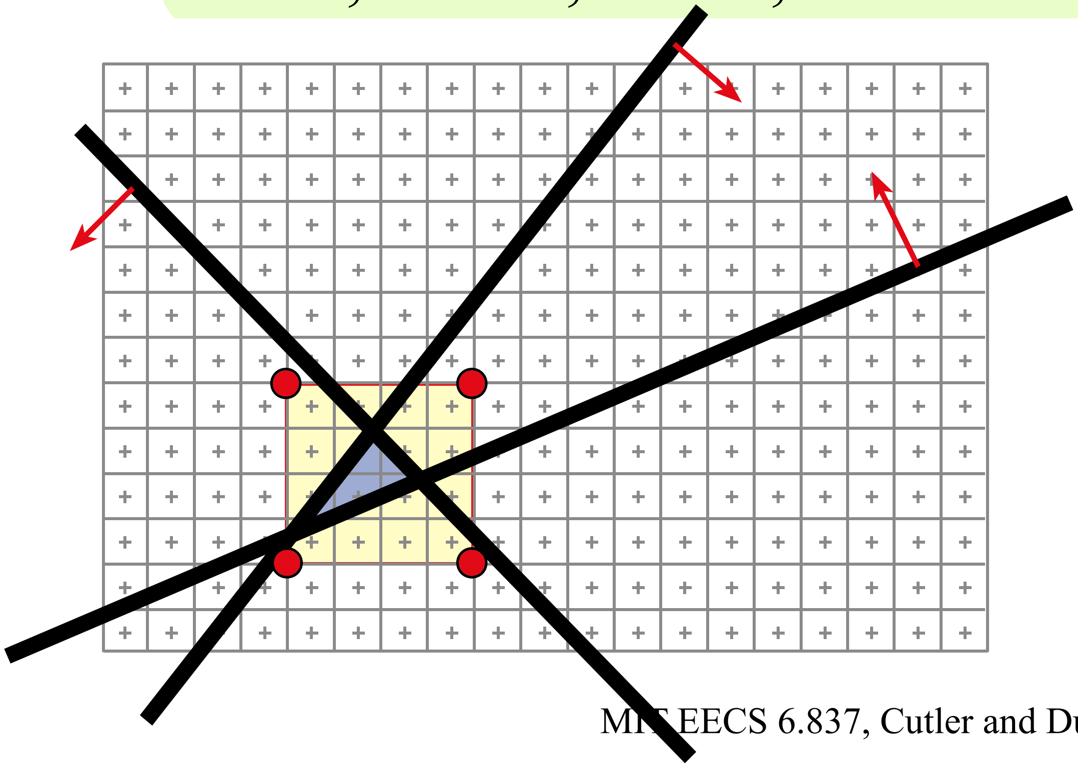


Problem?

If the triangle is small, a lot of useless computation

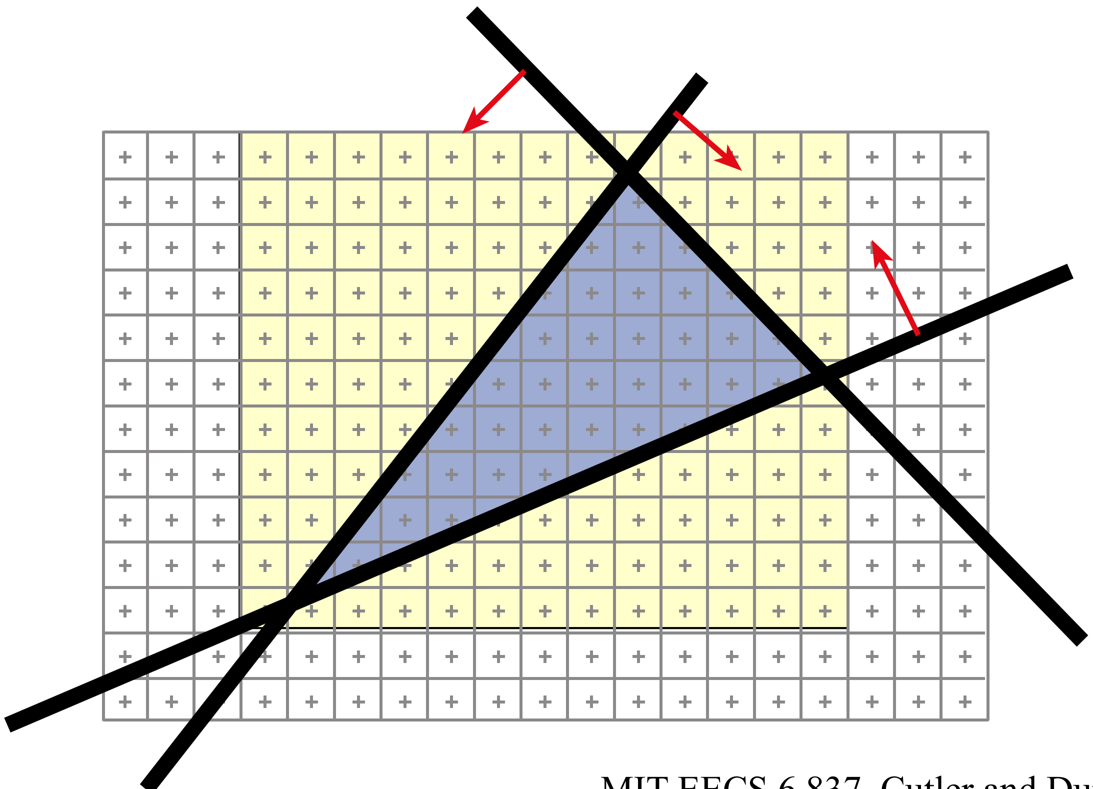
# Brute force solution for triangles

- Improvement: Compute only for the *screen bounding box* of the triangle
- How do we get such a bounding box?
  - Xmin, Xmax, Ymin, Ymax of the triangle vertices



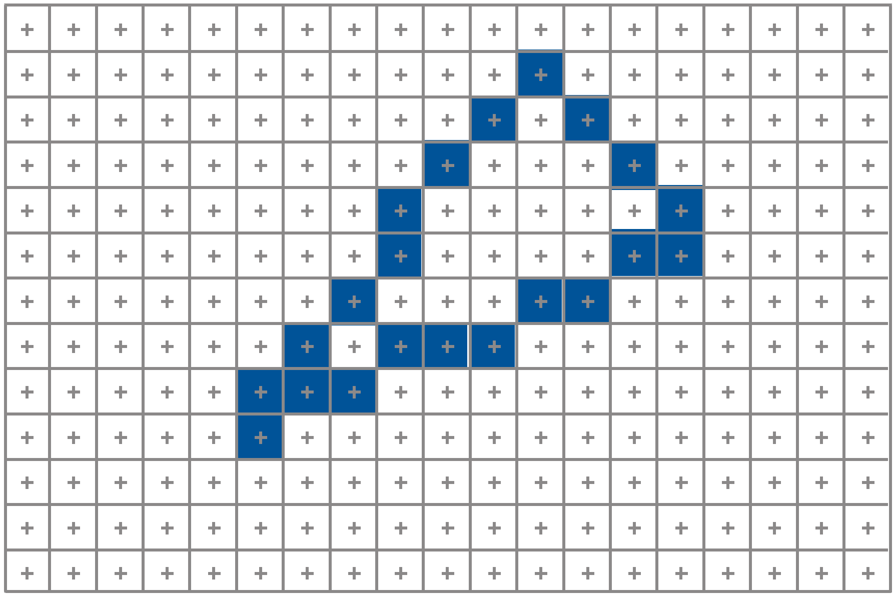
# Can we do better? Kind of!

- We compute the line equation for many useless pixels
- What could we do?



# Use line rasterization

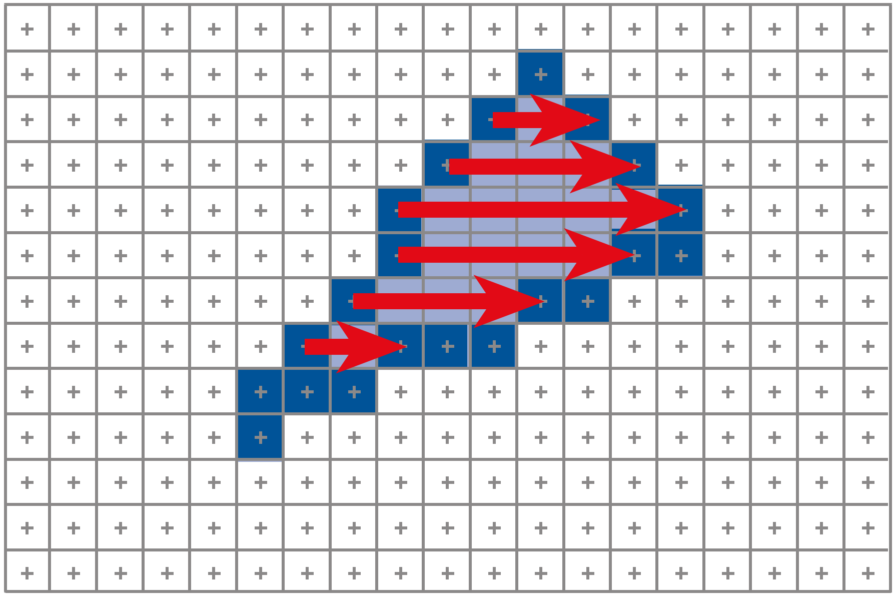
- Compute the boundary pixels



Shirley page 55

# Scan-line Rasterization

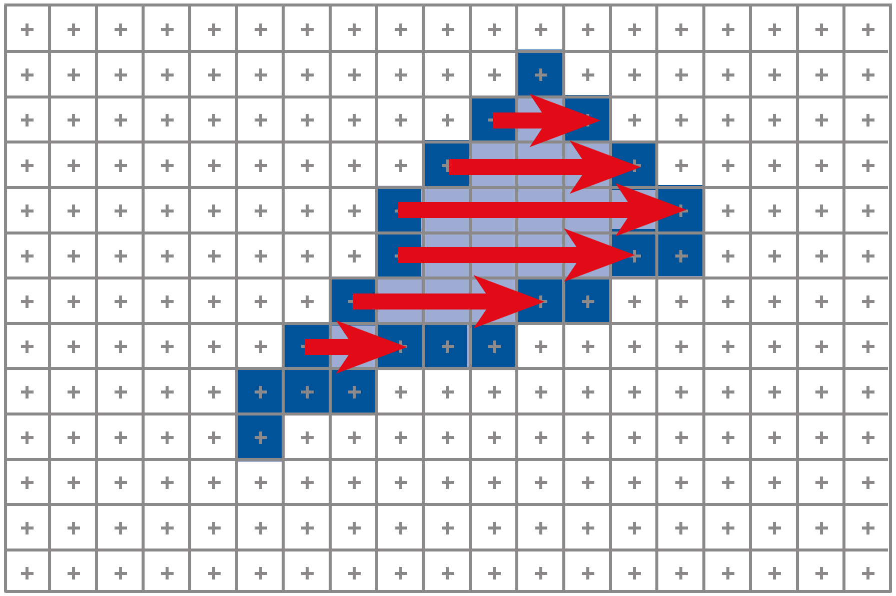
- Compute the boundary pixels
- Fill the spans



Shirley page 55

# Scan-line Rasterization

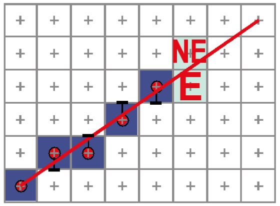
- Requires some initial setup to prepare



