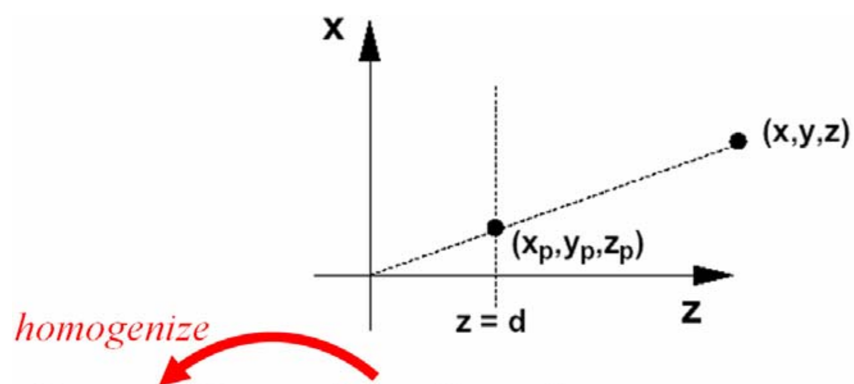
The Graphics Pipeline: Line Clipping & Line Rasterization

Last Time?

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

- Ray Tracing vs. Scan Conversion
- Overview of the Graphics Pipeline
- Projective Transformations



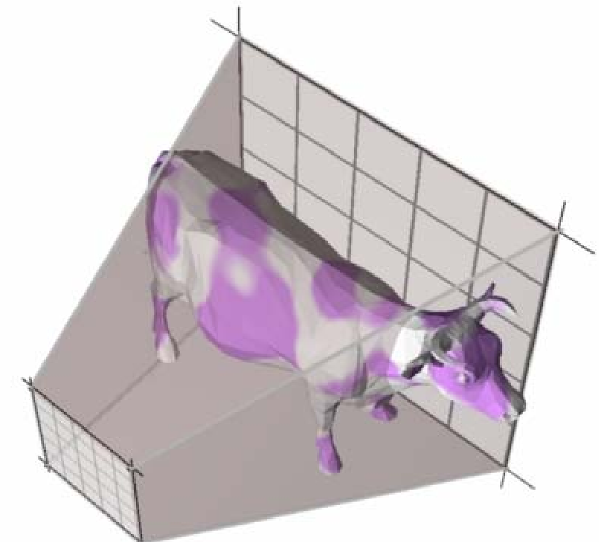
$$\begin{pmatrix} x * d / z \\ y * d / z \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

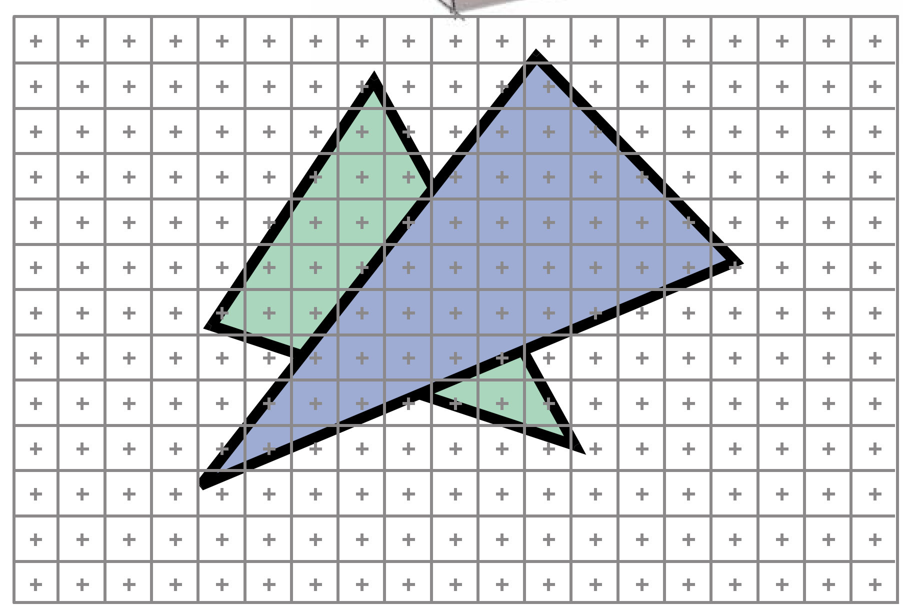
Questions?

Today: Line Clipping & Rasterization

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

- Portions of the object outside the view frustum are removed
- Rasterize objects into pixels



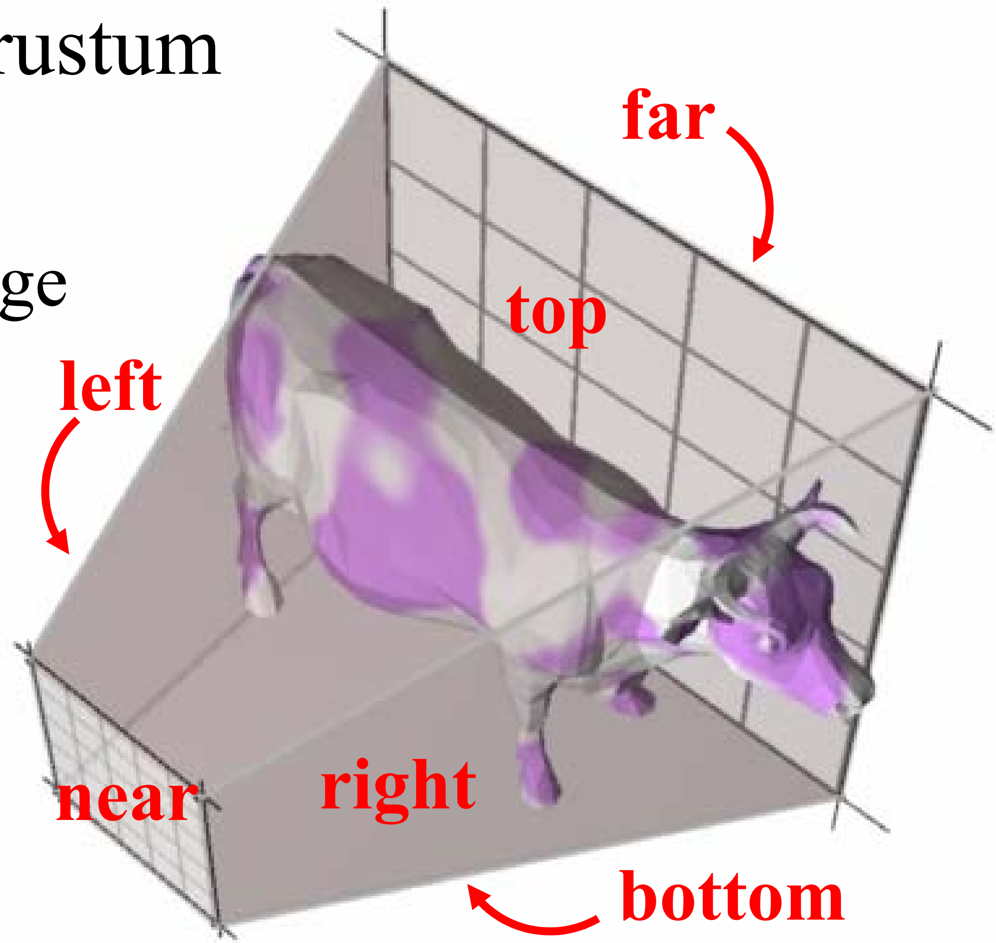


Today

- Why Clip?
- Line Clipping
- Overview of Rasterization
- Line Rasterization
- Circle Rasterization
- Antialiased Lines

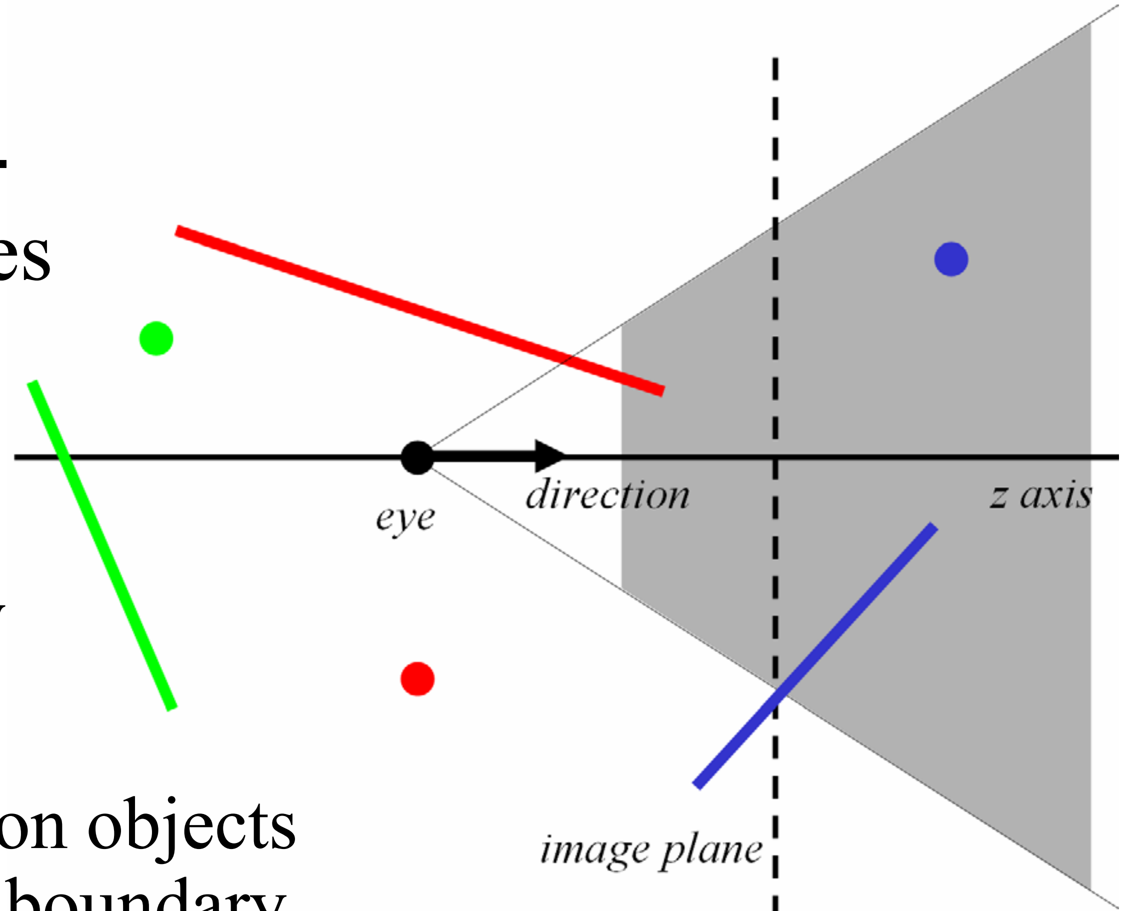
Clipping

- Eliminate portions of objects outside the viewing frustum
- View Frustum
  - boundaries of the image plane projected in 3D
  - a near & far clipping plane
- User may define additional clipping planes



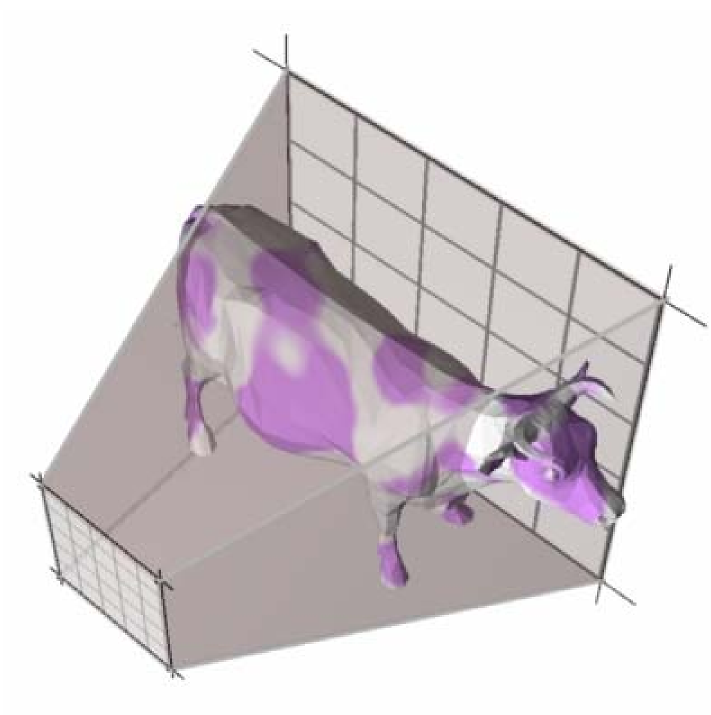
Why clip?