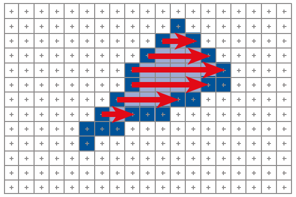
Shirley page 55

# Today

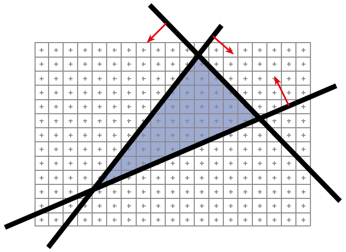
- Line scan-conversion




- Polygon scan conversion

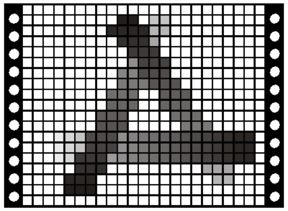


  - smart



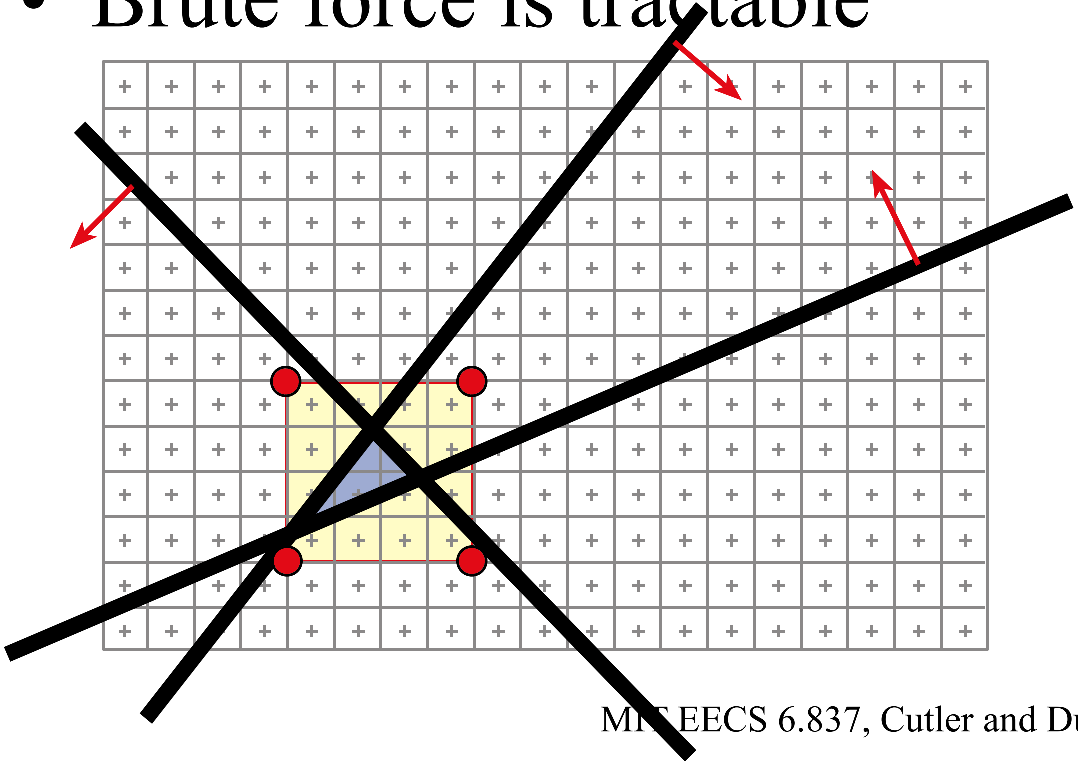
  - back to brute force

- Visibility



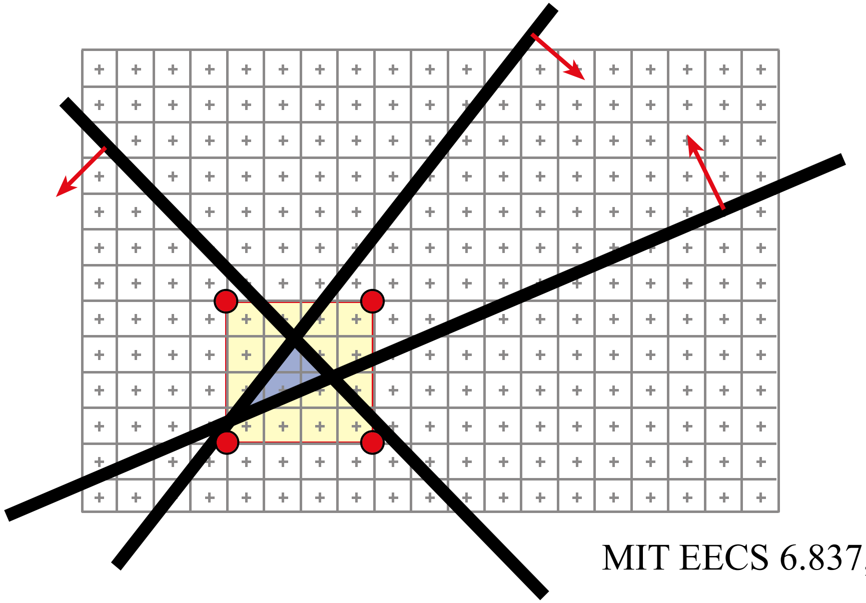
# For modern graphics cards

- Triangles are usually very small
- Setup cost are becoming more troublesome
- Clipping is annoying
- Brute force is tractable



# Modern rasterization

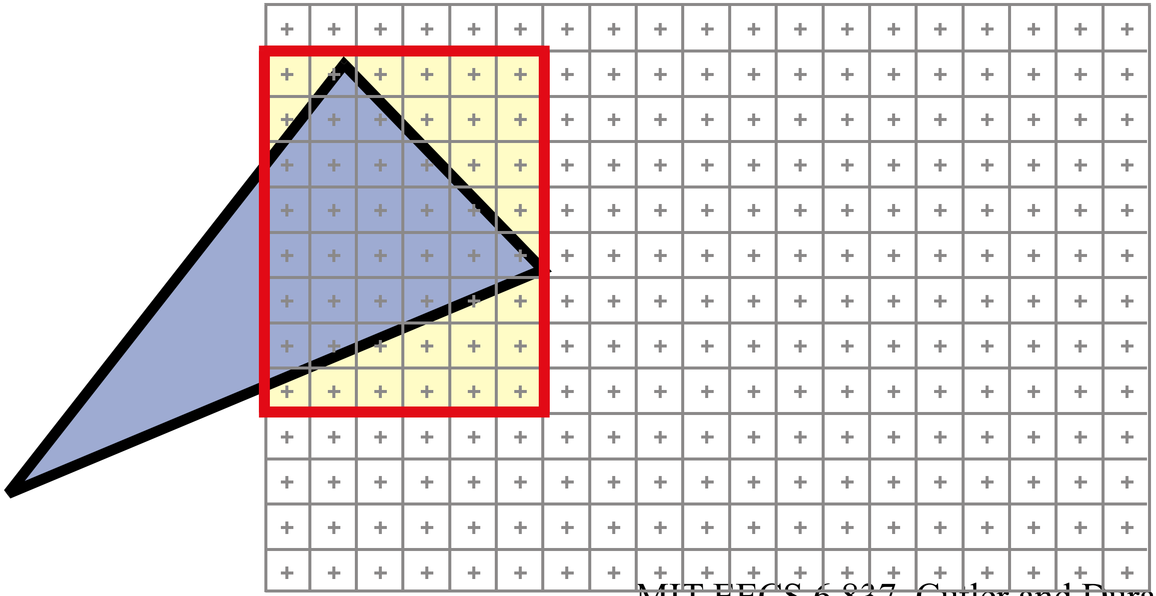
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations
        If all line equations>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



# Modern rasterization

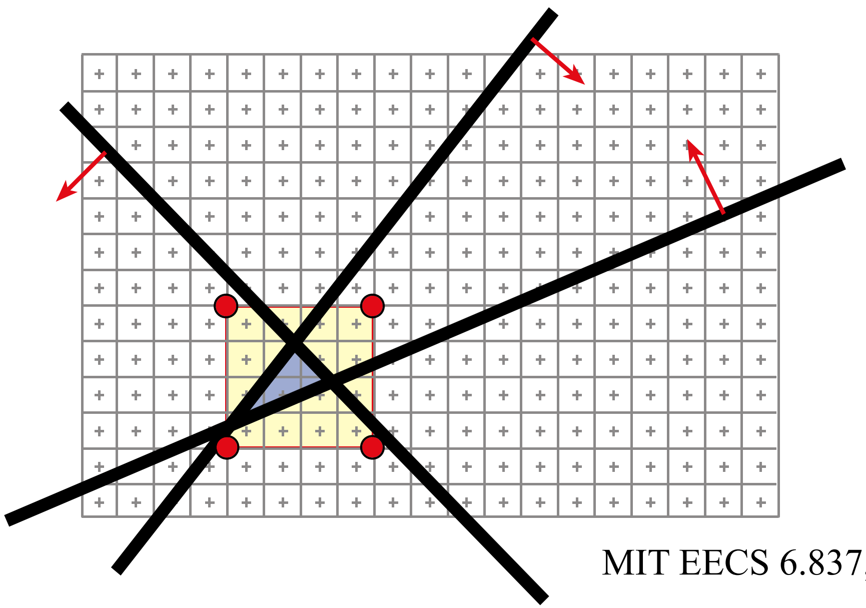
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations
        If all line equations>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

- Note that Bbox clipping is trivial



# Can we do better?

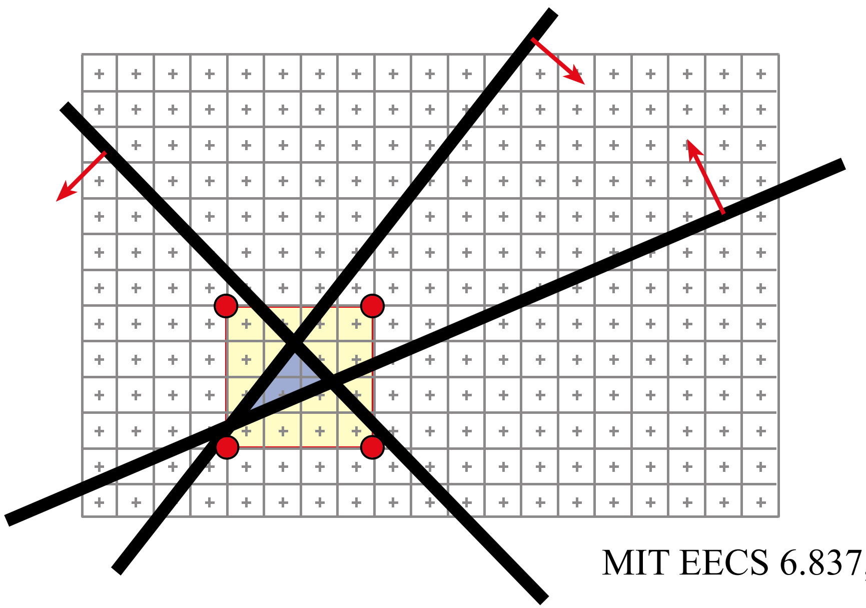
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations
        If all line equations>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```