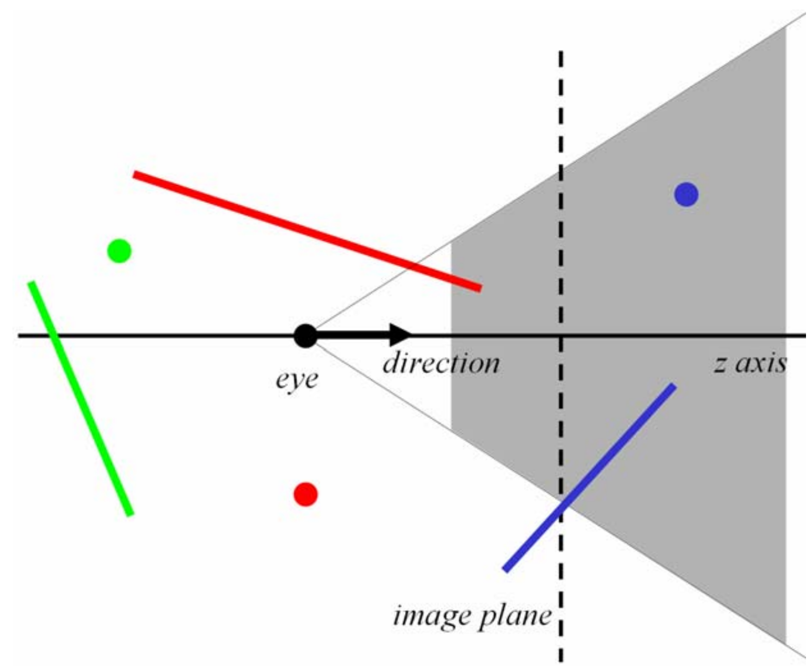
- Avoid degeneracies
  - Don’t draw stuff behind the eye
  - Avoid division by 0 and overflow
- Efficiency
  - Don’t waste time on objects outside the image boundary
- Other graphics applications (often non-convex)
  - Hidden-surface removal, Shadows, Picking, Binning, CSG (Boolean) operations (2D & 3D)



Clipping strategies

- Don't clip (and hope for the best)
- Clip on-the-fly during rasterization
- Analytical clipping: alter input geometry





Questions?

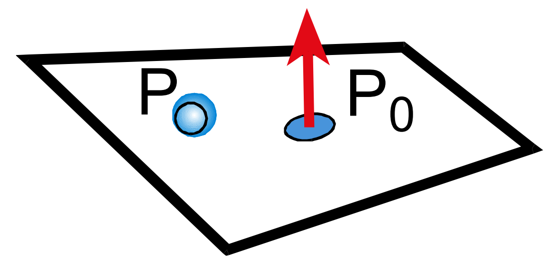
Today

- Why Clip?
- Point & Line Clipping
  - Plane – Line intersection
  - Segment Clipping
  - Acceleration using outcodes
- Overview of Rasterization
- Line Rasterization
- Circle Rasterization
- Antialiased Lines

Implicit 3D Plane Equation

- Plane defined by:

  point *p* & normal *n* OR
  normal *n* & offset *d* OR
  3 points



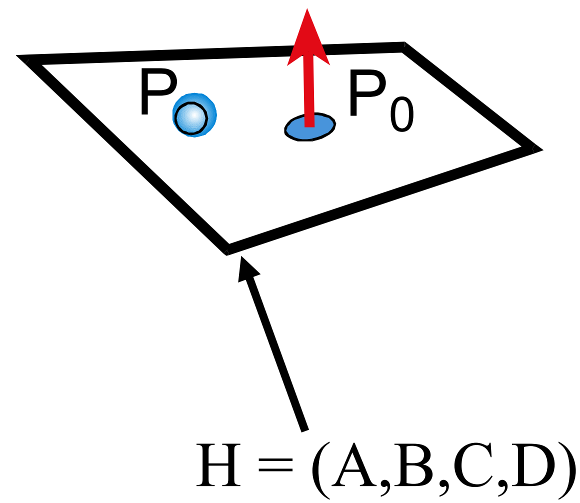
- Implicit plane equation

  Ax+By+Cz+D = 0

Homogeneous Coordinates

- Homogenous point: (x,y,z,w)

  infinite number of equivalent homogenous coordinates: (sx, sy, sz, sw)



- Homogenous Plane Equation:

  $Ax+By+Cz+D = 0 \rightarrow H = (A,B,C,D)$

  Infinite number of equivalent plane expressions:

  $sAx+sBy+sCz+sD = 0 \rightarrow H = (sA,sB,sC,sD)$

Point-to-Plane Distance

- If (A,B,C) is normalized:

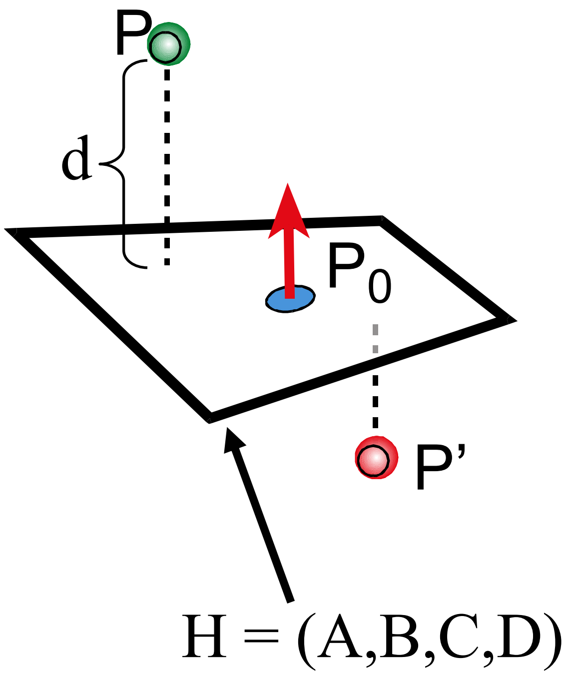
  $d = H \bullet p = H^T p$

  (the dot product in homogeneous coordinates)

- d is a *signed distance*

  positive = "inside"

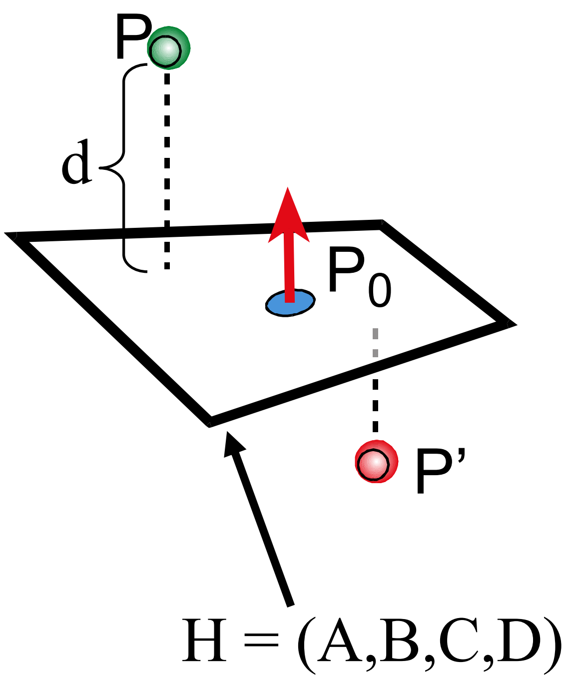
  negative = "outside"



Clipping a Point with respect to a Plane

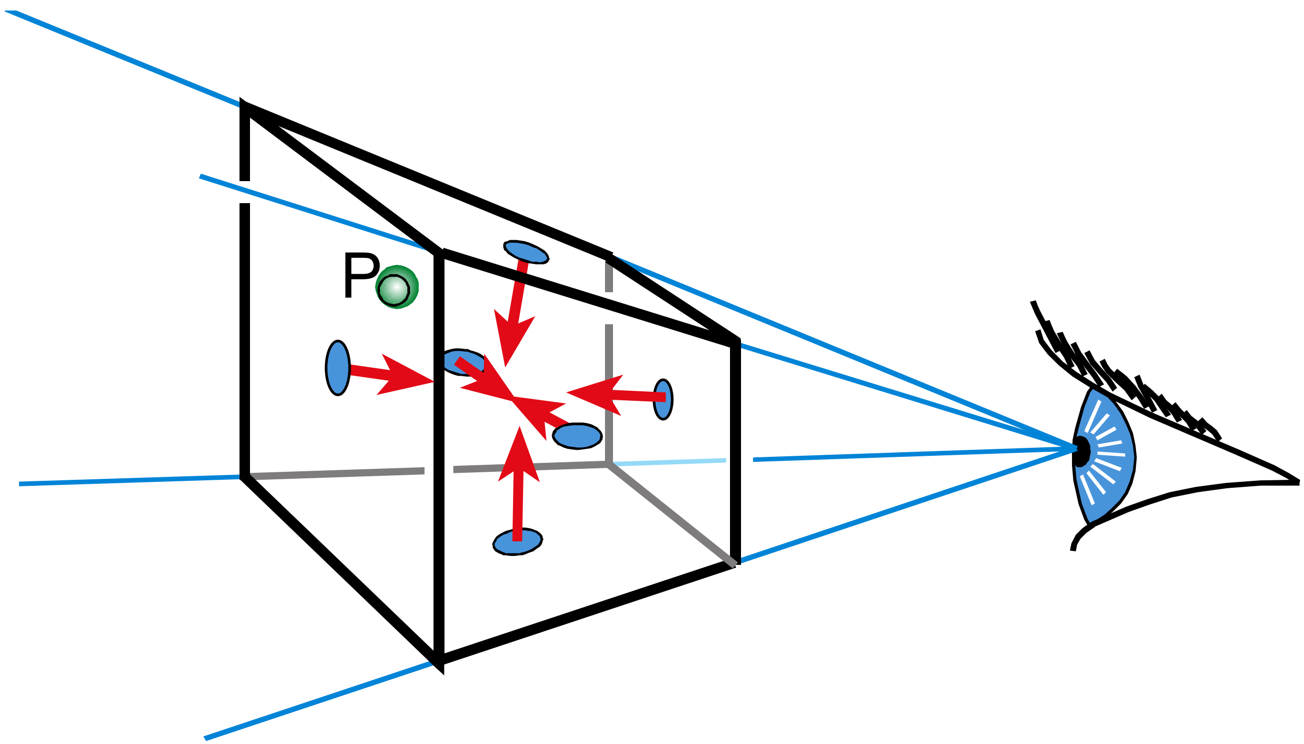
- If $d = H \bullet p \geq 0$
  Pass through

- If $d = H \bullet p < 0$:
  Clip (or cull or reject)

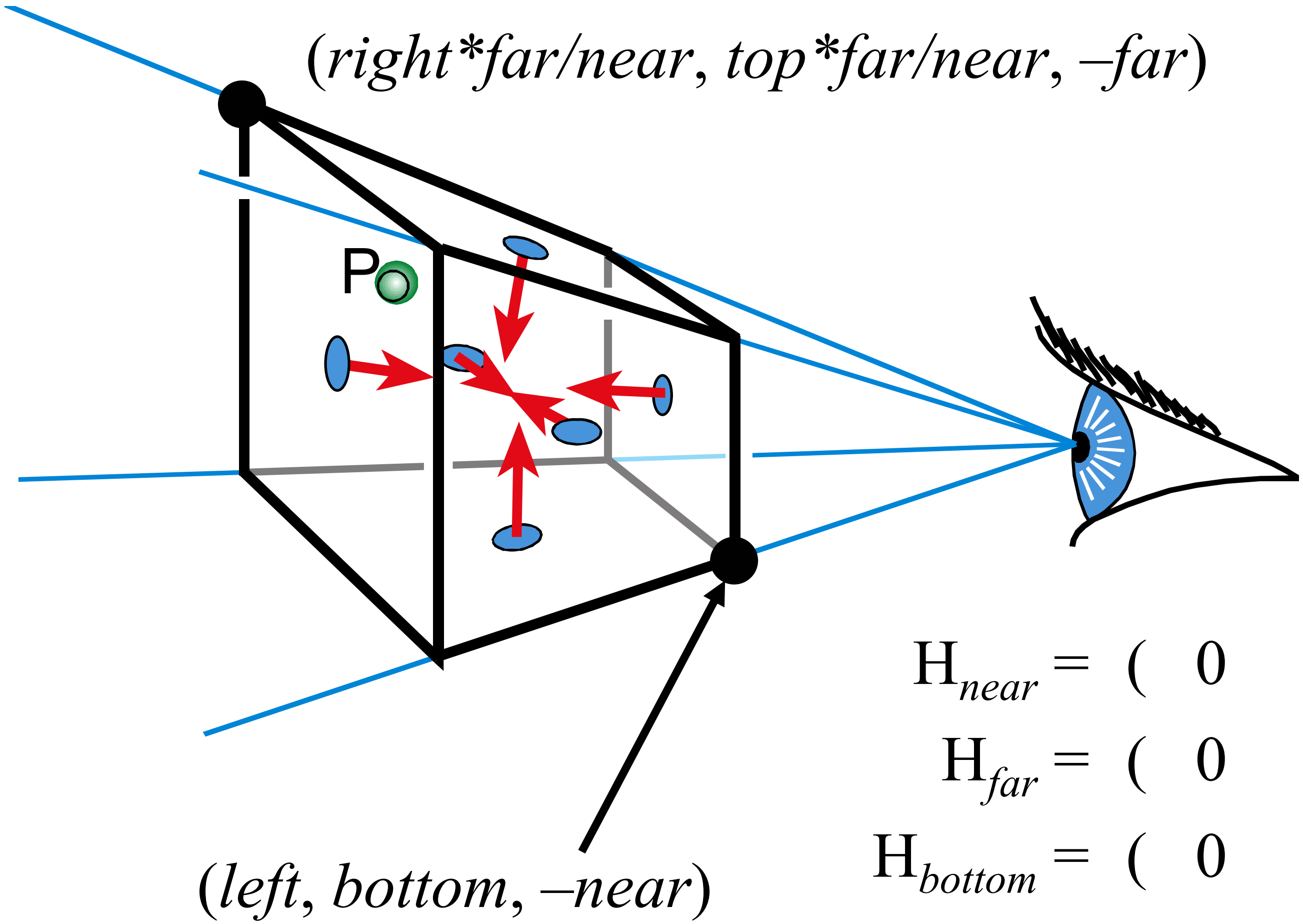


Clipping with respect to View Frustum

- Test against each of the 6 planes
  - Normals oriented towards the interior
- Clip (or cull or reject) point p if any $H \cdot p < 0$