# Can we do better?

```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Compute line equations ax+by+c
        If all line equations>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

- We don't need to recompute line equation from scratch



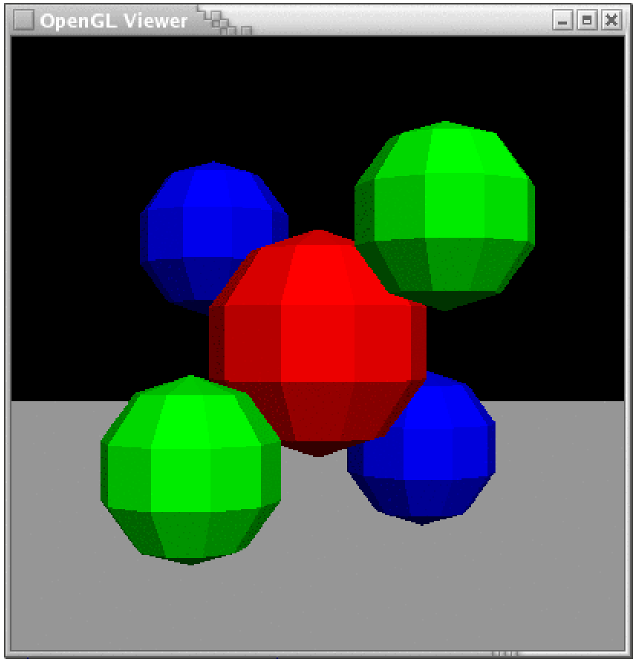
# Can we do better?

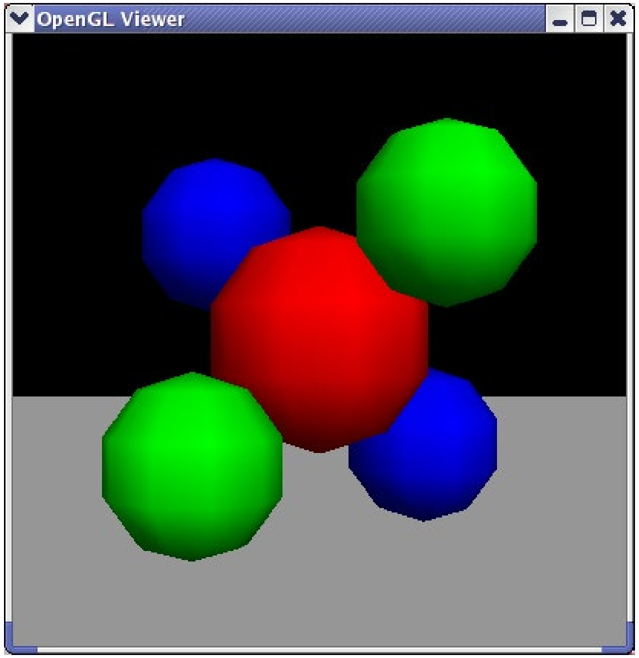
```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   Setup line eq
      compute a_idx, b_idy for the 3 lines
      Initialize line eq, values for bbox corner
         L_i=a_ix0+b_iy+c_i
   For all scanline y in bbox
      For 3 lines, update Li
      For all x in bbox
         Increment line equations: Li+=adx
      If all Li>0  //pixel [x,y] in triangle
            Framebuffer[x,y]=triangleColor
```

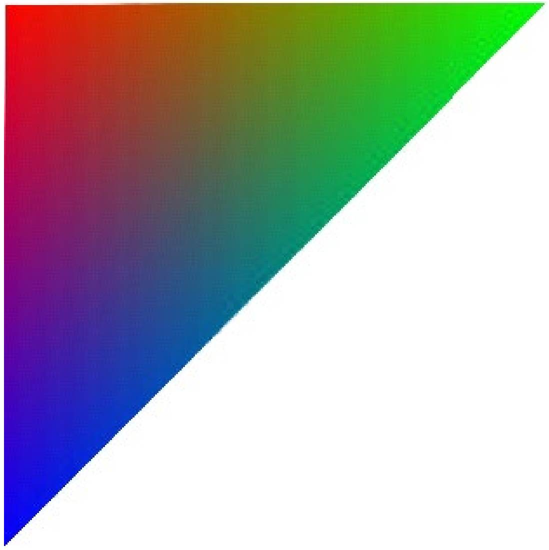
- We save one multiplication per pixel

# Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation

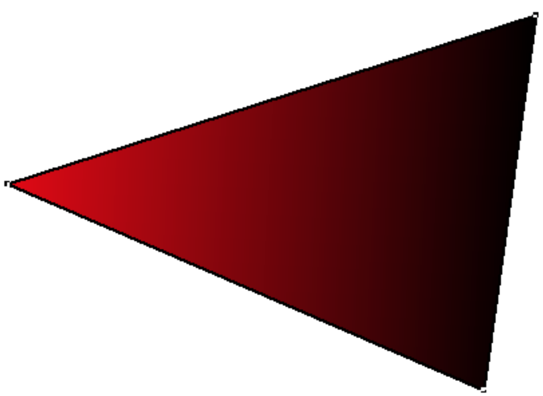


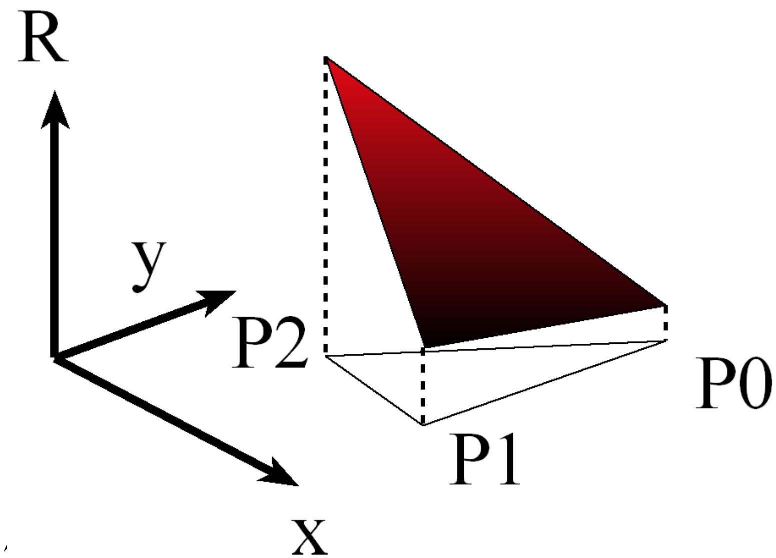



# Adding Gouraud shading

- Interpolate colors of the 3 vertices
- Linear interpolation, e.g. for R channel:
  - $R=a_R x+b_R y+c_R$
  - Such that R[x0,y0]=R0; R[x1, y1]=R1; R[x2,y2]=R2
  - Same as a plane equation in (x,y,R)



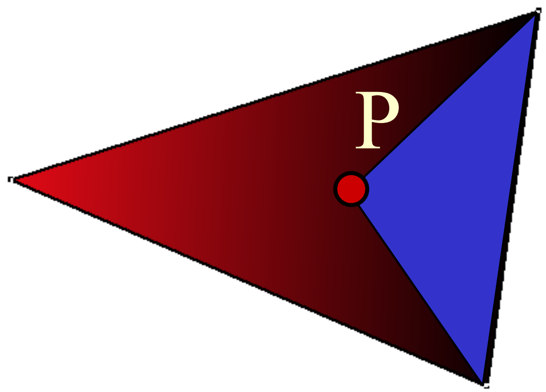


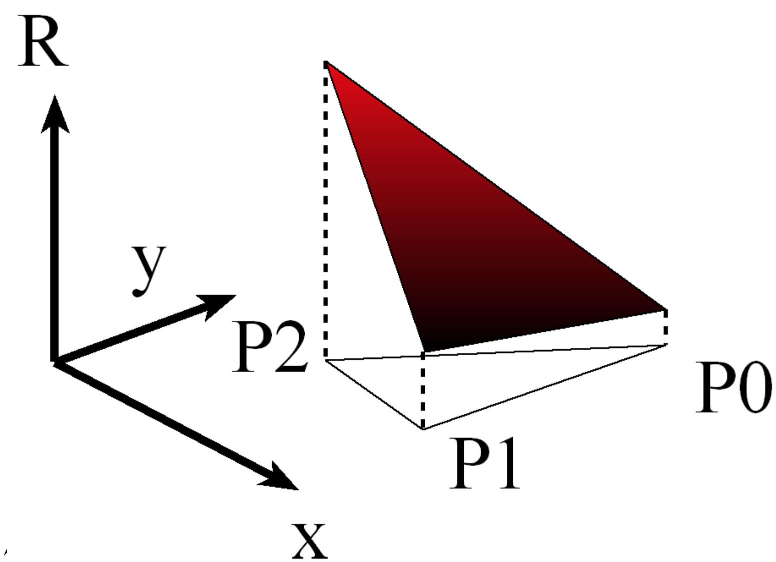
# Adding Gouraud shading