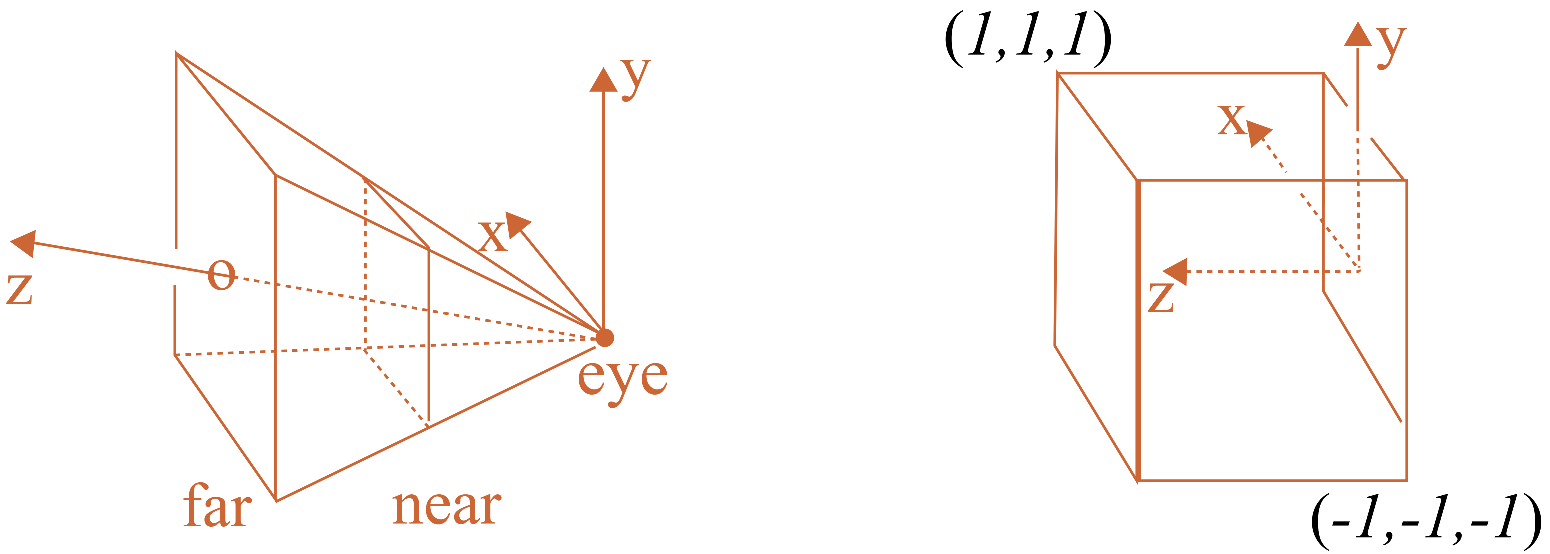
What are the View Frustum Planes?



$$
\begin{array}{lcl}
H_{near} & = & (\;0 \quad 0 \quad -1 \quad -near\;) \\
H_{far} & = & (\;0 \quad 0 \quad 1 \quad far\;) \\
H_{bottom} & = & (\;0 \quad near \quad bottom \quad 0\;) \\
H_{top} & = & (\;0 \quad -near \quad -top \quad 0\;) \\
H_{left} & = & (\;left \quad near \quad 0 \quad 0\;) \\
H_{right} & = & (\;-right \quad -near \quad 0 \quad 0\;)
\end{array}
$$

Clipping & Transformation

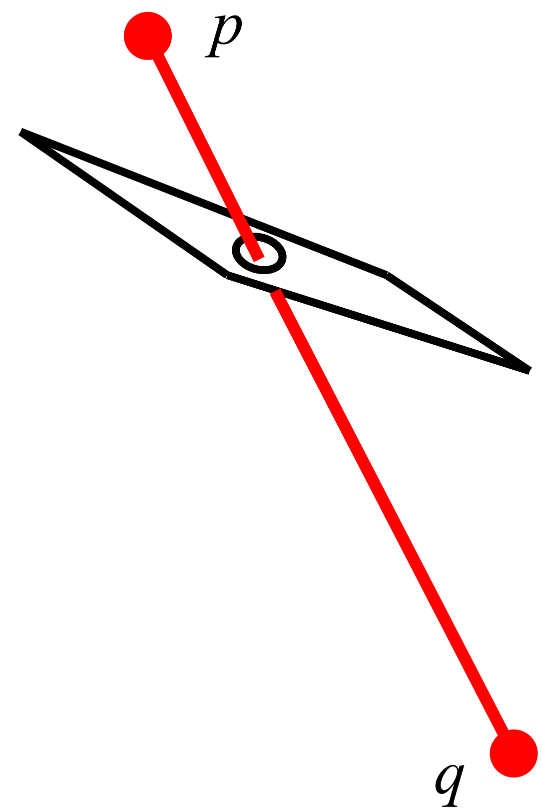
- Transform M (e.g. from world space to NDC)



- The plane equation is transformed with $(M^{-1})^T$

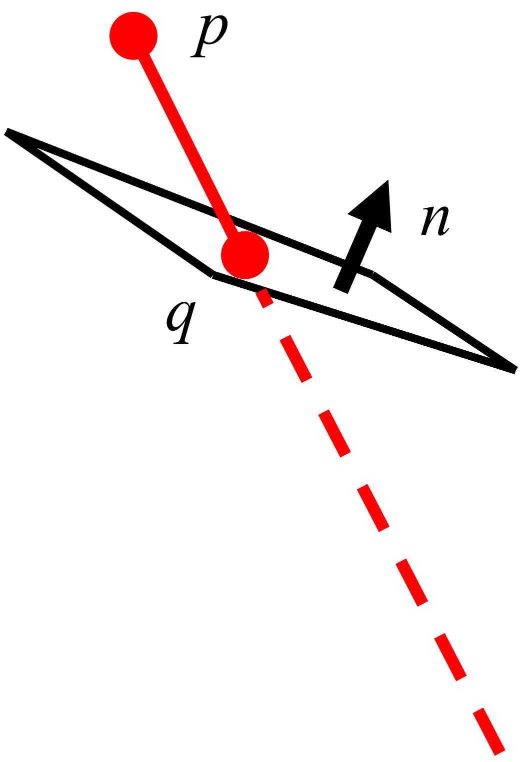
Segment Clipping

- If H•p > 0 and H•q < 0
- If H•p < 0 and H•q > 0
- If H•p > 0 and H•q > 0
- If H•p < 0 and H•q < 0



Segment Clipping

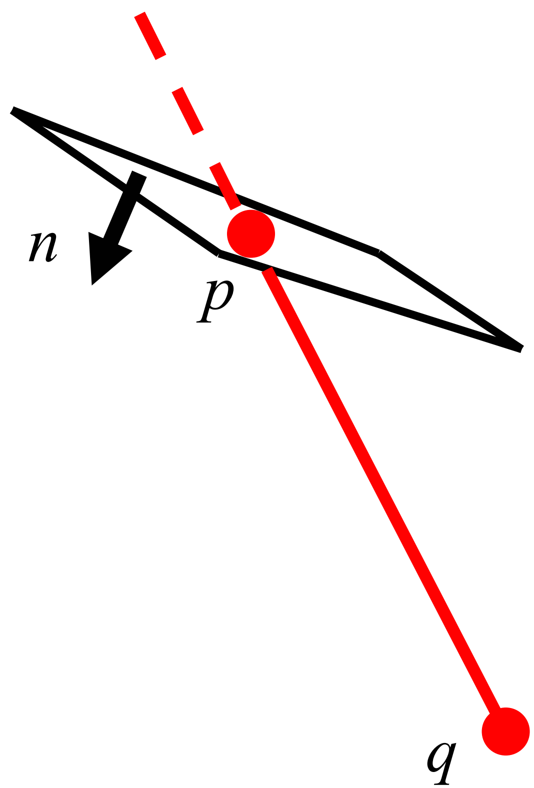
- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
- If H•p > 0 and H•q > 0
- If H•p < 0 and H•q < 0



Segment Clipping

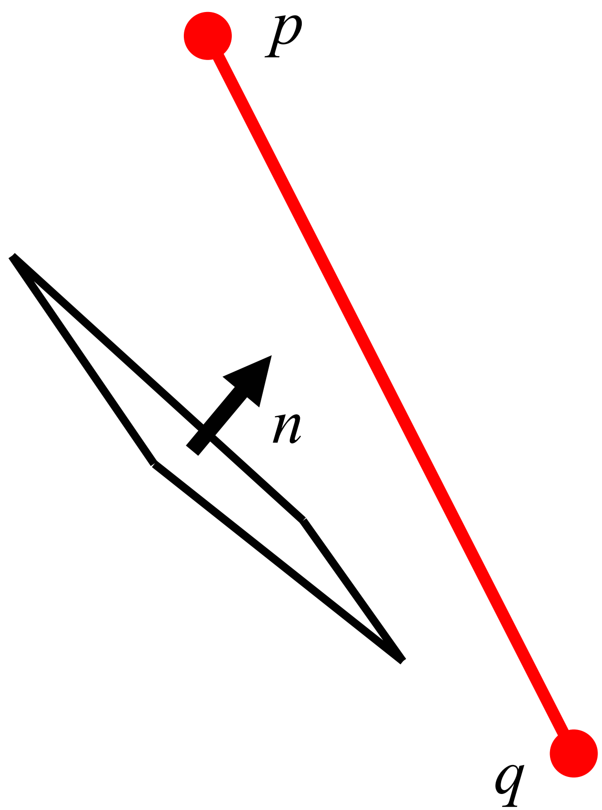
- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
  - clip p to plane
- If H•p > 0 and H•q > 0

- If H•p < 0 and H•q < 0



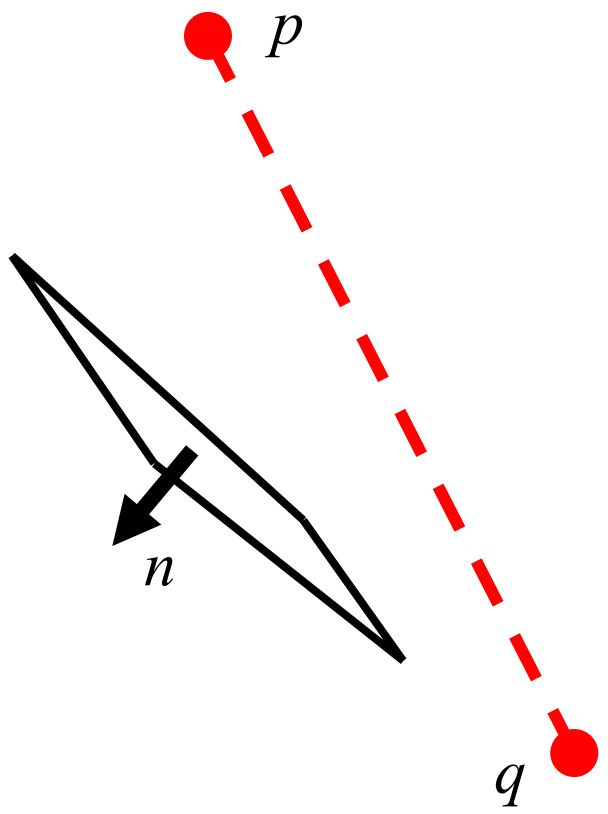
Segment Clipping

- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
  - clip p to plane
- If H•p > 0 and H•q > 0
  - pass through
- If H•p < 0 and H•q < 0



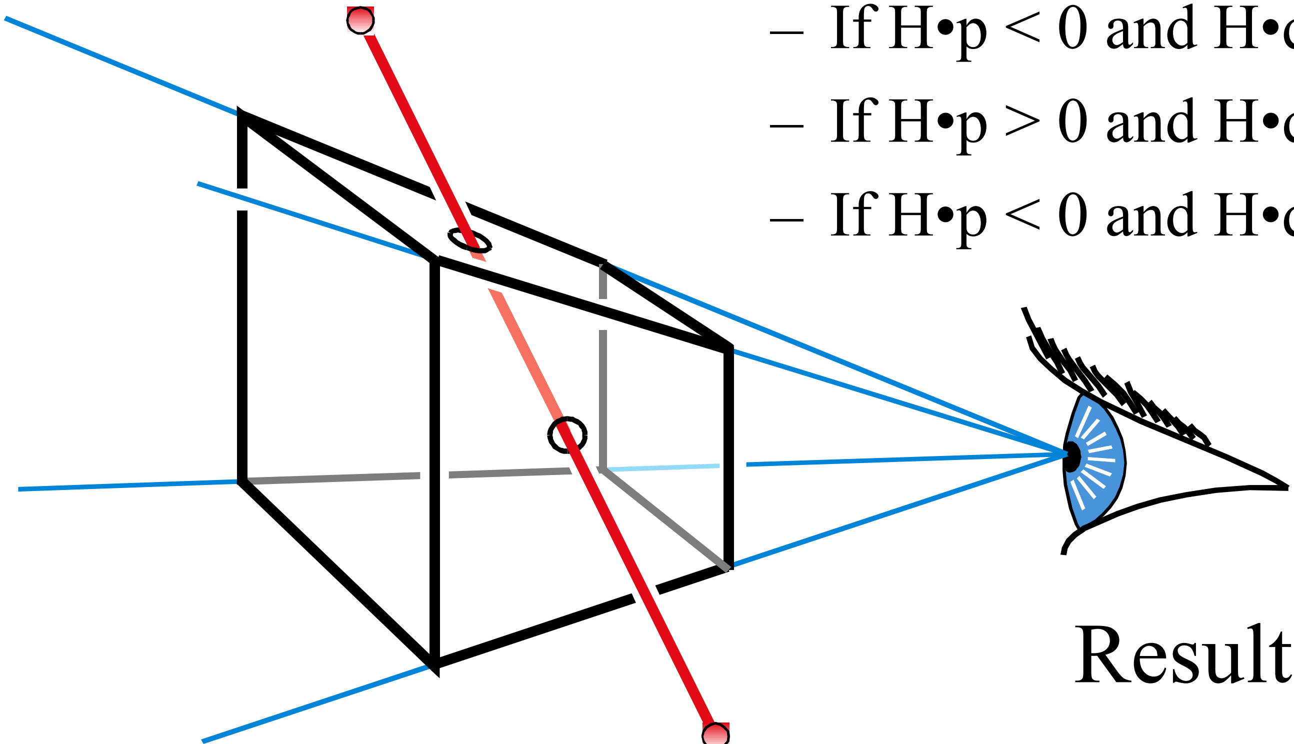
Segment Clipping

- If H•p > 0 and H•q < 0
  - clip q to plane
- If H•p < 0 and H•q > 0
  - clip p to plane
- If H•p > 0 and H•q > 0
  - pass through
- If H•p < 0 and H•q < 0
  - clipped out



Clipping against the frustum

- For each frustum plane H
  - If H•p > 0 and H•q < 0, clip q to H
  - If H•p < 0 and H•q > 0, clip p to H
  - If H•p > 0 and H•q > 0, pass through
  - If H•p < 0 and H•q < 0, clipped out



Result is a single segment. Why?

Line – Plane Intersection

- Explicit (Parametric) Line Equation

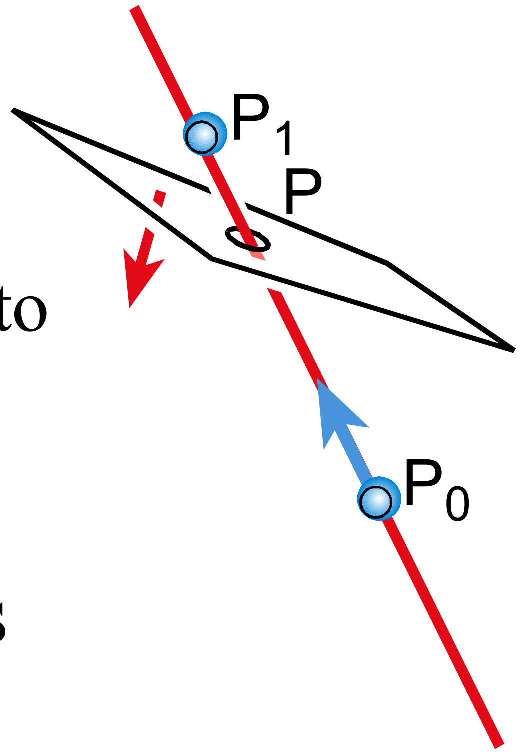
  $L(t) = P_0 + t * (P_1 - P_0)$

  $L(t) = (1-t) * P_0 + t * P_1$

- How do we intersect?

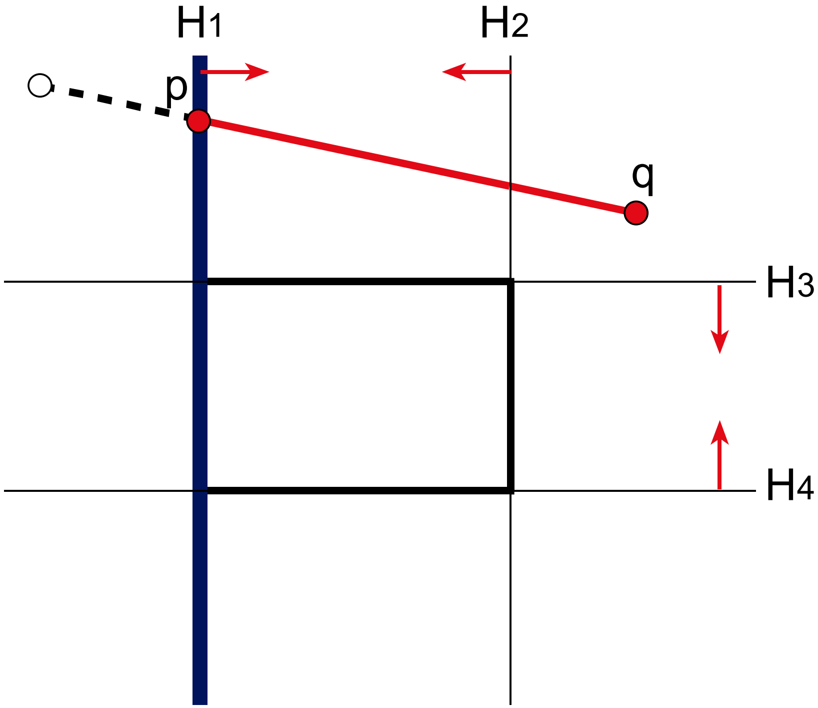
  Insert explicit equation of line into implicit equation of plane

- Parameter t is used to interpolate associated attributes (color, normal, texture, etc.)



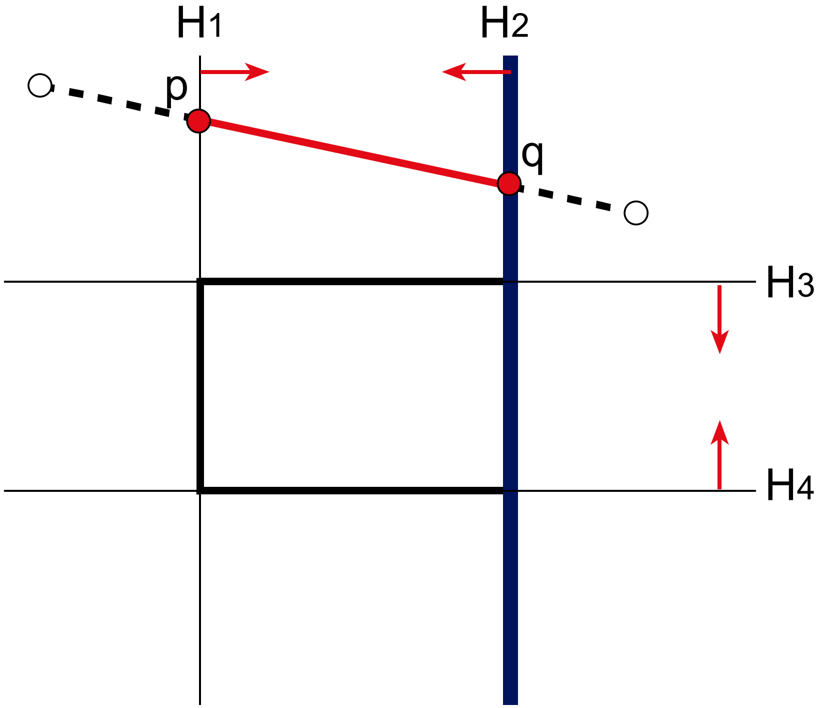
Is this Clipping Efficient?

- For each frustum plane H
  - If H•p > 0 and H•q < 0, clip q to H
  - If H•p < 0 and H•q > 0, clip p to H
  - If H•p > 0 and H•q > 0, pass through
  - If H•p < 0 and H•q < 0, clipped out



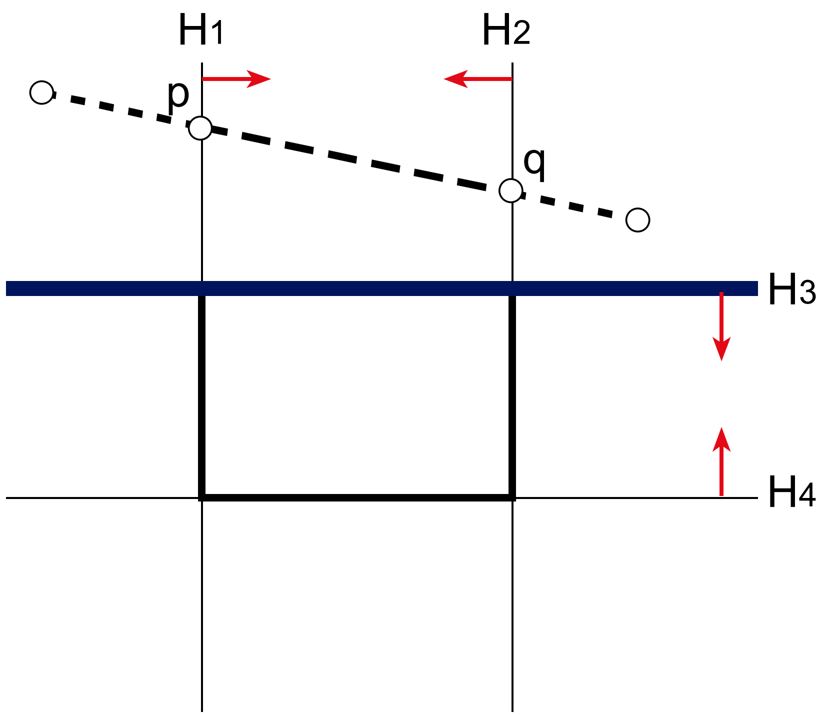
Is this Clipping Efficient?

- For each frustum plane H
  - If H•p > 0 and H•q < 0, clip q to H
  - If H•p < 0 and H•q > 0, clip p to H
  - If H•p > 0 and H•q > 0, pass through
  - If H•p < 0 and H•q < 0, clipped out



Is this Clipping Efficient?

- For each frustum plane H
  - If H•p > 0 and H•q < 0, clip q to H
  - If H•p < 0 and H•q > 0, clip p to H
  - If H•p > 0 and H•q > 0, pass through
  - If H•p < 0 and H•q < 0, clipped out



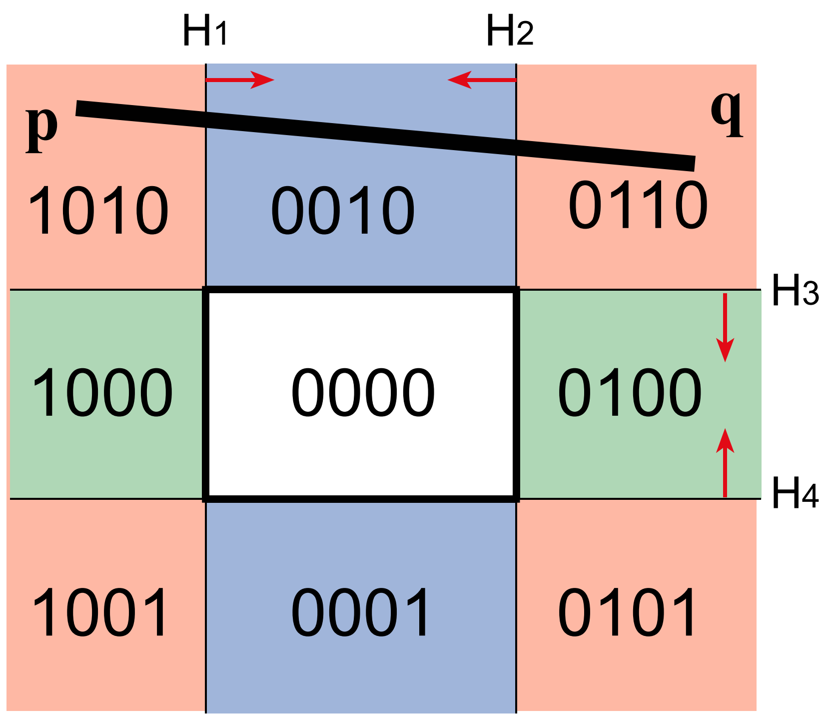
What is the problem?

The computation of the intersections, and any corresponding interpolated values is unnecessary

Can we detect this earlier?

Improving Efficiency: Outcodes

- Compute the sidedness of each vertex with respect to each bounding plane (0 = valid)
- Combine into binary outcode using logical AND



Outcode of p : 1010

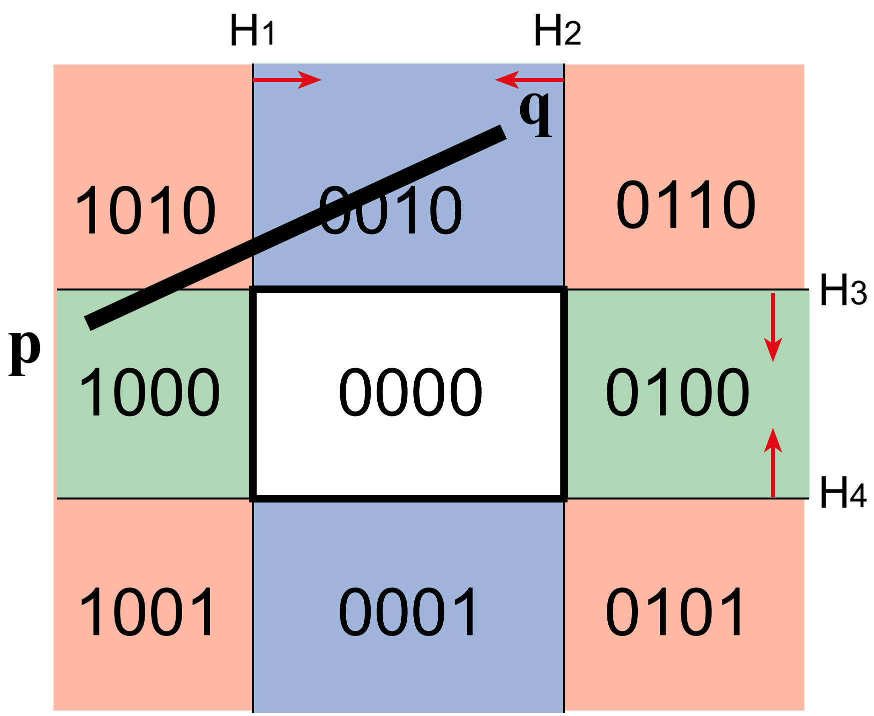
Outcode of q : 0110

Outcode of [pq] : 0010

Clipped because there is a 1

Improving Efficiency: Outcodes

- When do we fail to save computation?