```
Interpolate colors
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   Setup line eq
   Setup color equation
   For all pixels in bbox
      Increment line equations
      Increment color equation
      If all Li>0 //pixel [x,y] in triangle
            Framebuffer[x,y]=interpolatedColor
```

# Adding Gouraud shading

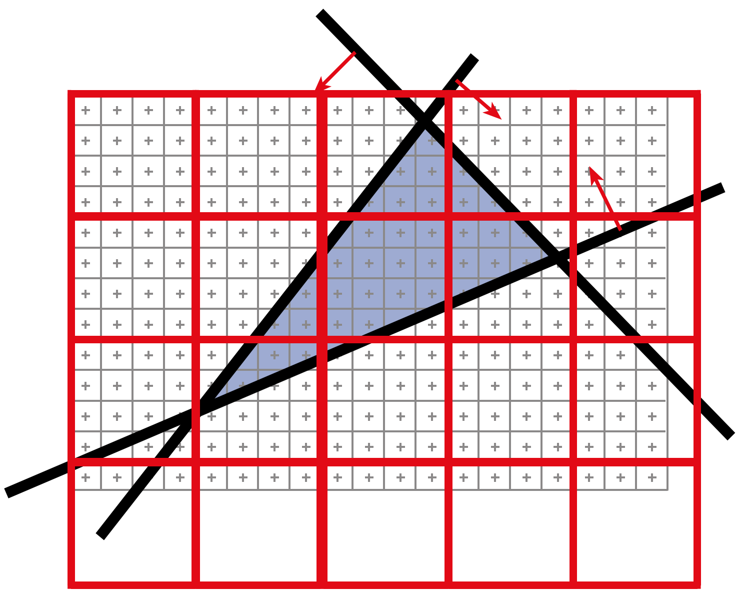
- Interpolate colors of the 3 vertices
- Other solution: use barycentric coordinates
- $R = \alpha R_0 + \beta R_1 + \gamma R_2$
- Such that $P = \alpha P_0 + \beta P_1 + \gamma P_2$
-



# In the modern hardware

- Edge eq. in homogeneous coordinates [x, y, w]
- Tiles to add a mid-level granularity
  - Early rejection of tiles
  - Memory access coherence



# Ref


- Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, Jr., John Eyles and John Poulton, “Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes”, Proceedings of SIGGRAPH ‘85 (San Francisco, CA, July 22–26, 1985). In *Computer Graphics*, v19n3 (July 1985), ACM SIGGRAPH, New York, NY, 1985.
- Juan Pineda, “A Parallel Algorithm for Polygon Rasterization”, Proceedings of SIGGRAPH ‘88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, v22n4 (August 1988), ACM SIGGRAPH, New York, NY, 1988. Figure 7: Image from the spinning teapot performance test.
- Triangle Scan Conversion using 2D Homogeneous Coordinates, Marc Olano Trey Greer [http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf](http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf)

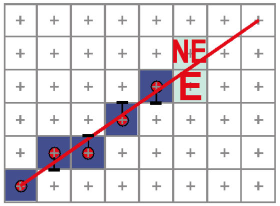
# Take-home message

- The appropriate algorithm depends on
  - Balance between various resources (CPU, memory, bandwidth)
  - The input (size of triangles, etc.)
- Smart algorithms often have initial preprocess
  - Assess whether it is worth it
- To save time, identify redundant computation
  - Put outside the loop and interpolate if needed

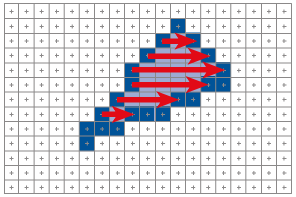
# Questions?

# Today

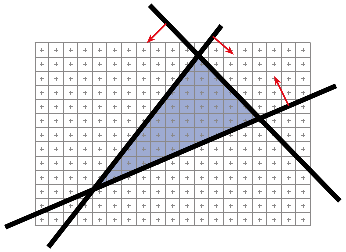
- Line scan-conversion




- Polygon scan conversion

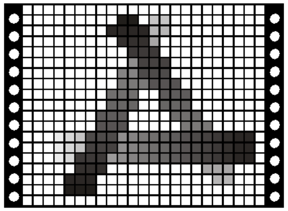


  – smart



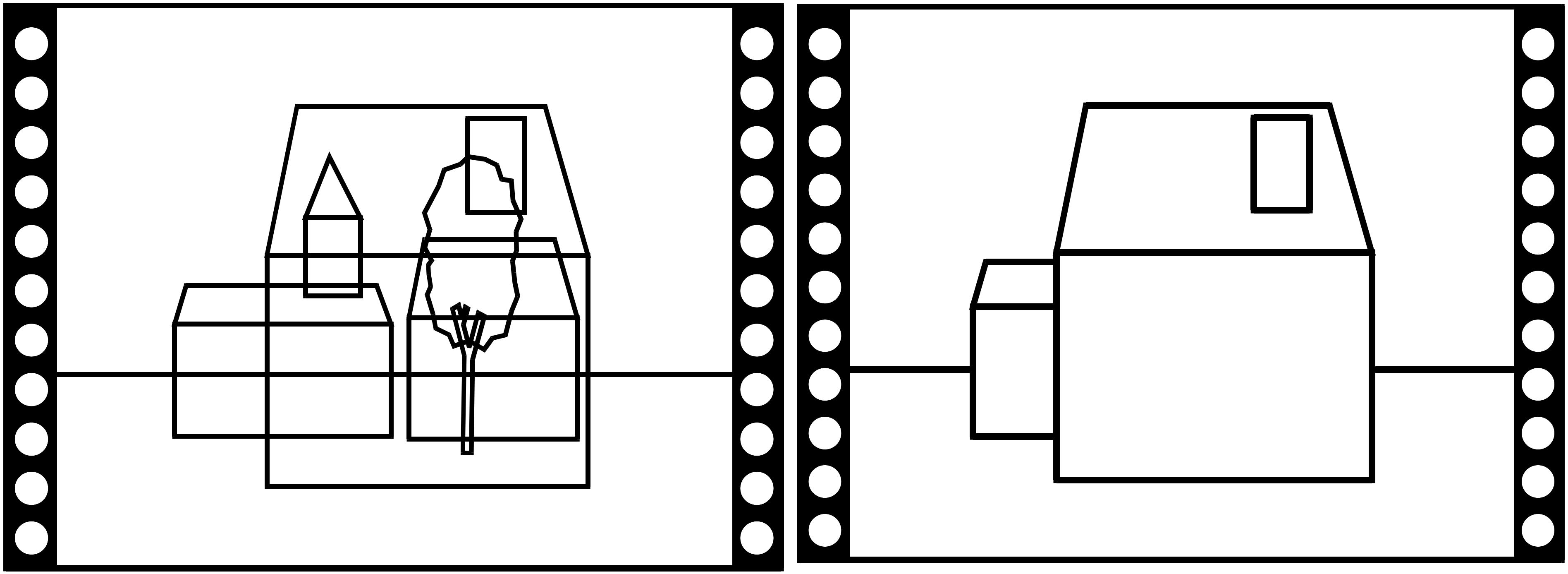
  – back to brute force

- Visibility



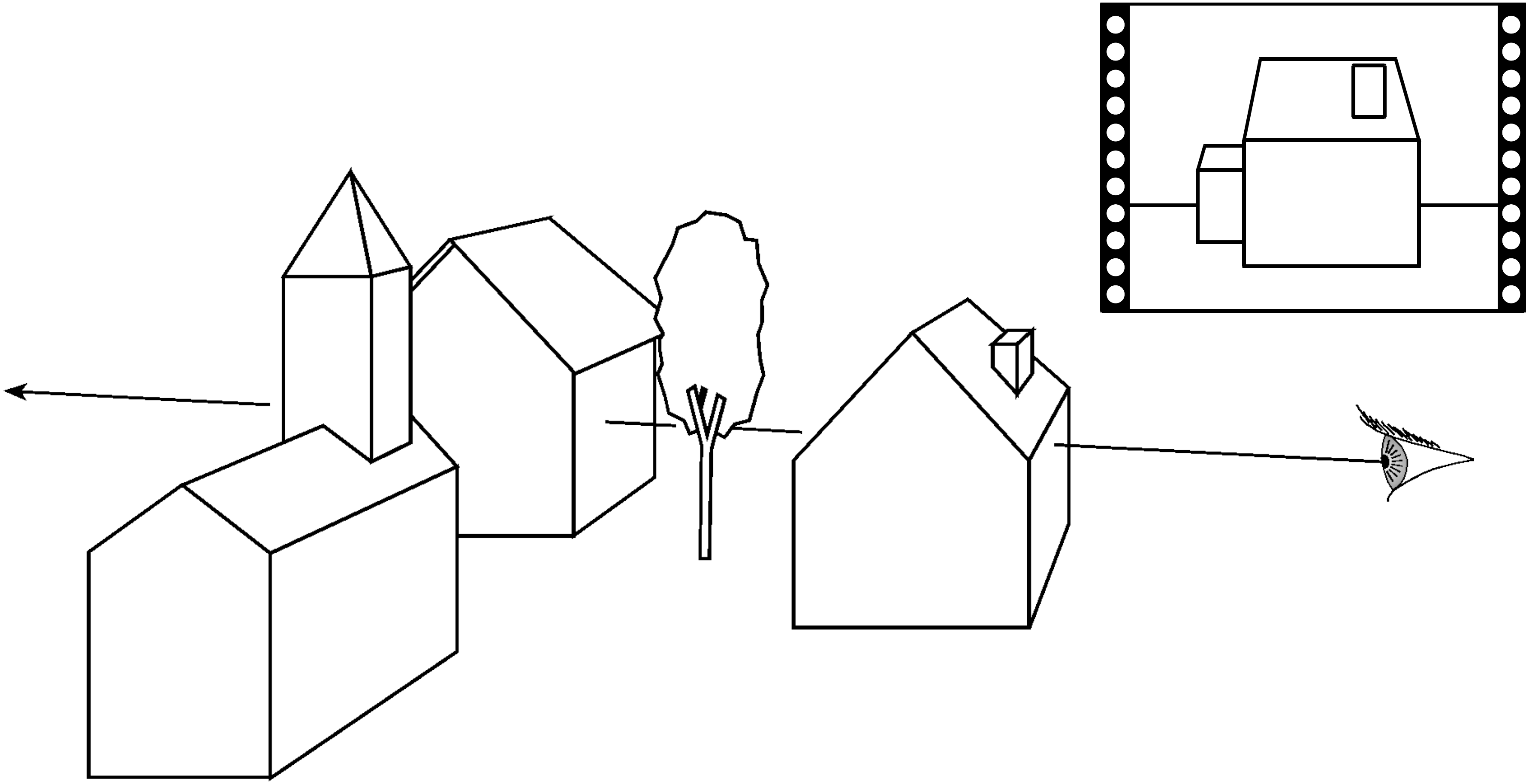
# Visibility

- How do we know which parts are visible/in front?



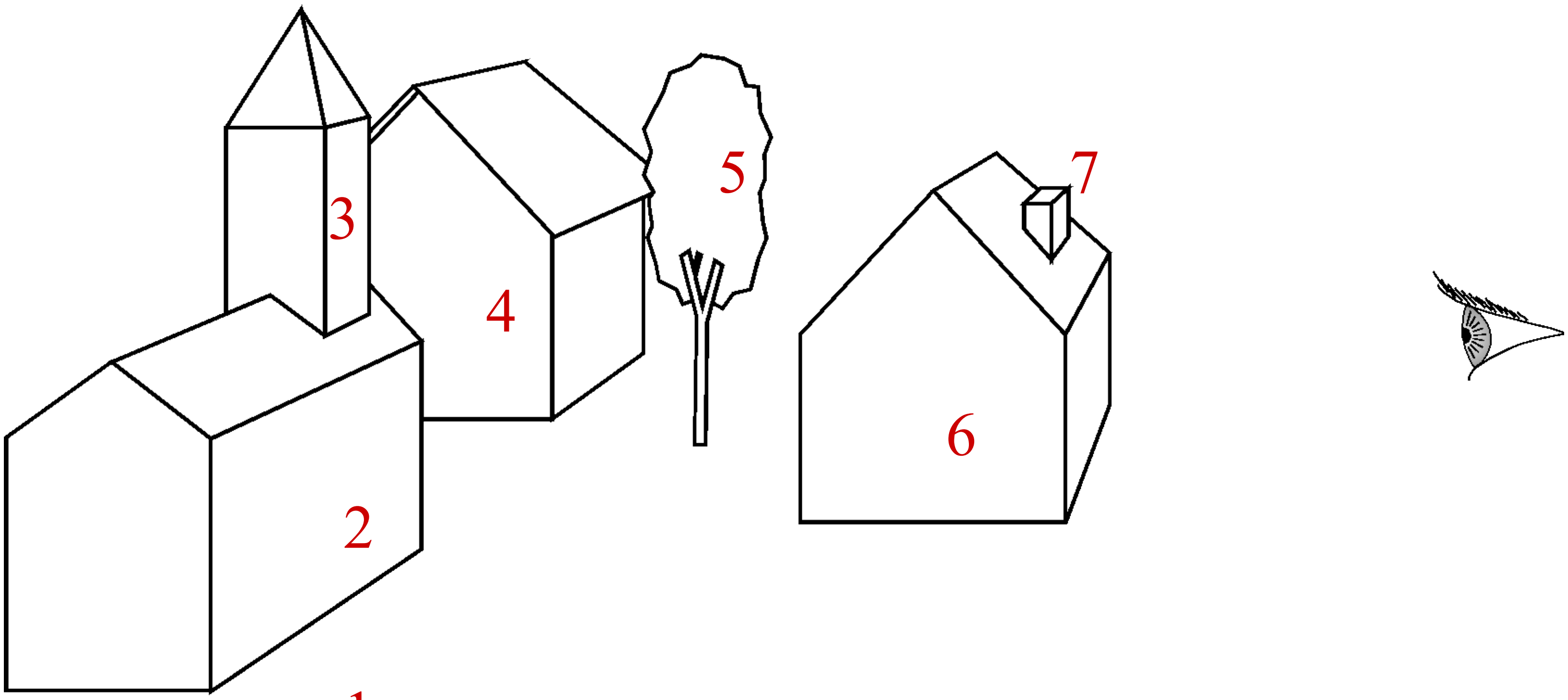
# Ray Casting

- Maintain intersection with closest object



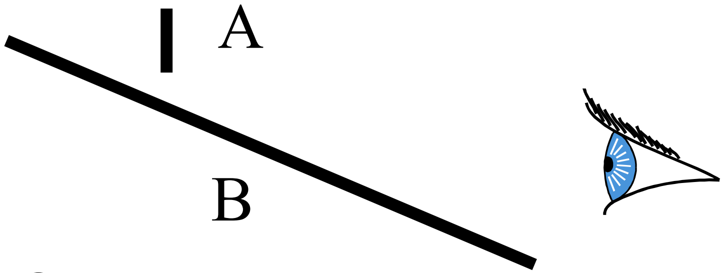
# Painter's algorithm

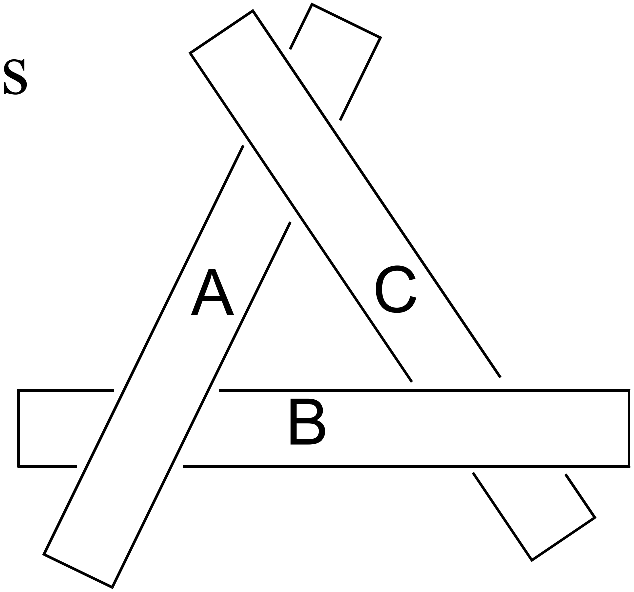
- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?



# Painter's algorithm

- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?
  – No, there can be cycles
  – Requires to split polygons



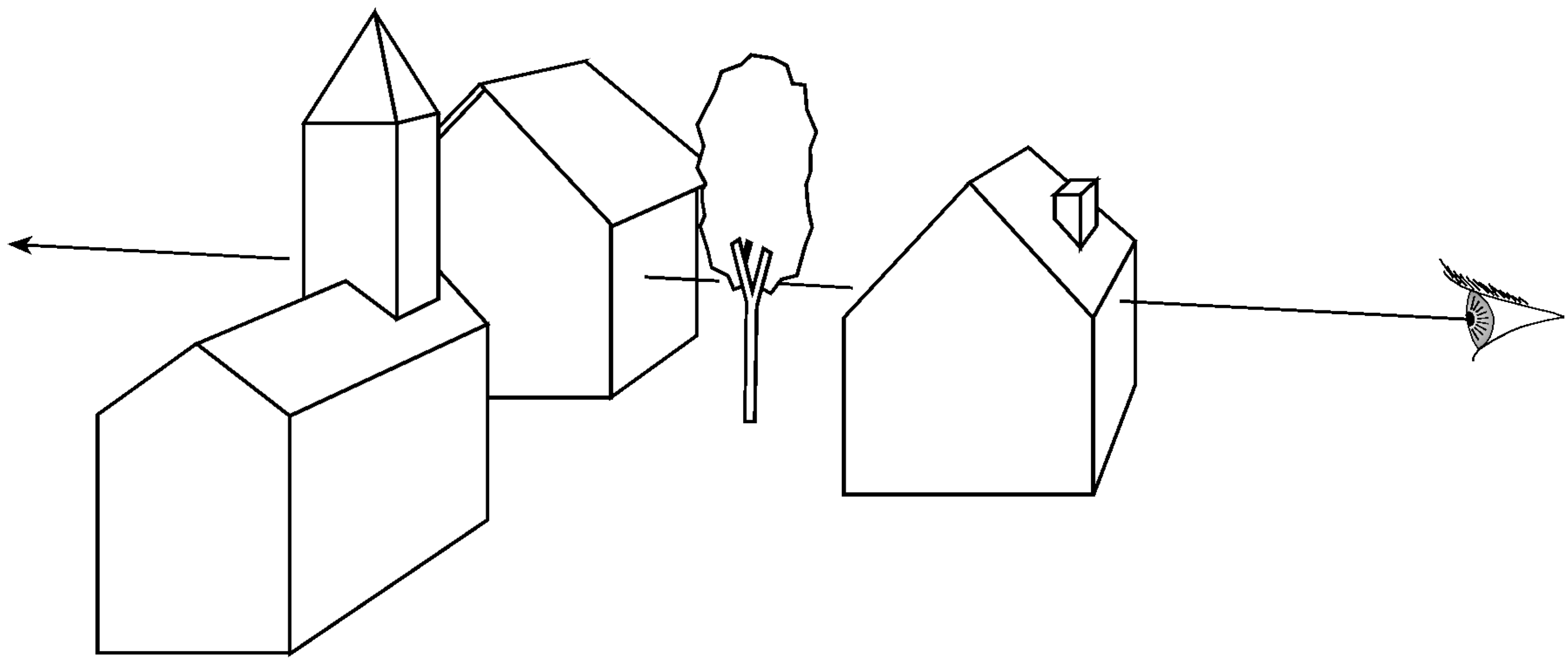



# Painter's algorithm

- Old solution for hidden-surface removal
  - Good because ordering is useful for other operations (transparency, antialiasing)
- But
  - Ordering is tough
  - Cycles
  - Must be done by CPU
- Hardly used now
- But some sort of partial ordering is sometimes useful
  - Usuall front-to-back
  - To make sure foreground is rendered first
  - For transparency

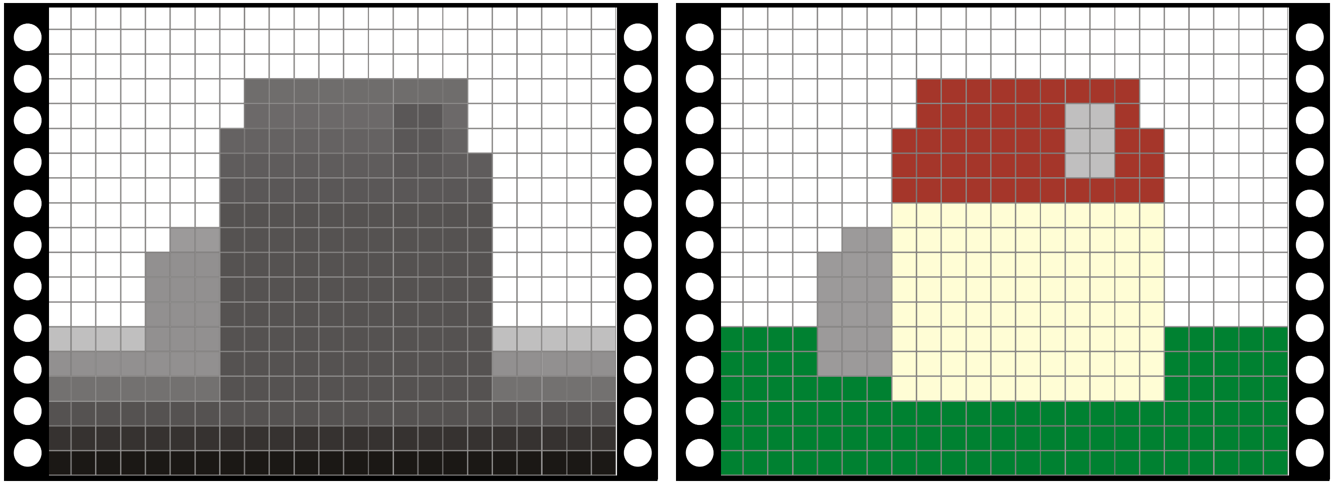
# Visibility

- In ray casting, use intersection with closest t
- Now we have swapped the loops (pixel, object)
- How do we do?



# Z buffer

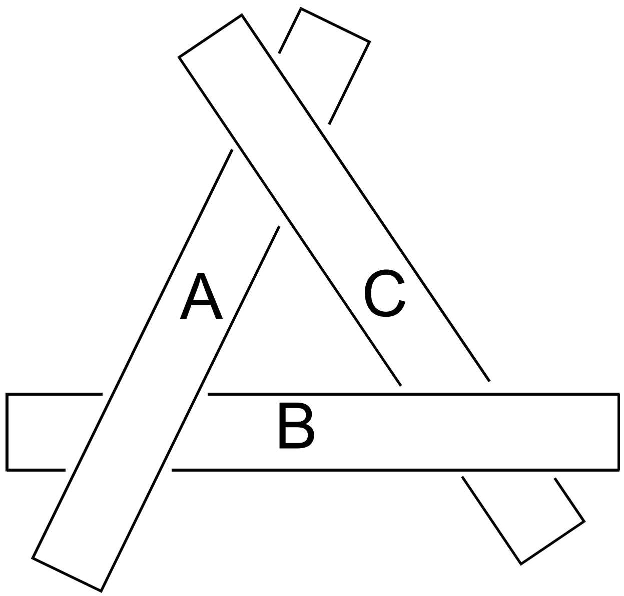
- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if new z is closer than z-buffer value

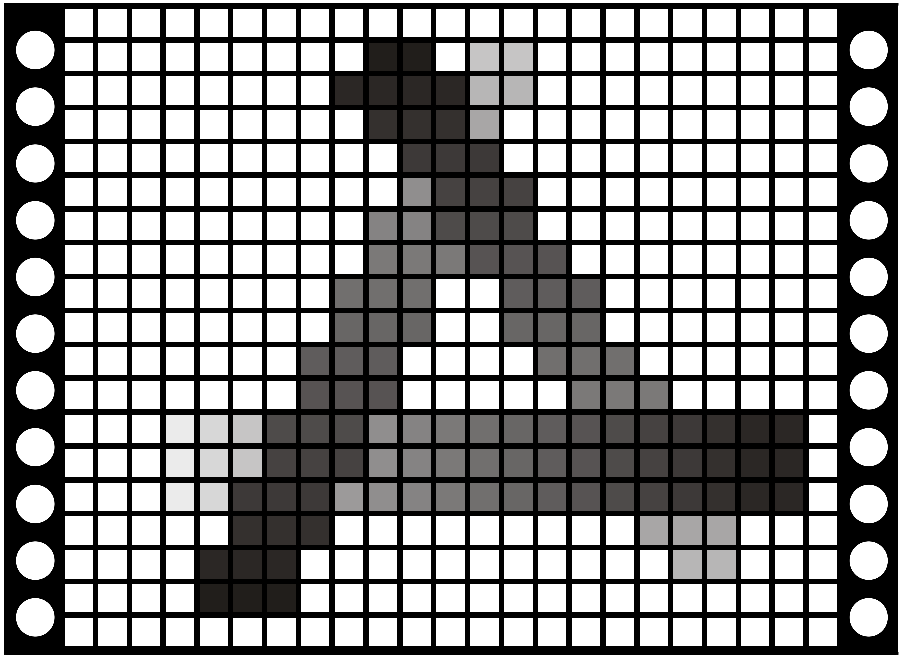


# Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Compute curentZ
      Increment currentColor
      If all line equations>0  //pixel [x,y] in triangle
         If currentZ<zBuffer[x,y]  //pixel is visible
            Framebuffer[x,y]=currentColor
            zBuffer[x,y]=currentZ
```