Outcode of p : 1000

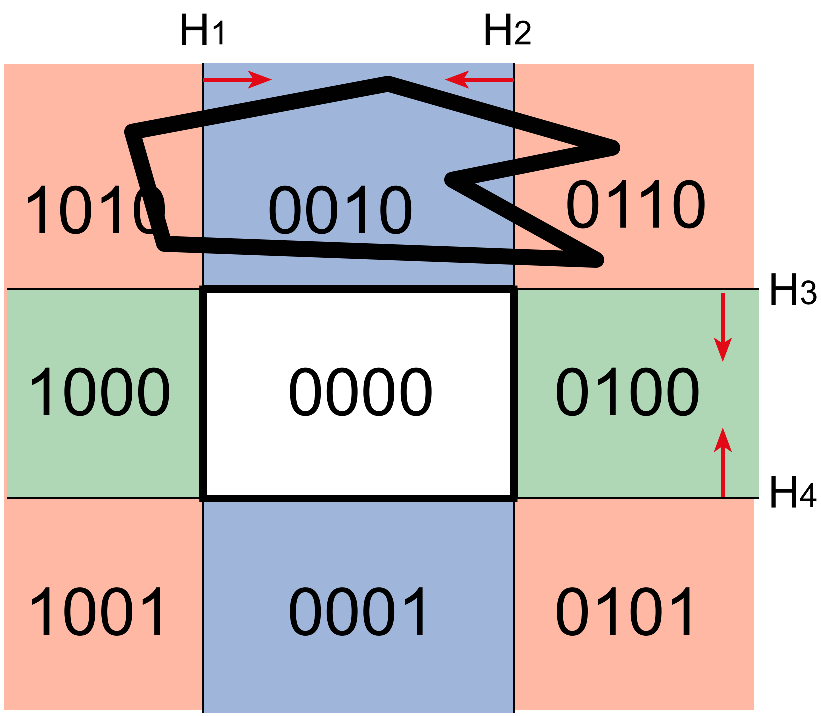
Outcode of q : 0010

Outcode of [pq] : 0000

Not clipped

Improving Efficiency: Outcodes

- It works for arbitrary primitives
- And for arbitrary dimensions



Outcode of p : 1010
Outcode of q : 1010
Outcode of r : 0110
Outcode of s : 0010
Outcode of t : 0110
Outcode of u : 0010

Outcode : 0010

Clipped

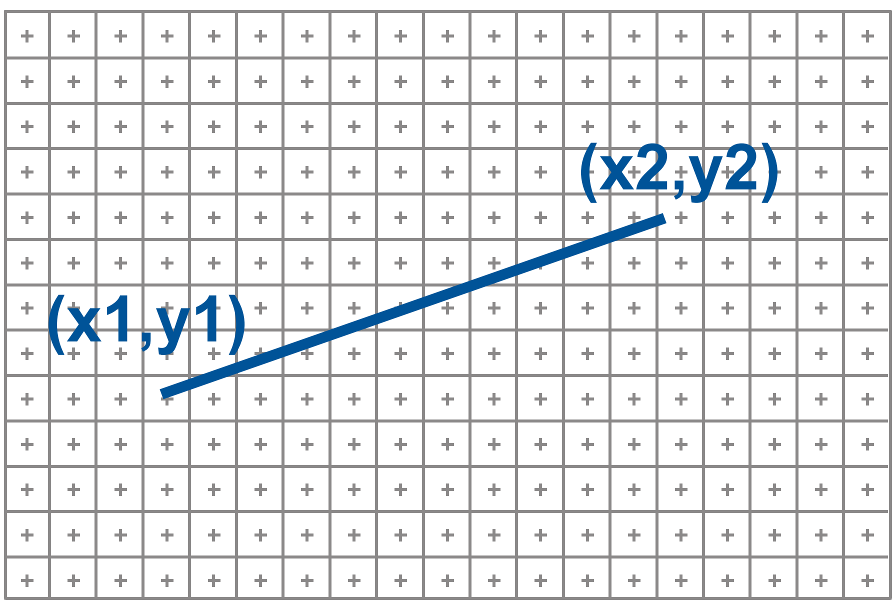
Questions?

Today

- Why Clip?
- Line Clipping
- Overview of Rasterization
- Line Rasterization
- Circle Rasterization
- Antialiased Lines

Framebuffer Model

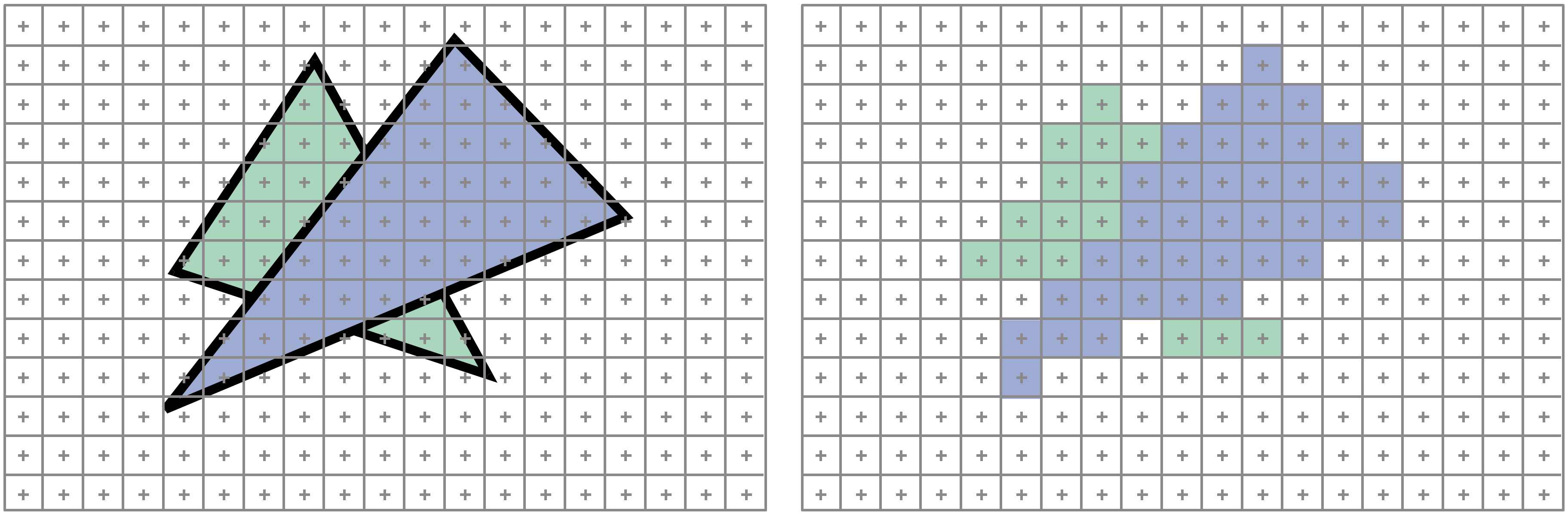
- Raster Display: 2D array of picture elements (pixels)
- Pixels individually set/cleared (greyscale, color)
- Window coordinates: pixels centered at integers




```
glBegin(GL_LINES)
glVertex3f(...)
glVertex3f(...)
glEnd();
```

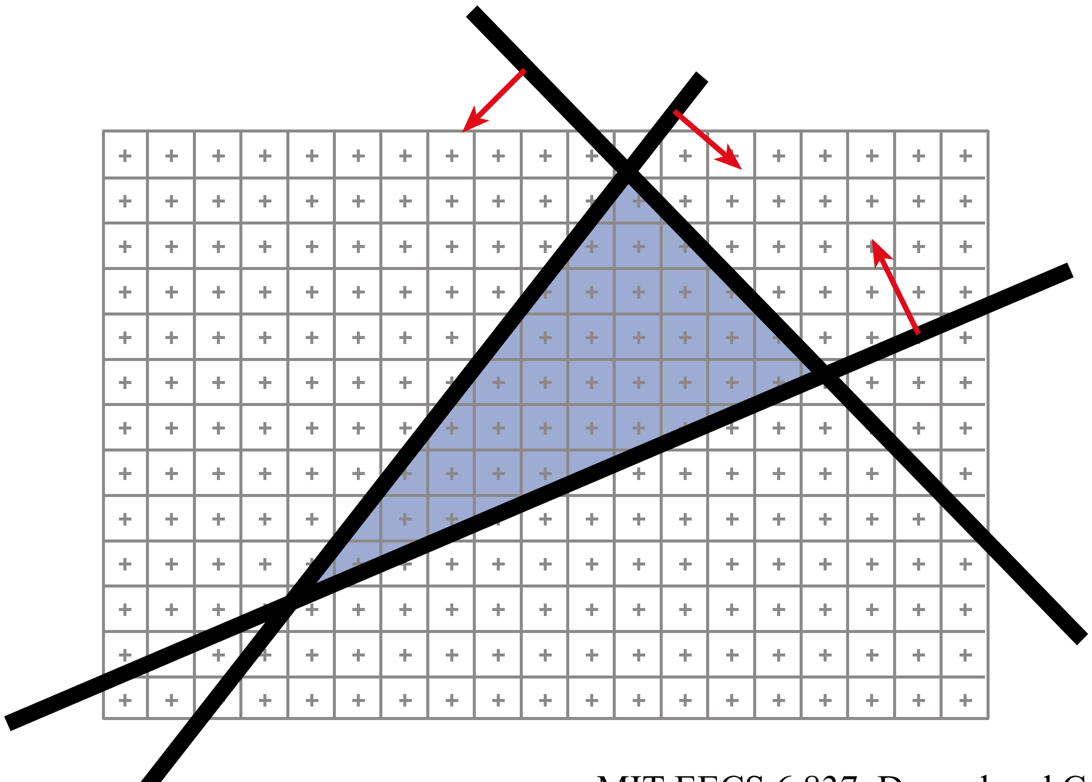
2D Scan Conversion

- Geometric primitives
  (point, line, polygon, circle, polyhedron, sphere...)
- Primitives are continuous; screen is discrete
- Scan Conversion: algorithms for *efficient* generation of the samples comprising this approximation



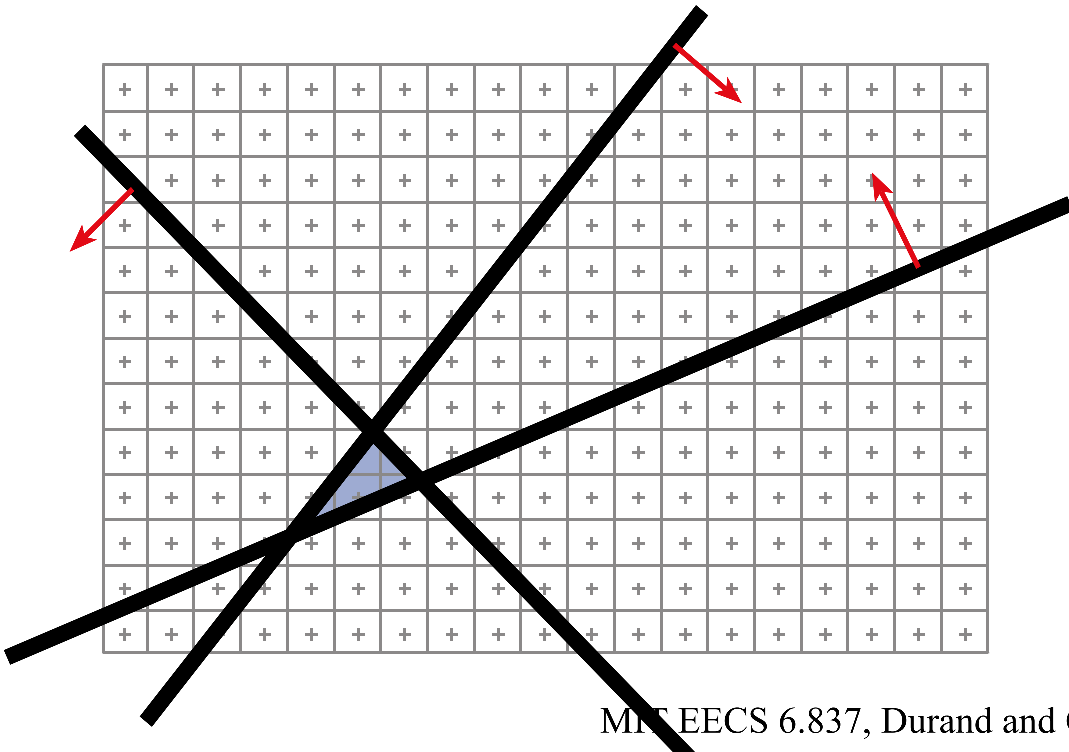
Brute force solution for triangles

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



Brute force solution for triangles

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

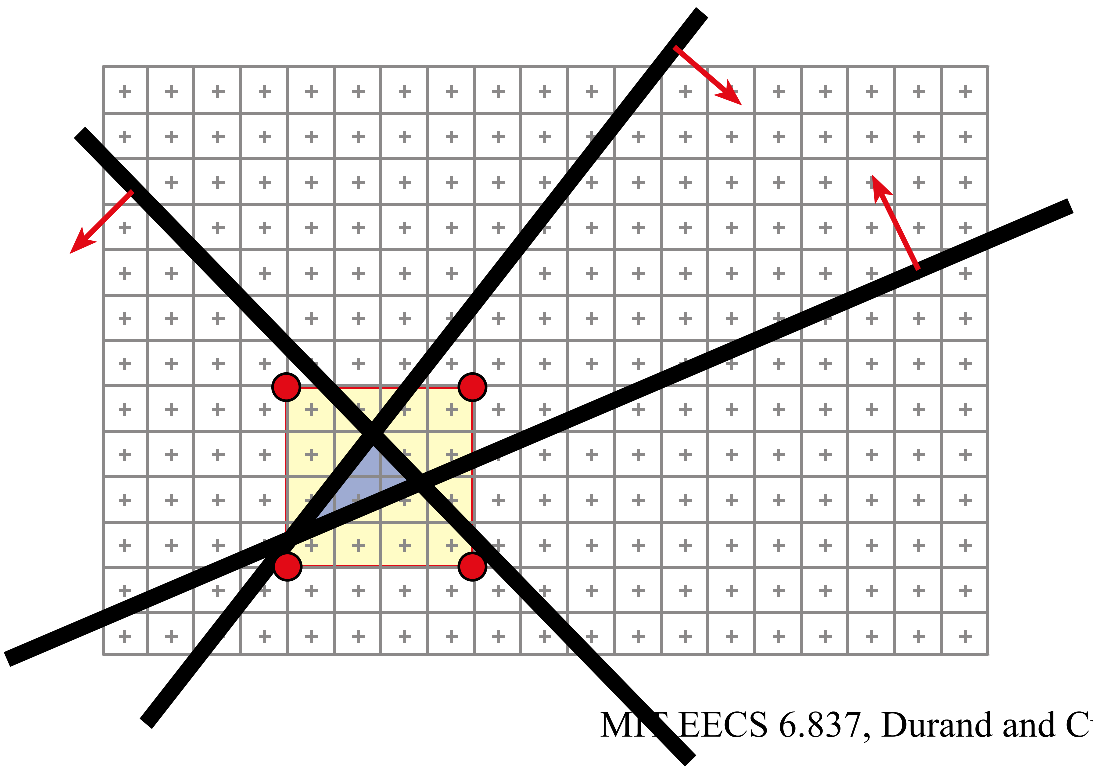


Problem?

If the triangle is small, a lot of useless computation

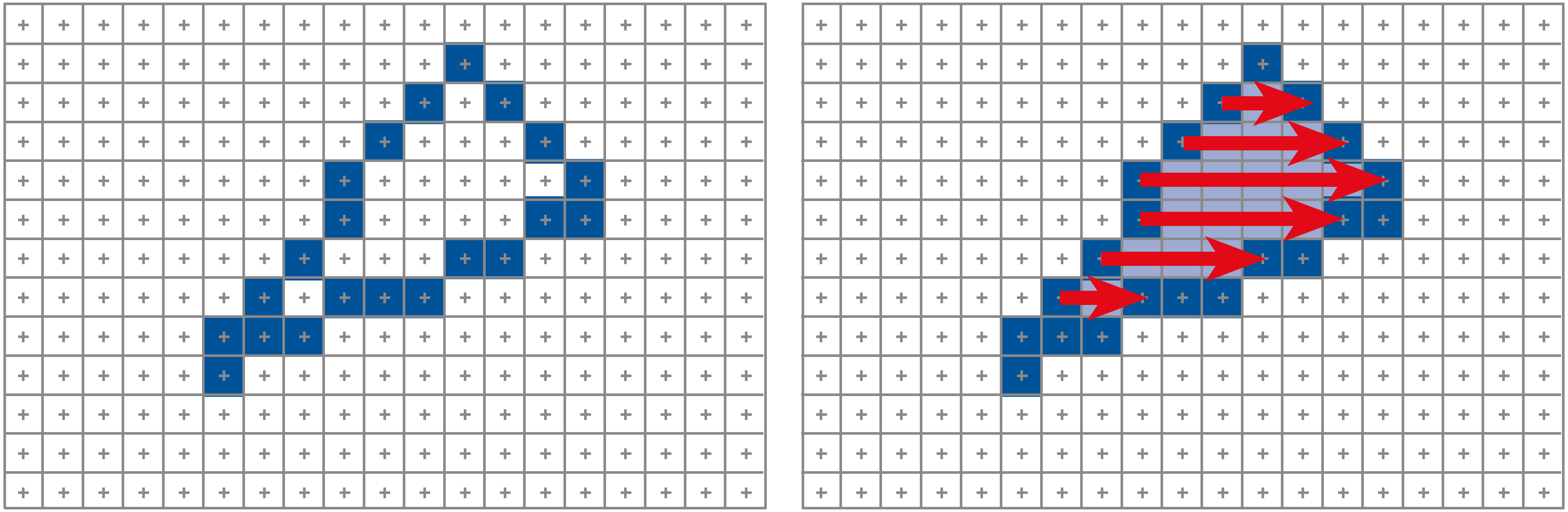
Brute force solution for triangles

- Improvement:
  - Compute only for the screen bounding box of the triangle
  - Xmin, Xmax, Ymin, Ymax of the triangle vertices



Can we do better? Yes!

- More on polygons next week.
- Today: line rasterization



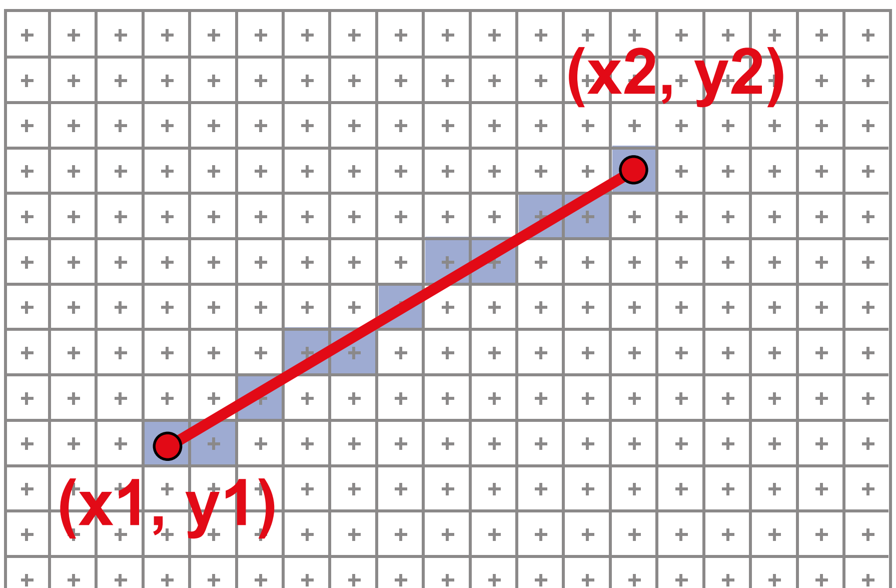
Questions?

Today

- Why Clip?
- Line Clipping
- Overview of Rasterization
- Line Rasterization
  - naive method
  - Bresenham's (DDA)
- Circle Rasterization
- Antialiased Lines