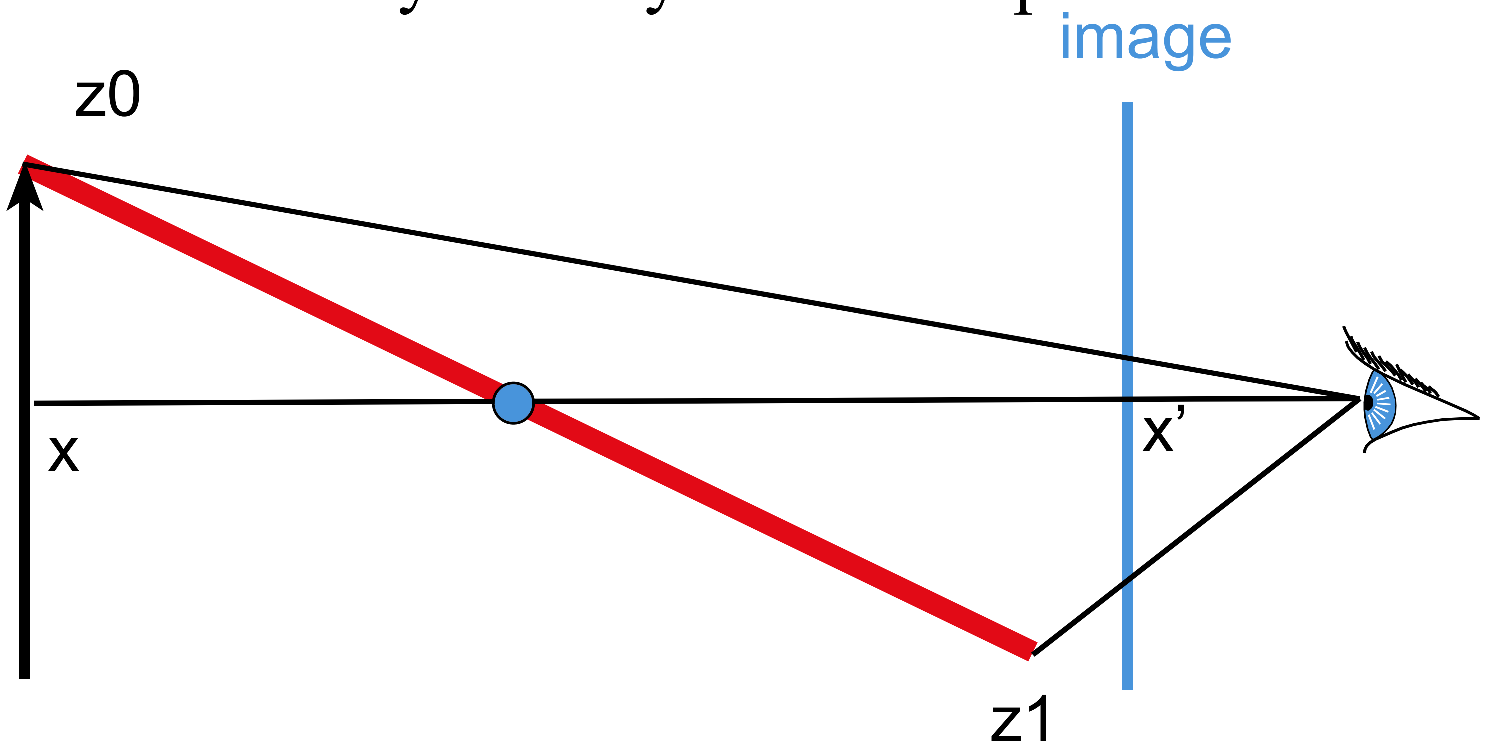
# Works for hard cases!



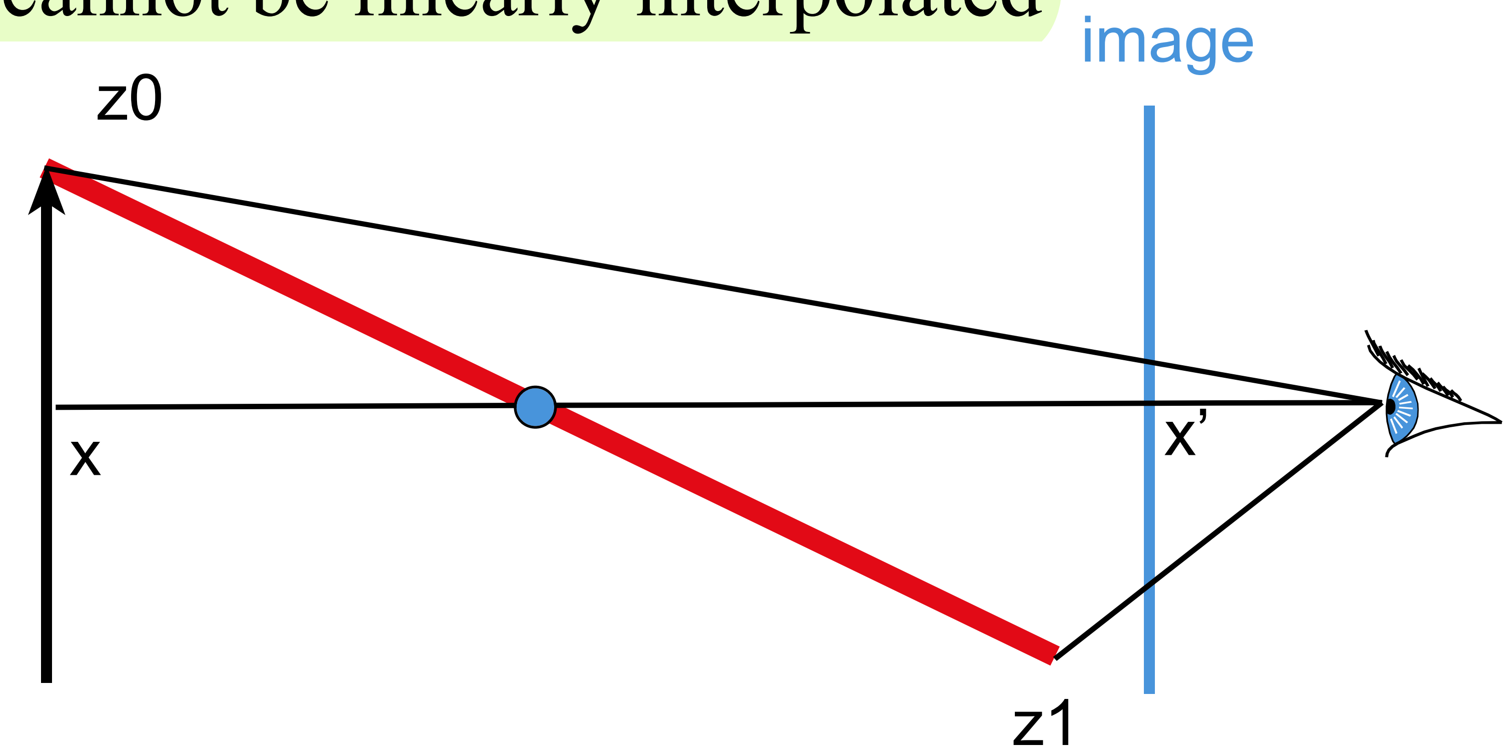
# What exactly do we store

- Floating point distance
- Can we interpolate *z* in screen space?
  - i.e. does *z* vary linearly in screen space?



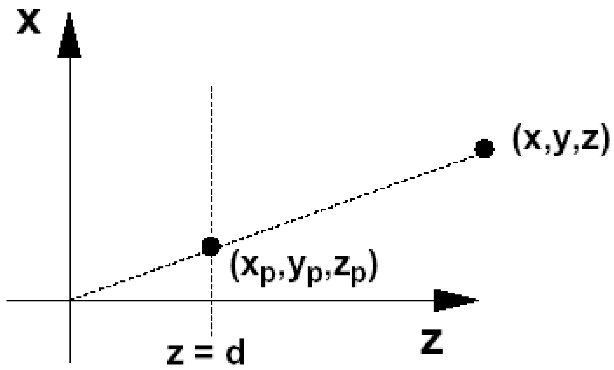
# Z interpolation

- X'=x/z
- Hyperbolic variation
- Z cannot be linearly interpolated



# Simple Perspective Projection

- Project all points along the *z* axis to the $z = d$ plane, eyepoint at the origin

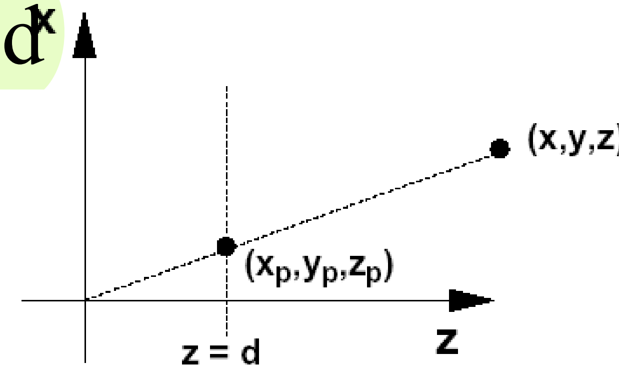



*homogenize*

$$\begin{pmatrix} x * d / z \\ y * d / z \\ d \\ 1 \end{pmatrix} = \begin{bmatrix} x \\ y \\ z \\ z / d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

# Yet another Perspective Projection

- Change the z component
- Compute d/z
- Can be linearly interpolated




*homogenize*

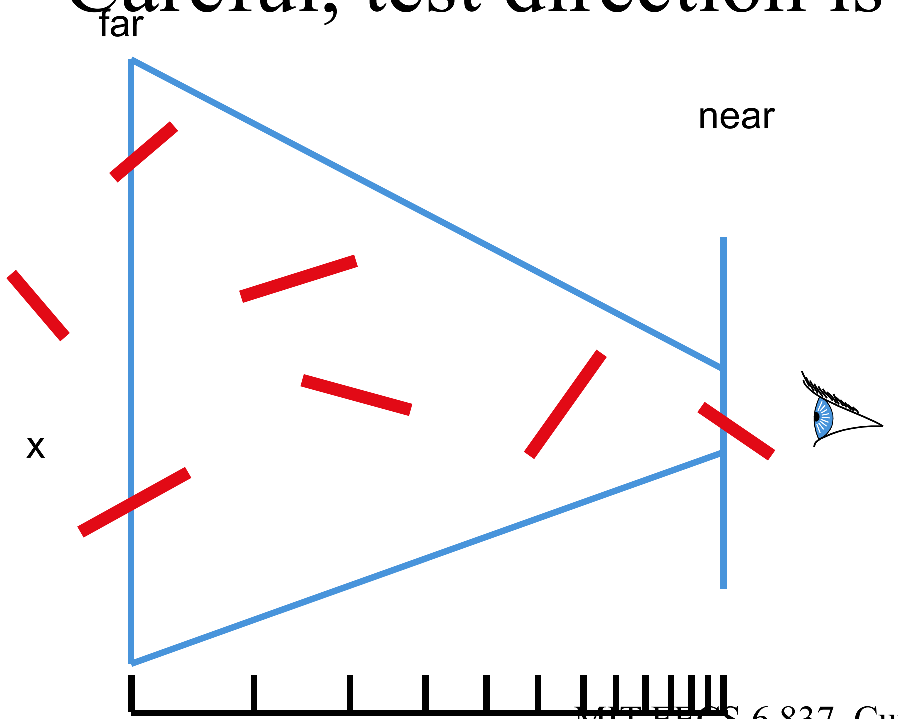
$$\begin{pmatrix} x * d / z \\ y * d / z \\ d/z \\ 1 \end{pmatrix} \xleftarrow{\text{homogenize}} \begin{pmatrix} x \\ y \\ 1 \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

# Advantages of 1/z

- Can be interpolated linearly in screen space
- Puts more precision for close objects
- Useful when using integers
  - more precision where perceptible

# Integer z-buffer

- Use 1/z to have more precision in the foreground
- Set a near and far plane
  - 1/z values linearly encoded between 1/near and 1/far
- Careful, test direction is reversed

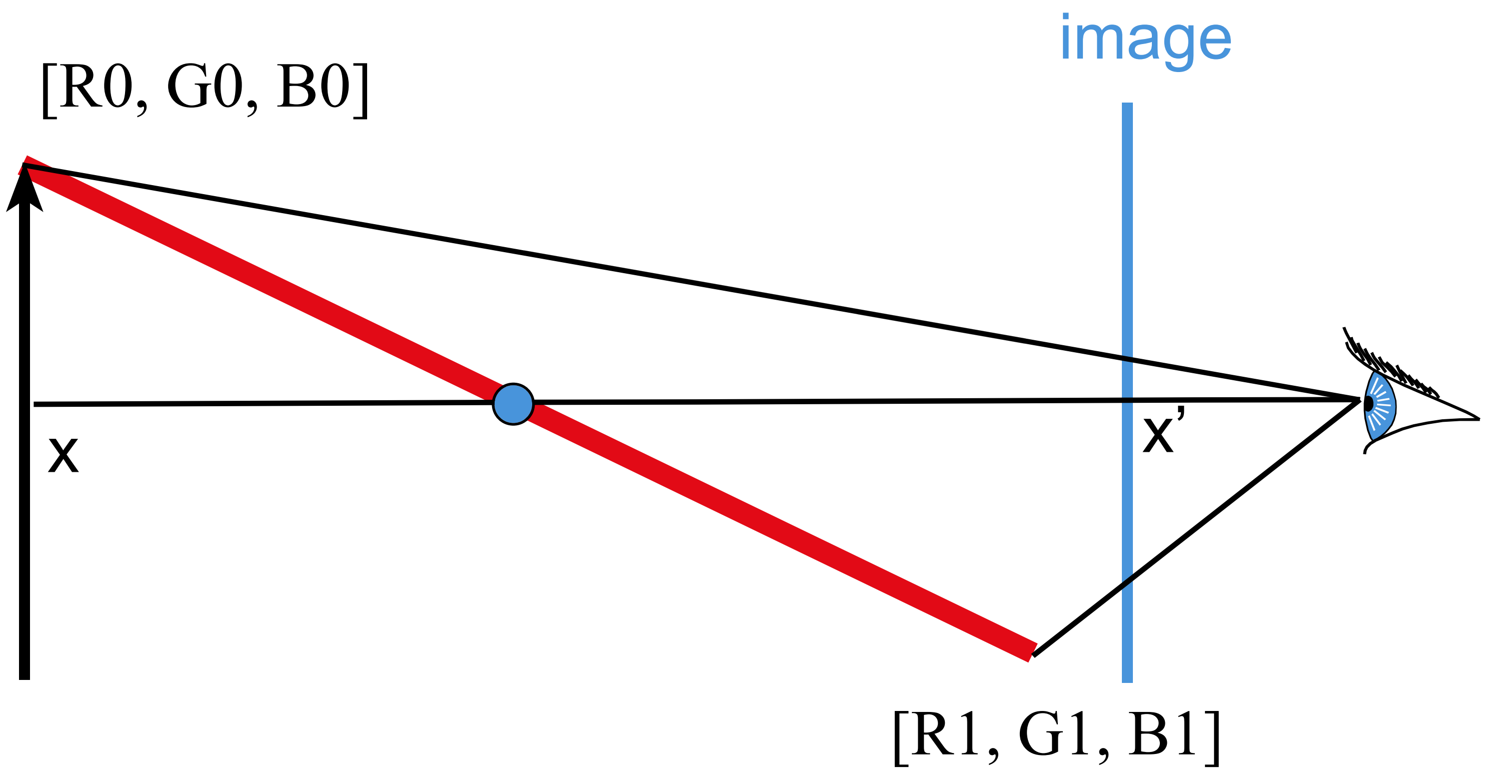


# Integer Z-buffer pseudo code

```
For every triangle
    Compute Projection, color at vertices
    Setup line equations, depth equation
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Increment line equations
        Increment curent_1ovZ
        Increment currentColor
        If all line equations>0  //pixel [x,y] in triangle
            If current_1ovZ>1ovzBuffer[x,y]//pixel is visible
                Framebuffer[x,y]=currentColor
                1ovzBuffer[x,y]=current1ovZ
```

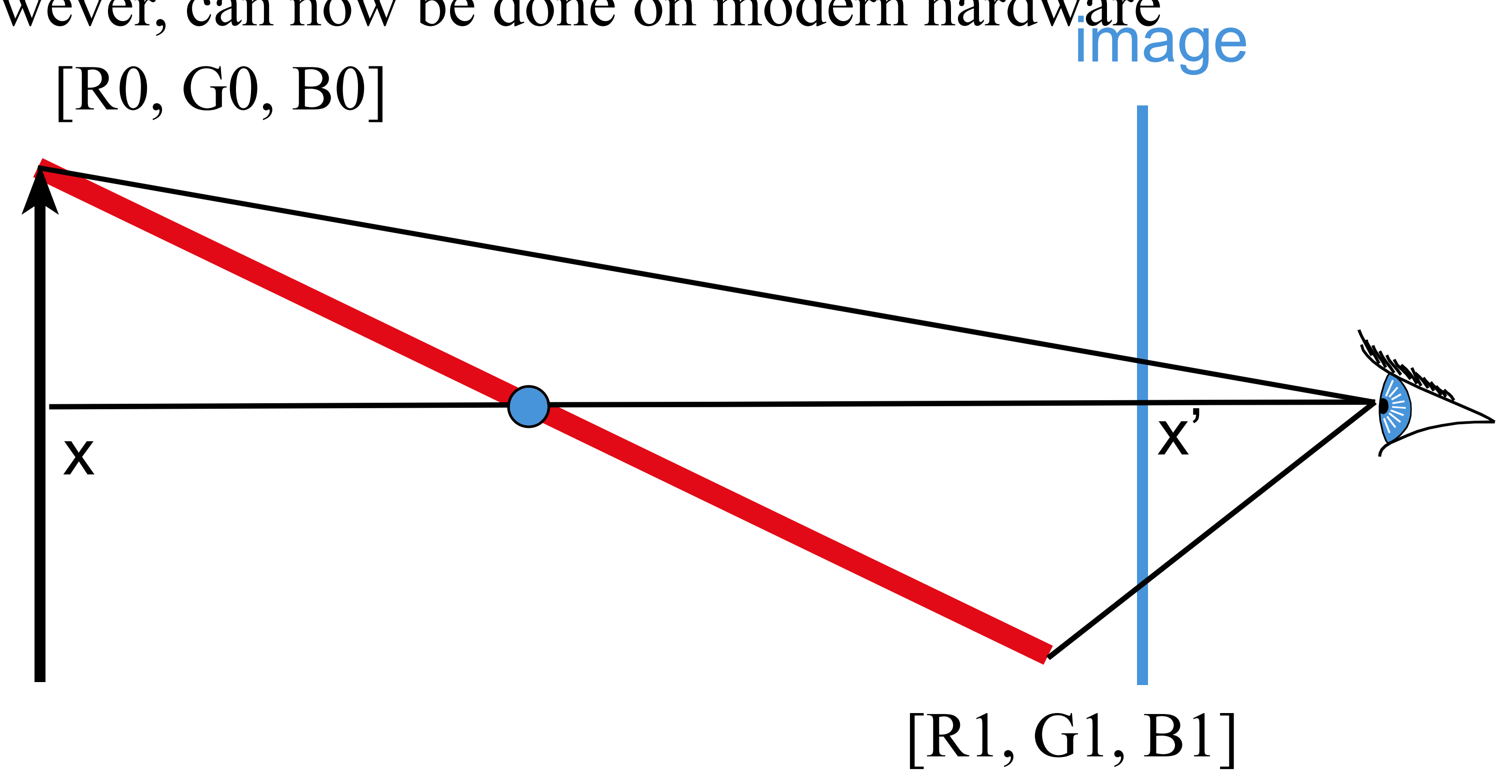
# Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Is it correct?