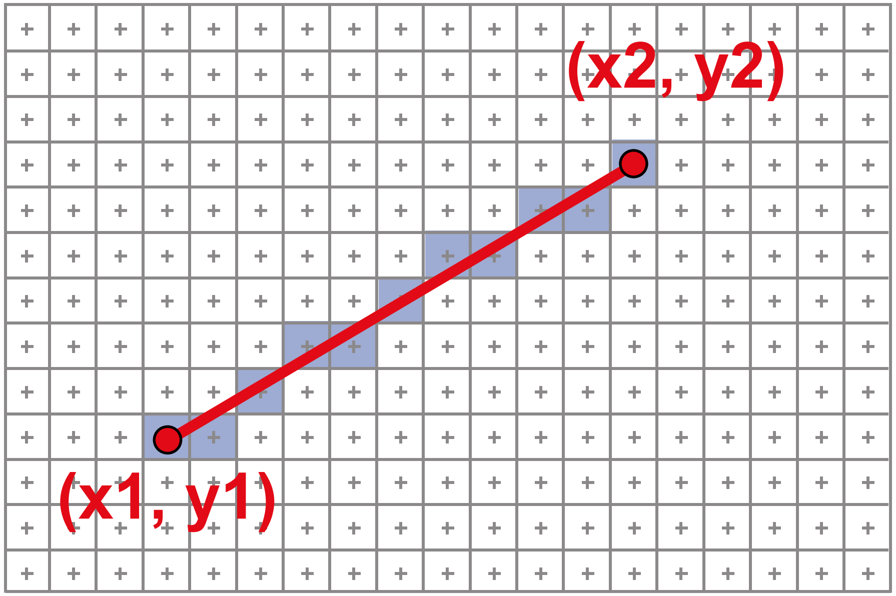
Scan Converting 2D Line Segments

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



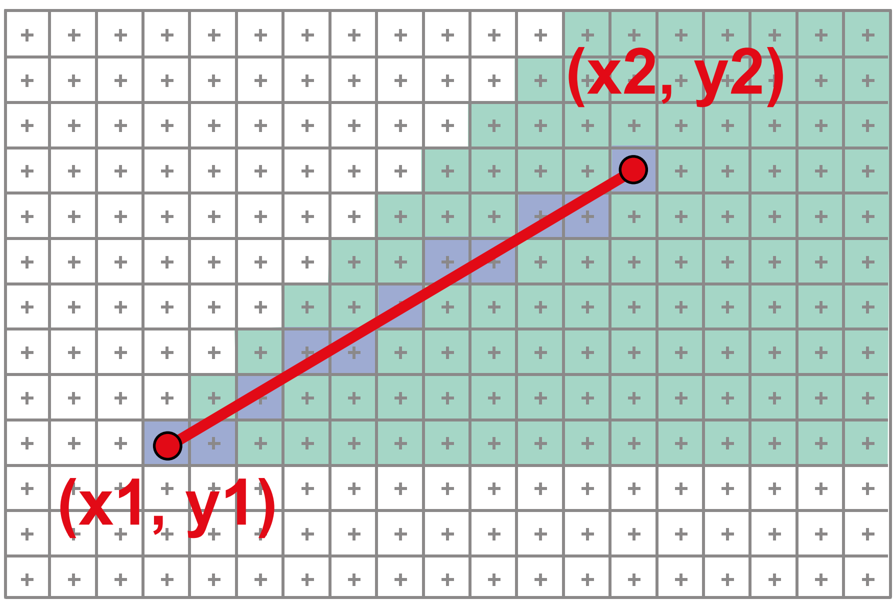
Line Rasterization Requirements

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



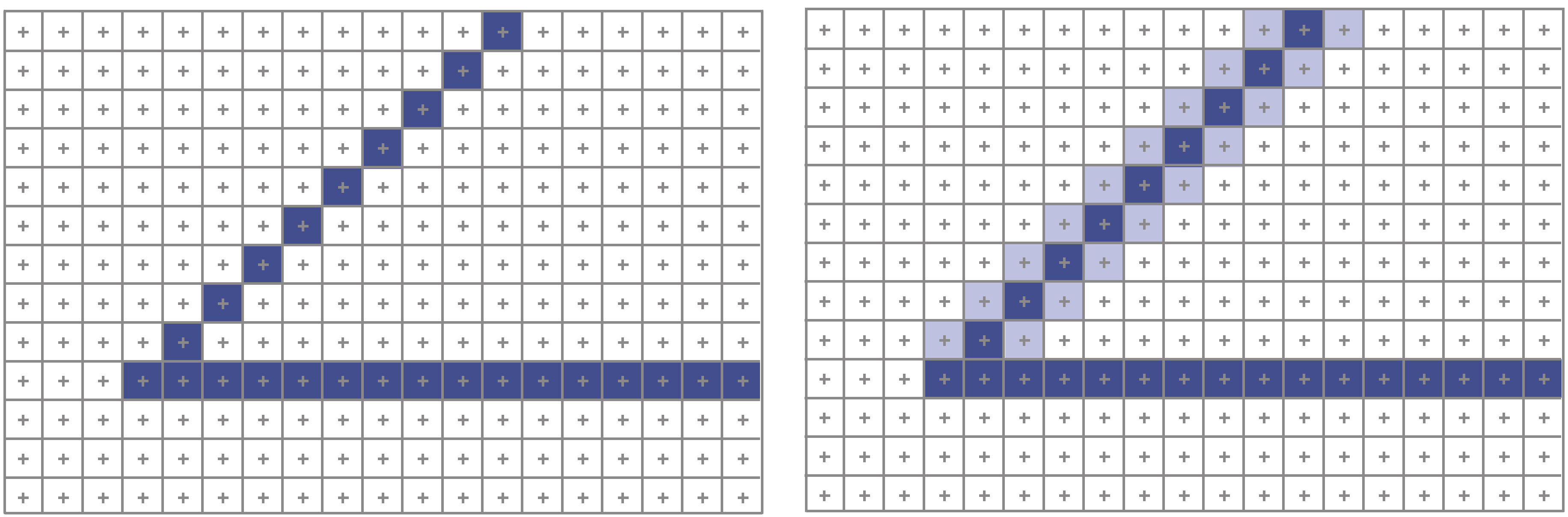
Algorithm Design Choices

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



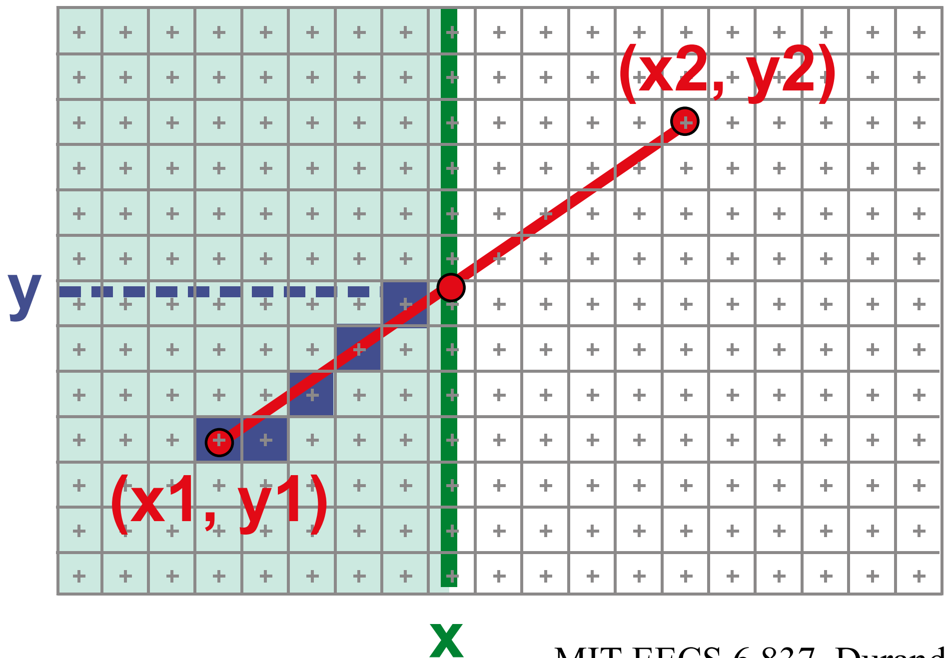
Algorithm Design Choices

- Note: brightness can vary with slope
  - What is the maximum variation? $\sqrt{2}$
- How could we compensate for this?
  - Answer: antialiasing



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)))



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

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

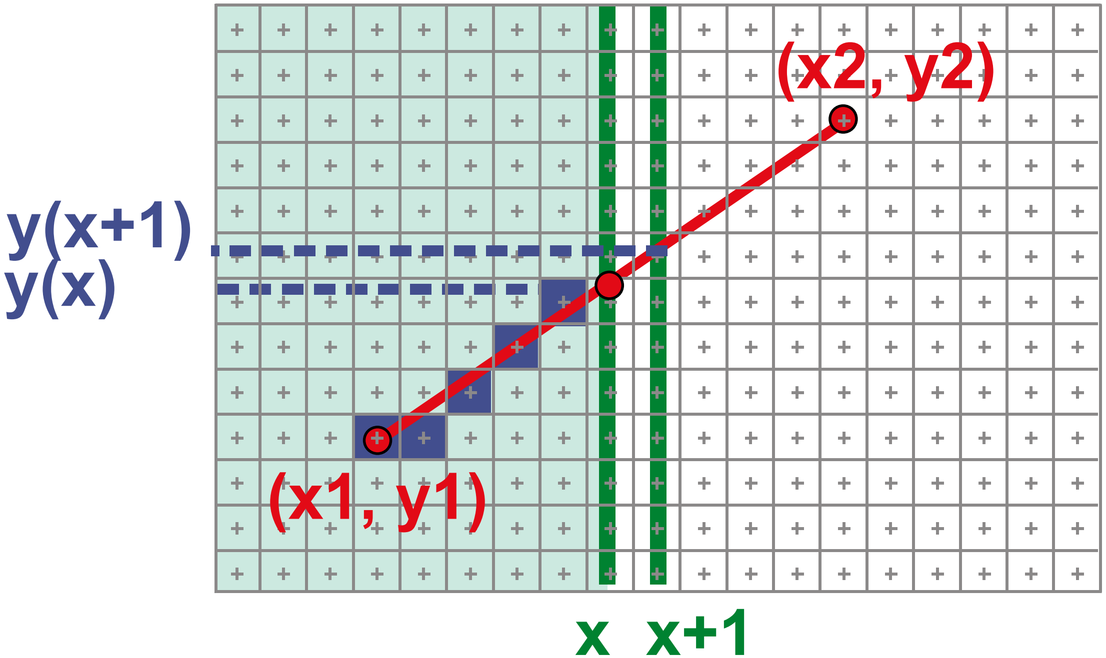
$$m = \frac{dy}{dx}$$

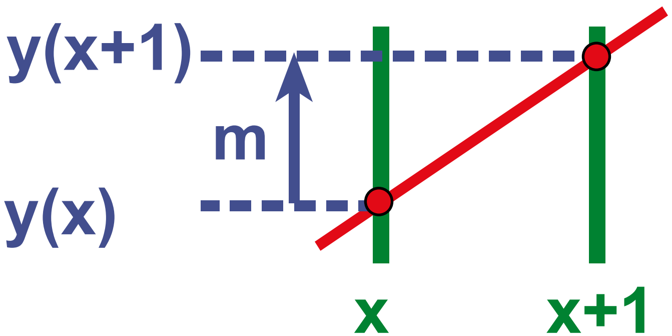
Efficiency

- Computing y value is expensive

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

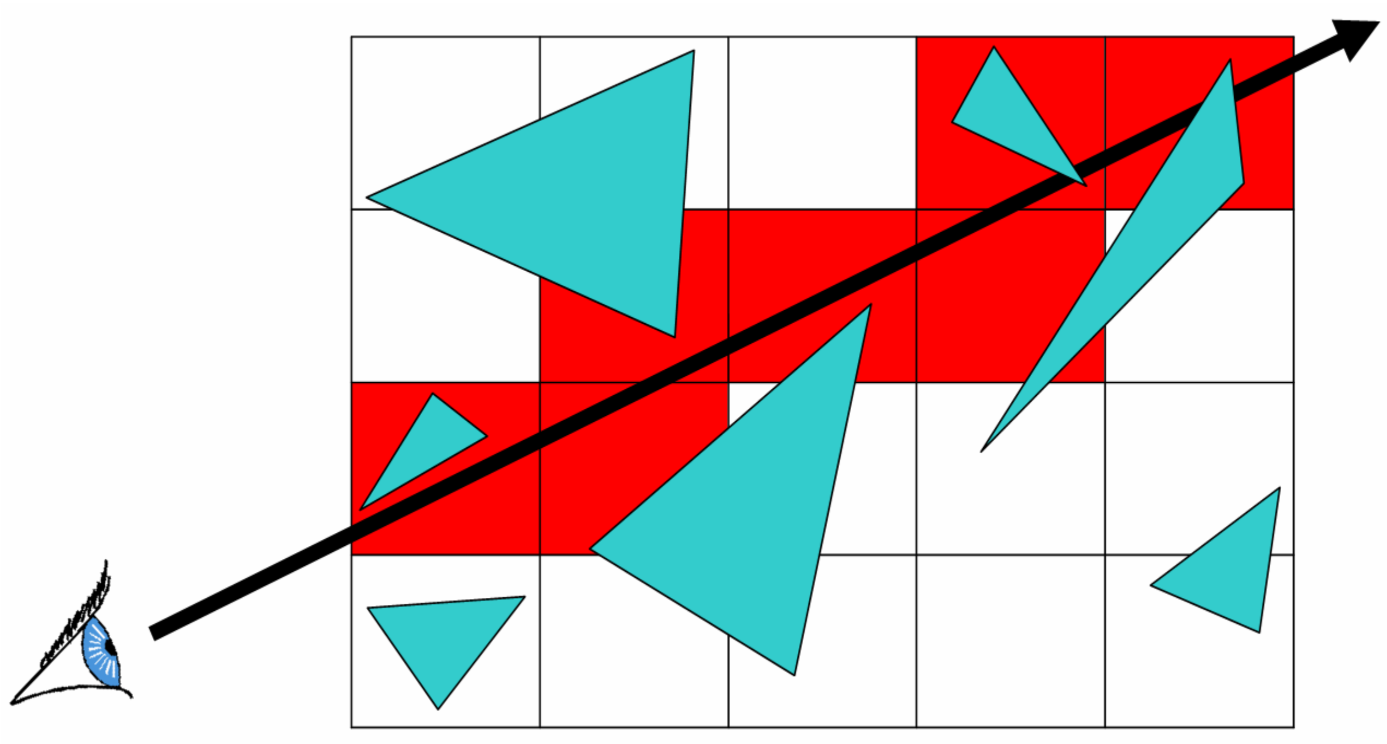
- Observe: y += m at each x step ($m = dy/dx$)



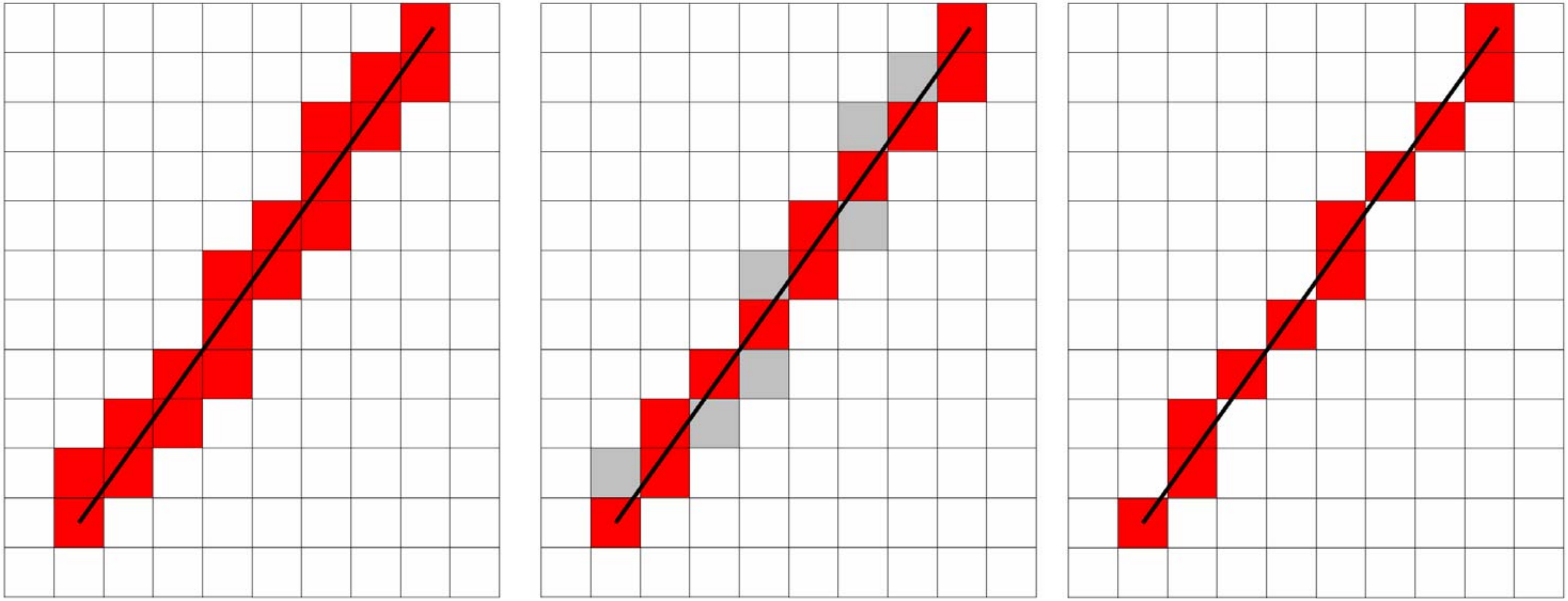


Line Rasterization

- It's like marching a ray through the grid
- Also uses DDA (Digital Difference Analyzer)



Grid Marching vs. Line Rasterization



Ray Acceleration:

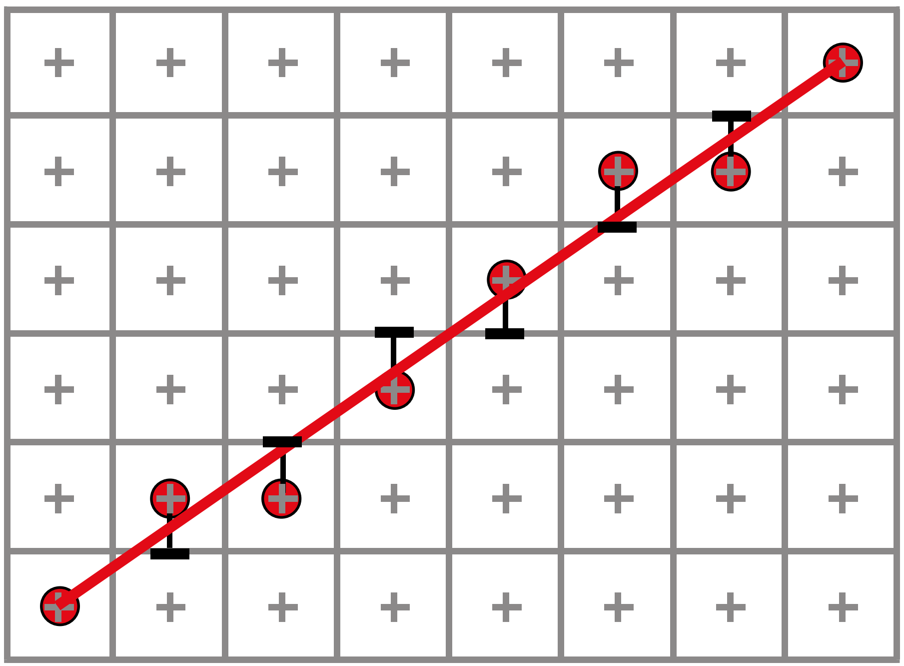
Must examine every cell the line touches