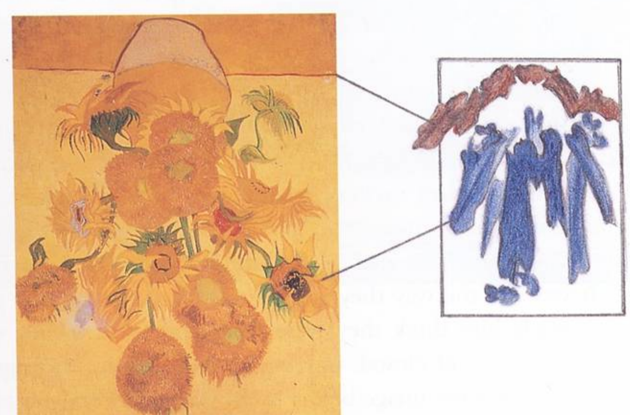


# Gouraud interpolation

- Gouraud: interpolate color linearly in screen space
- Not correct. We should use hyperbolic interpolation
- But quite costly (division)
- However, can now be done on modern hardware



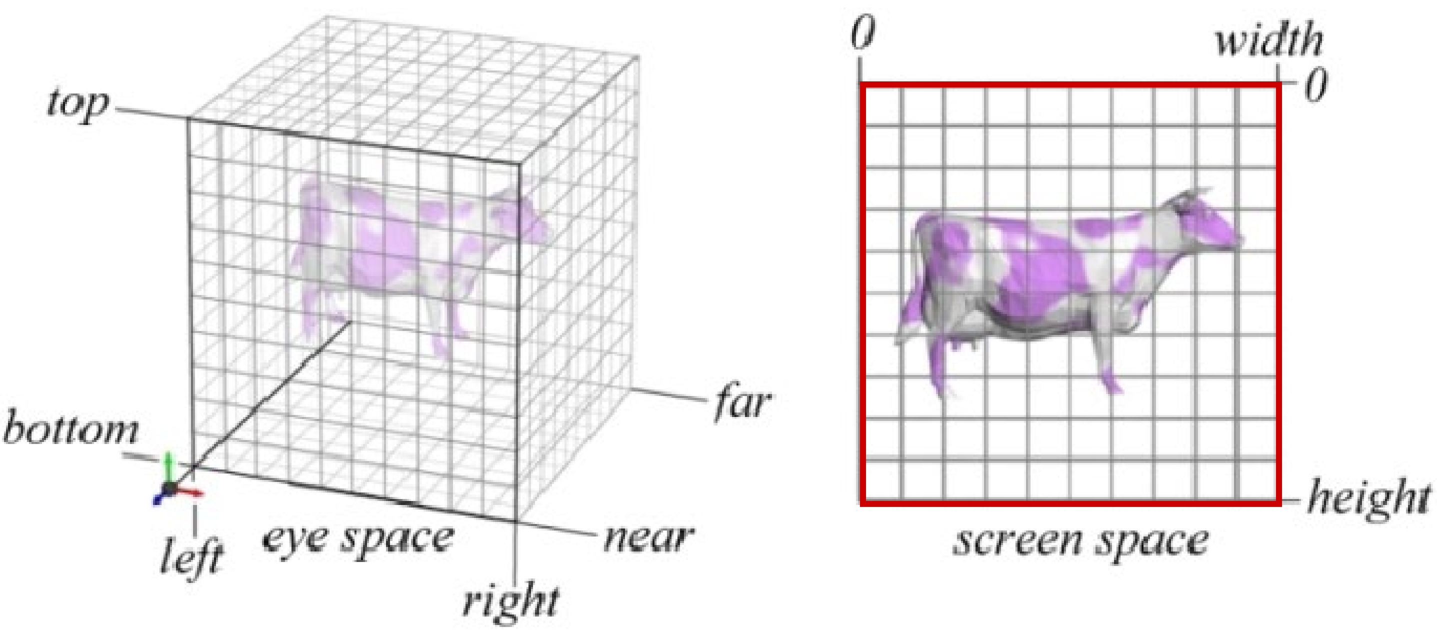
# Questions?



*Above & left:*
Buster's representation of Van Gogh's *Sunflowers* is a good example of invertism. From a purely visual perspective, the brown mark at the top of the work clearly represents the dark line which defines the edge of the table and the bottom of the vase, as shown in the photograph (left), while the blue marks represent the flowers.
However, biologists interpret these blue marks as territorial and similar in function to the arrowhead paw marks cats make to demarcate their feces. In the painting these marks signify ownership of the inverted object and are thought to have the function of rendering its unfamiliarity 'safe.'

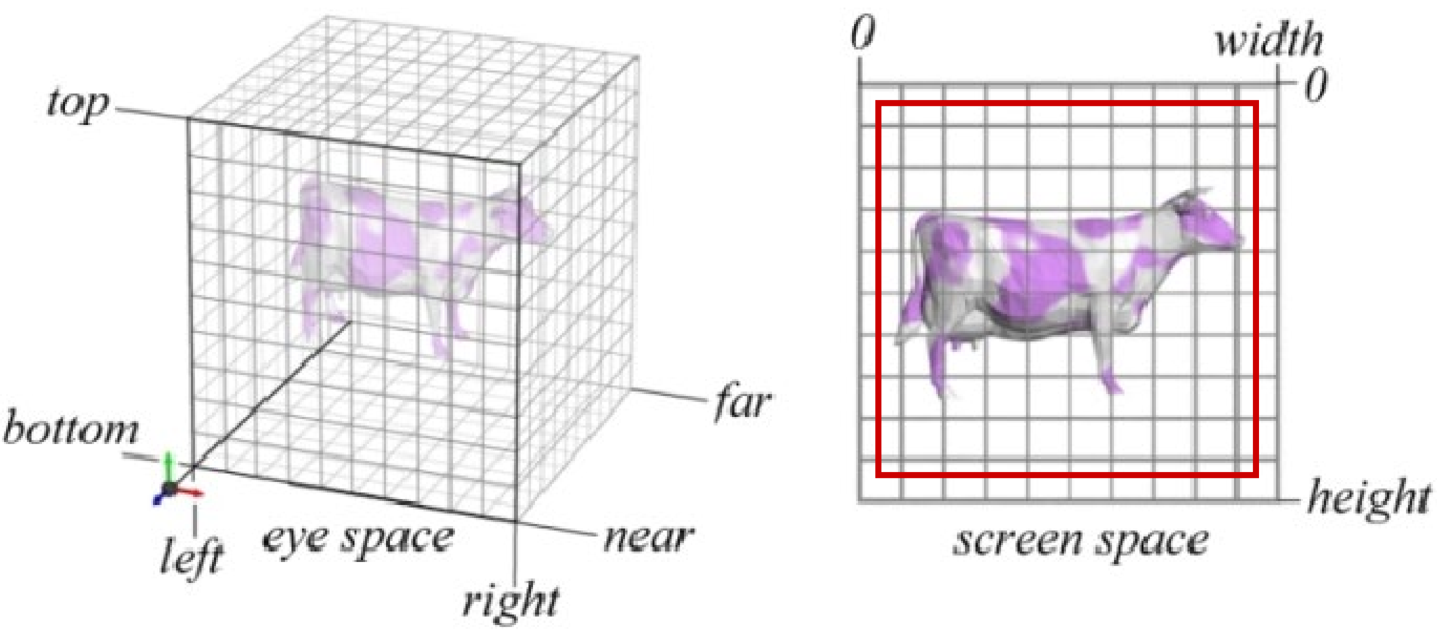
# The infamous half pixel

- I refuse to teach it, but it's an annoying issue you should know about
- Do a line drawing of a rectangle from [top, right] to [bottom,left]
- Do we actually draw the columns/rows of pixels?



# The infamous half pixel

- Displace by half a pixel so that top, right, bottom, left are in the middle of pixels
- Just change the viewport transform



# The Graphics Pipeline

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

**Input:**

*Geometric model:*
Description of all object, surface, and light source geometry and transformations

*Lighting model:*
Computational description of object and light properties, interaction (reflection)

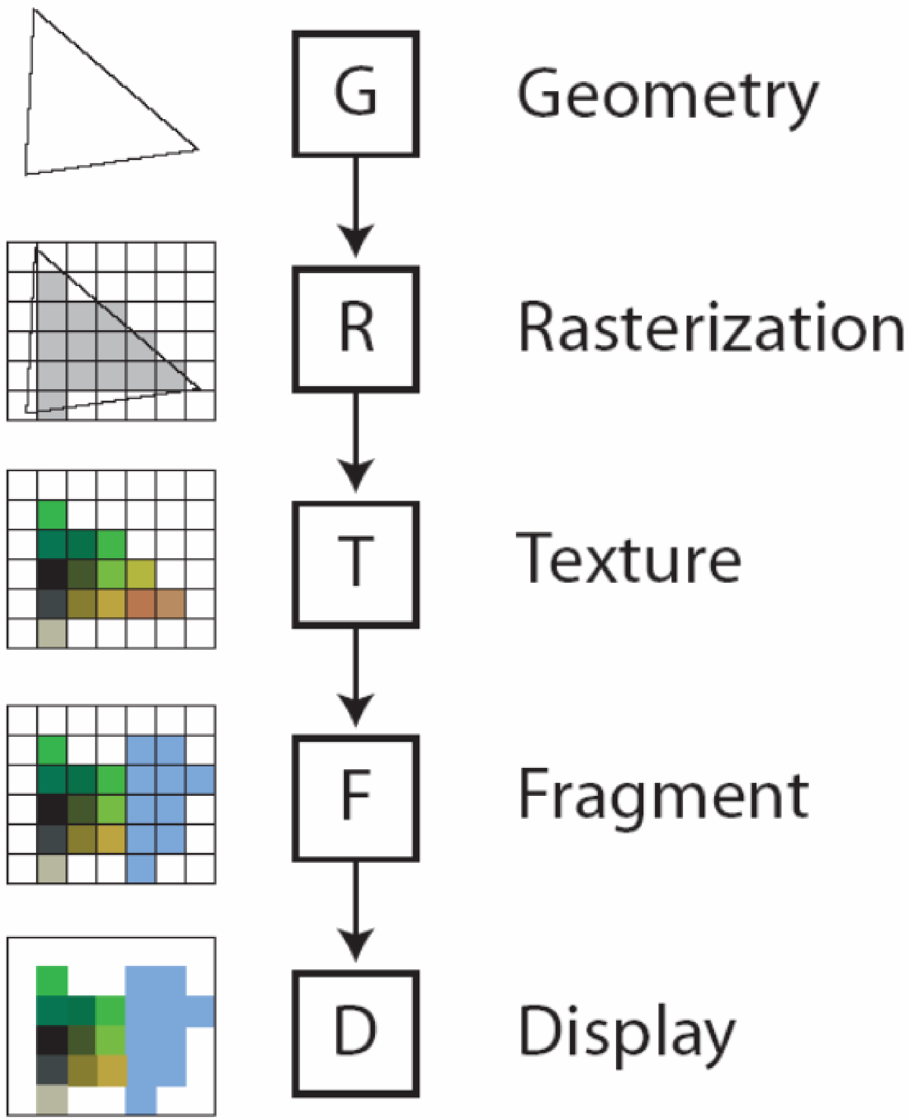
*Synthetic Viewpoint* (or *Camera*):
Eye position and viewing frustum

*Raster Viewport:*
Pixel grid onto which image plane is mapped

**Output:**

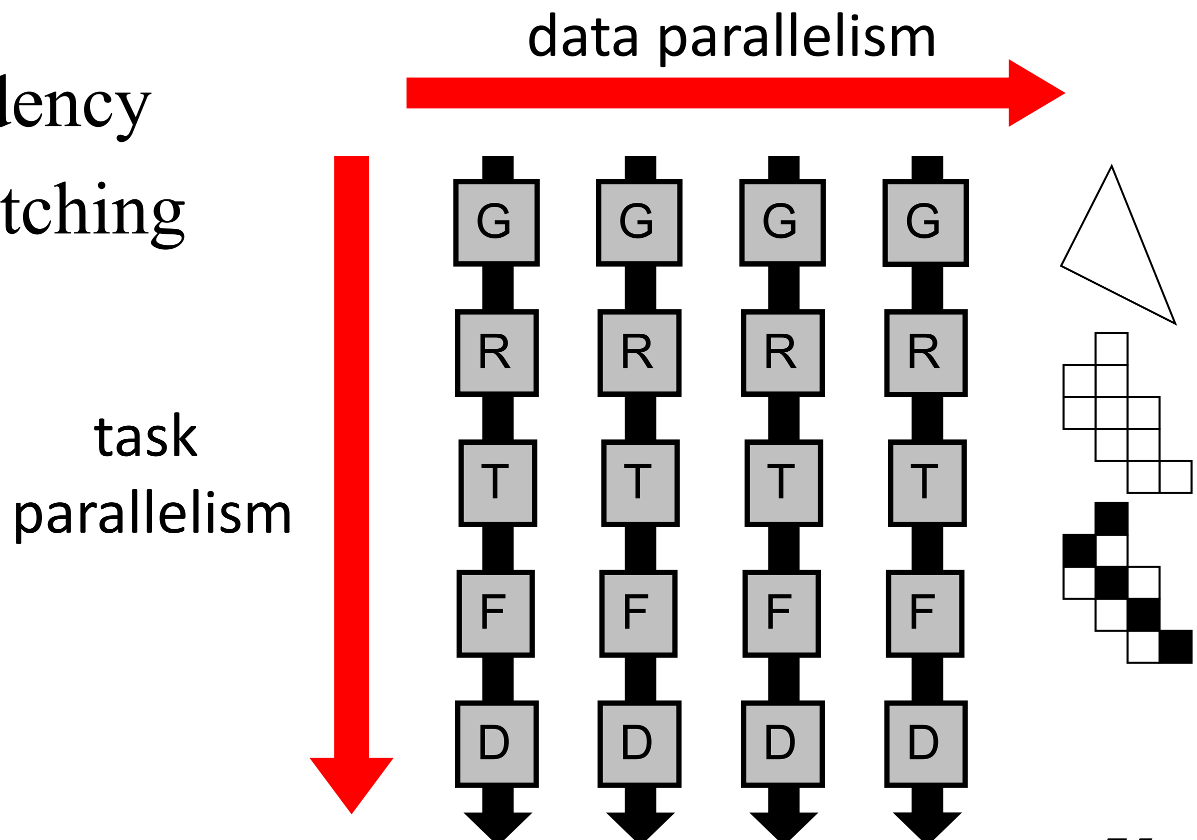
*Colors/Intensities* suitable for framebuffer display
(For example, 24-bit RGB value at each pixel)

# Modern Graphics Hardware

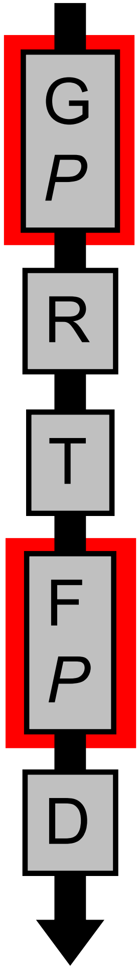


# Graphics Hardware

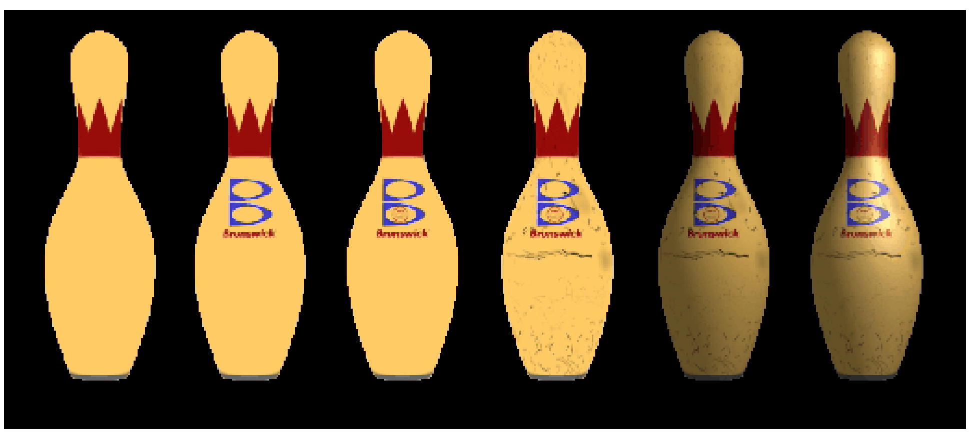
- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching



# Programmable Graphics Hardware




- Geometry and pixel (fragment) stage become programmable
  - Elaborate appearance
  - More and more general-purpose computation (GPU hacking)



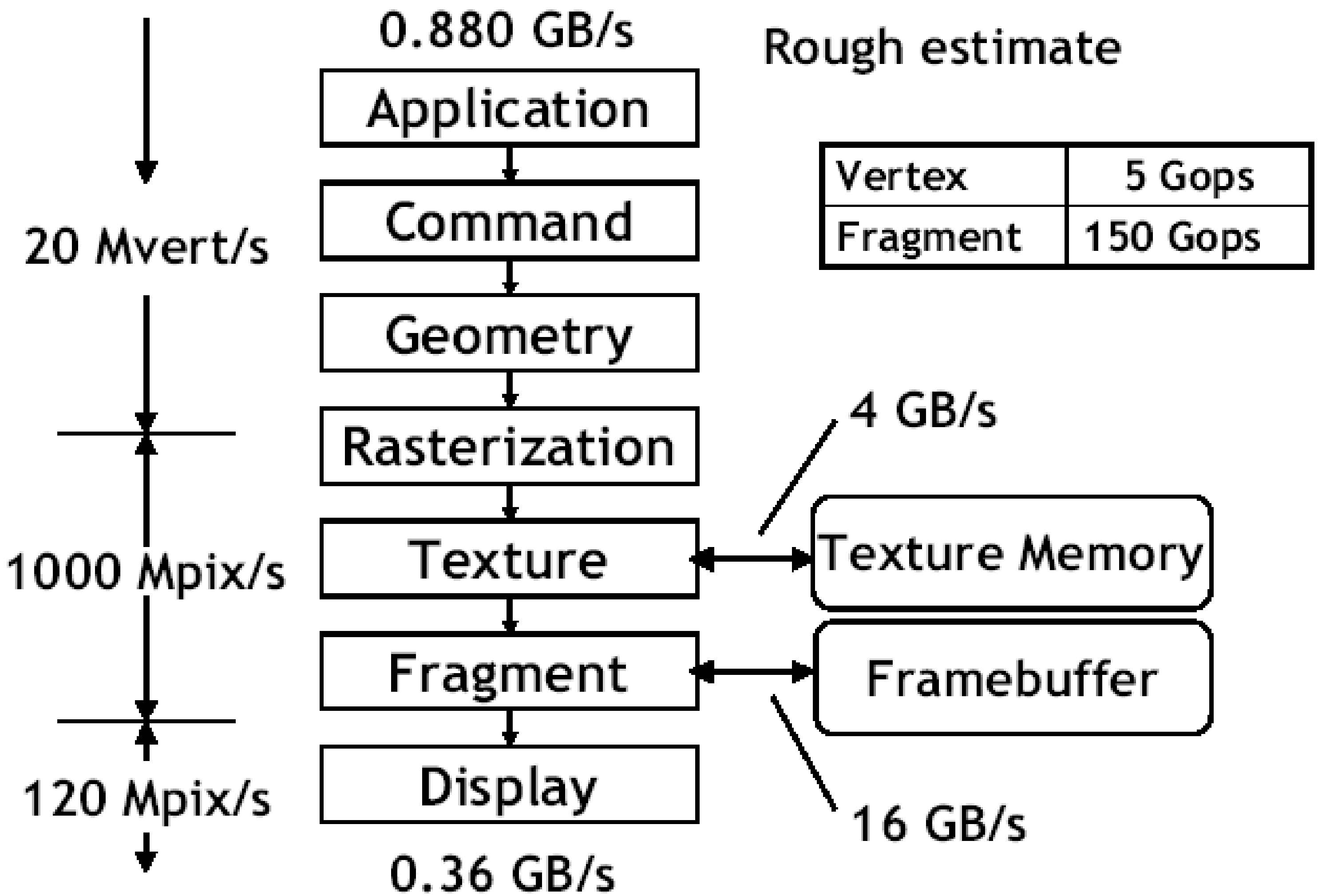
# Modern Graphics Hardware


- About 4-6 geometry units
- About 16 fragment units
- Deep pipeline (~800 stages)
- Tiling (about 4x4)
  - Early z-rejection if entire tile is occluded
- Pixels rasterized by quads (2x2 pixels)
  - Allows for derivatives
- Very efficient texture pre-fetching
  - And smart memory layout

# Current GPUs

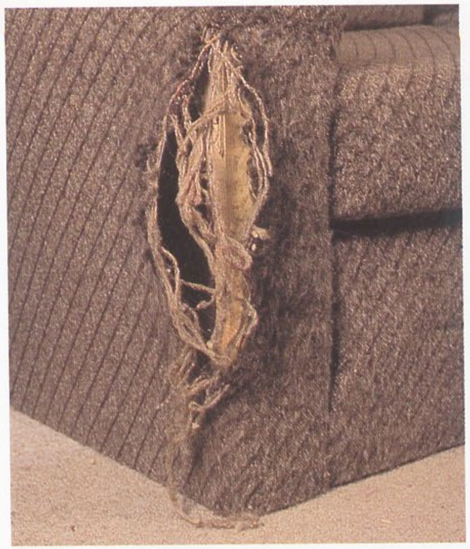
- Programmable geometry and fragment stages
- 600 million vertices/second, 6 billion texels/second
- In the range of tera operations/second
- Floating point operations only
- Very little cache

# Computational Requirements

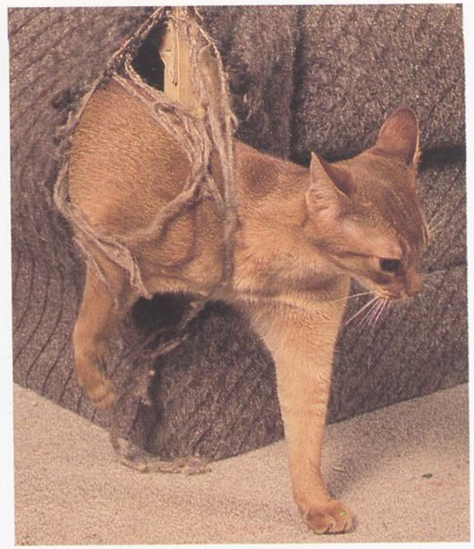



[Akeley, Hanrahan]

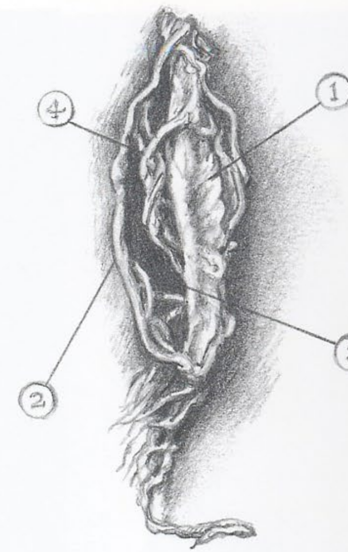
# Questions?



*Above:*
Bonny, *Come On In*, 1990. Appropriated lounge chair, 92 x 89 x 78cm. Private collection.



*Above:*
Clyde interacts with his sister's sculpture, allowing his whole body to become implicated in its heavily nuanced form.




*Above:*
Interpretive diagram by Peter Muxlow:
1. Tail form.
2. Erogenous edging.
3. Ovoidal aperture.
4. Restrictive vine forms.

"The synthetic fiber has been carefully frayed to resemble the texture and color of a cat's tail in the upright welcoming position—inviting, yet guarding the entrance beyond. However, this controlling tail is itself compromised by restrictive vines so that the whole erogenously edged aperture hints at pleasure tinged with the possibility of entanglement."
Muxlow, M. *Clawstraphobias*. Exhibition catalog, Drexel Gallery of Non-Primate Art, Philadelphia, 1992.

# Next Week: Ray Casting Acceleration