Line Rasterization:

Best discrete approximation of the line

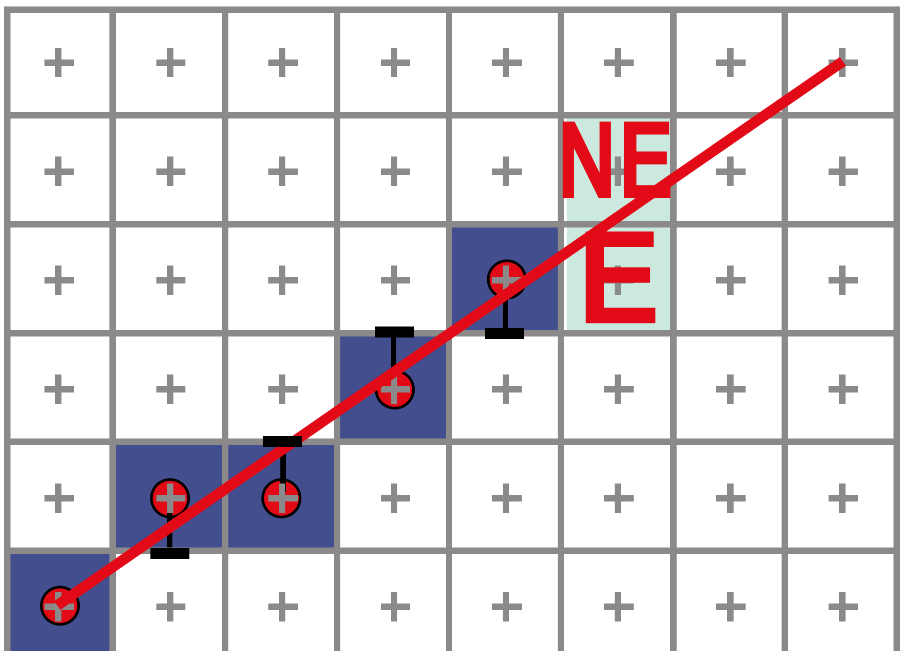
Bresenham's Algorithm (DDA)

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



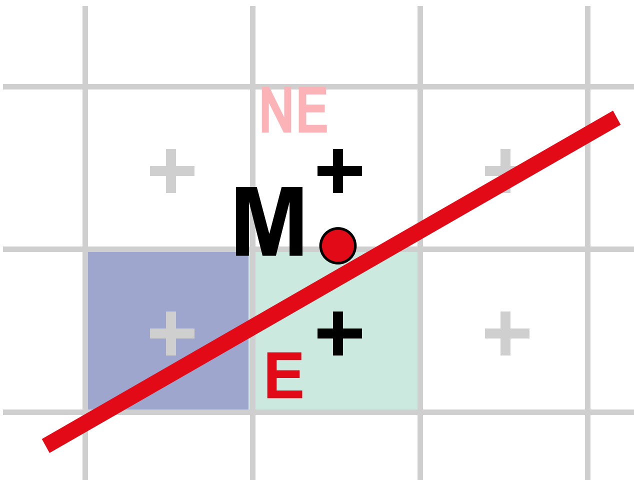
Bresenham's Algorithm (DDA)

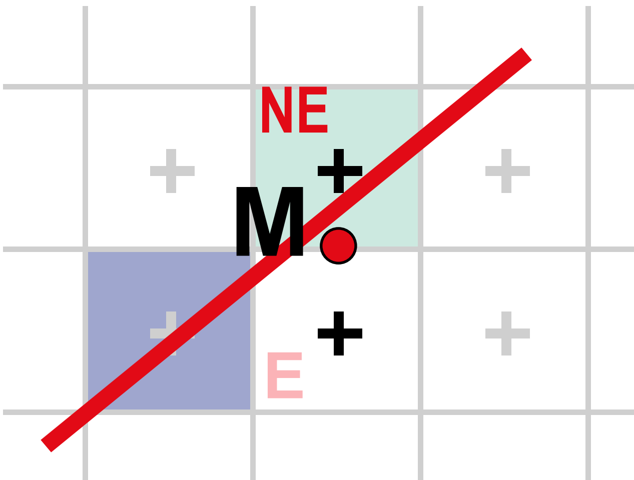
- Observation:
  - If we're at pixel P (x_p, y_p), the next pixel must be either E (x_p+1, y_p) or NE (x_p, y_p+1)
  - Why?



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



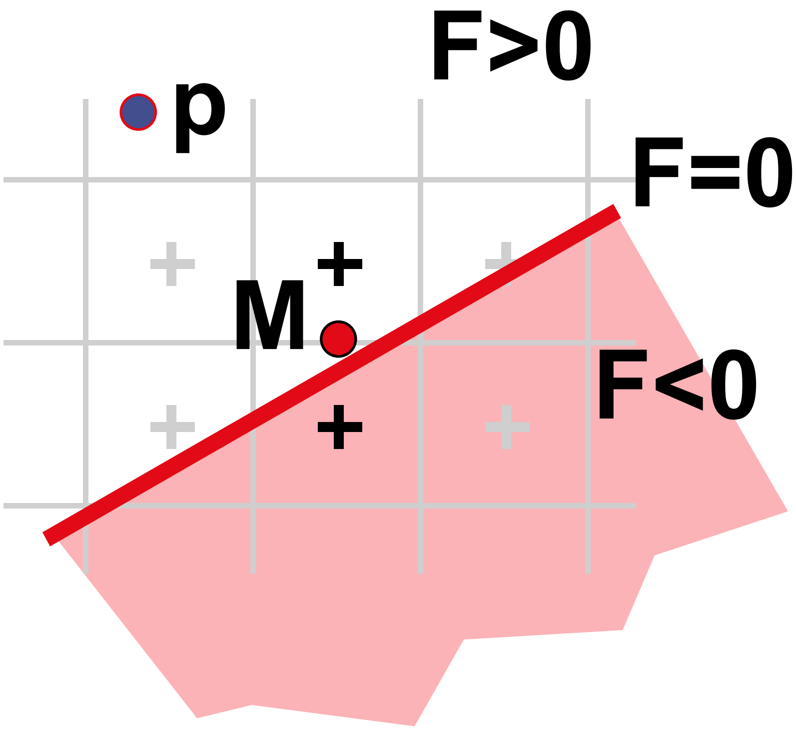


Bresenham Step

- 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

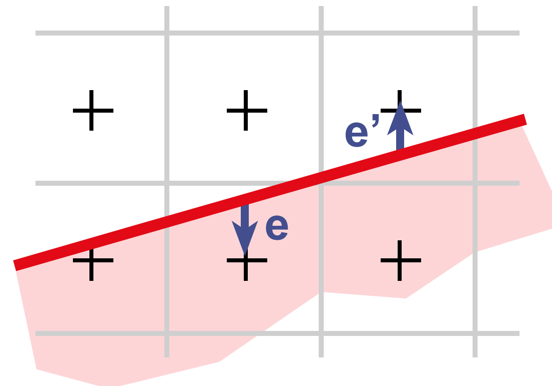


$D(p_x,p_y) =$ vertical distance from point to line

Bresenham's Algorithm (DDA)

- Decision Function:

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



- Initialize:

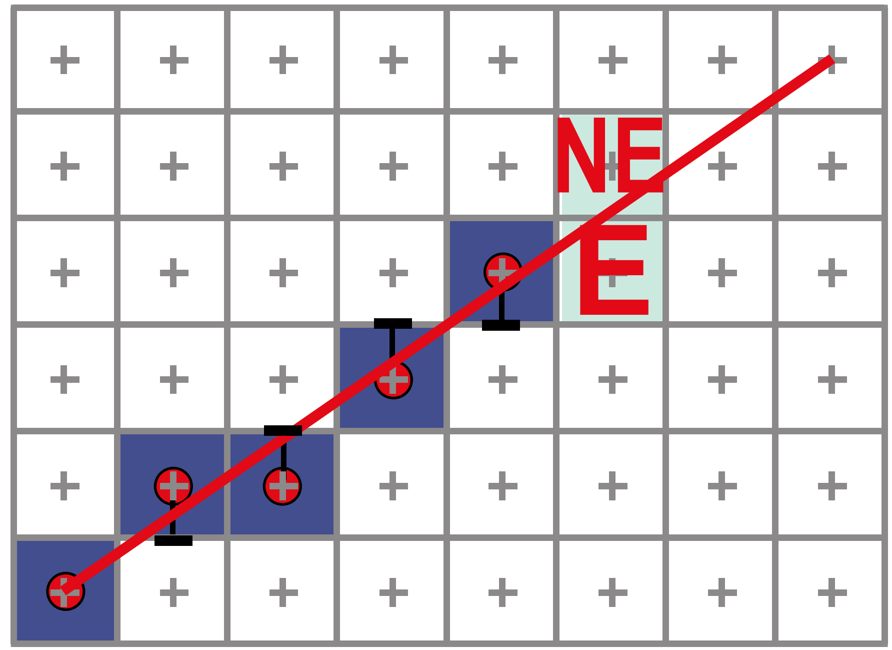
  error term $e = -D(x,y)$

- 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; $x \leq$ 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

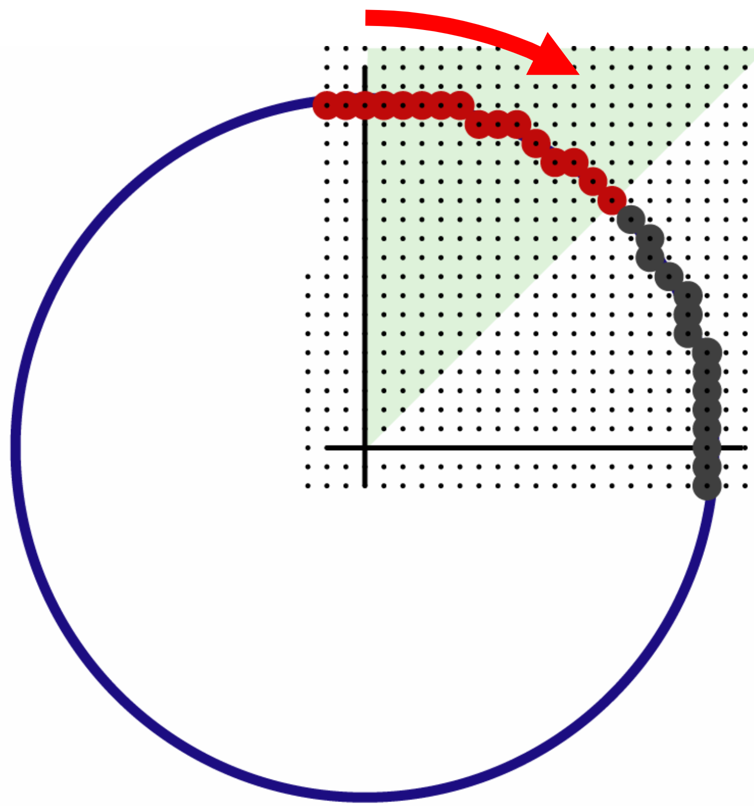
Questions?

Today

- Why Clip?
- Line Clipping
- Overview of Rasterization
- Line Rasterization
  - naive method
  - Bresenham's (DDA)
- Circle Rasterization
- Antialiased Lines

Circle Rasterization

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



Circle Rasterization

- Decision Function:

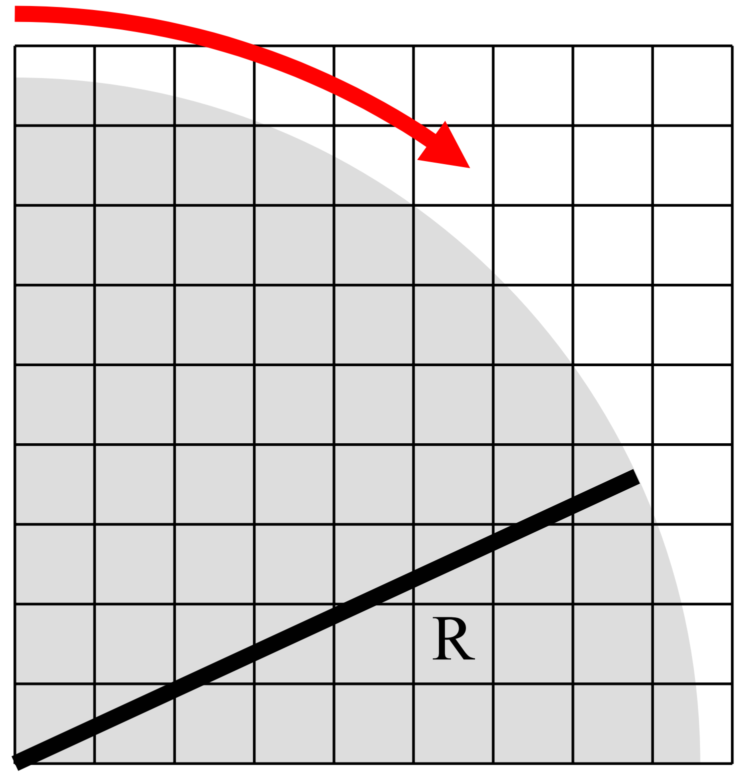
  $D(x, y) = \quad x^2 + y^2 - R^2$

- Initialize:

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

- On each iteration:

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

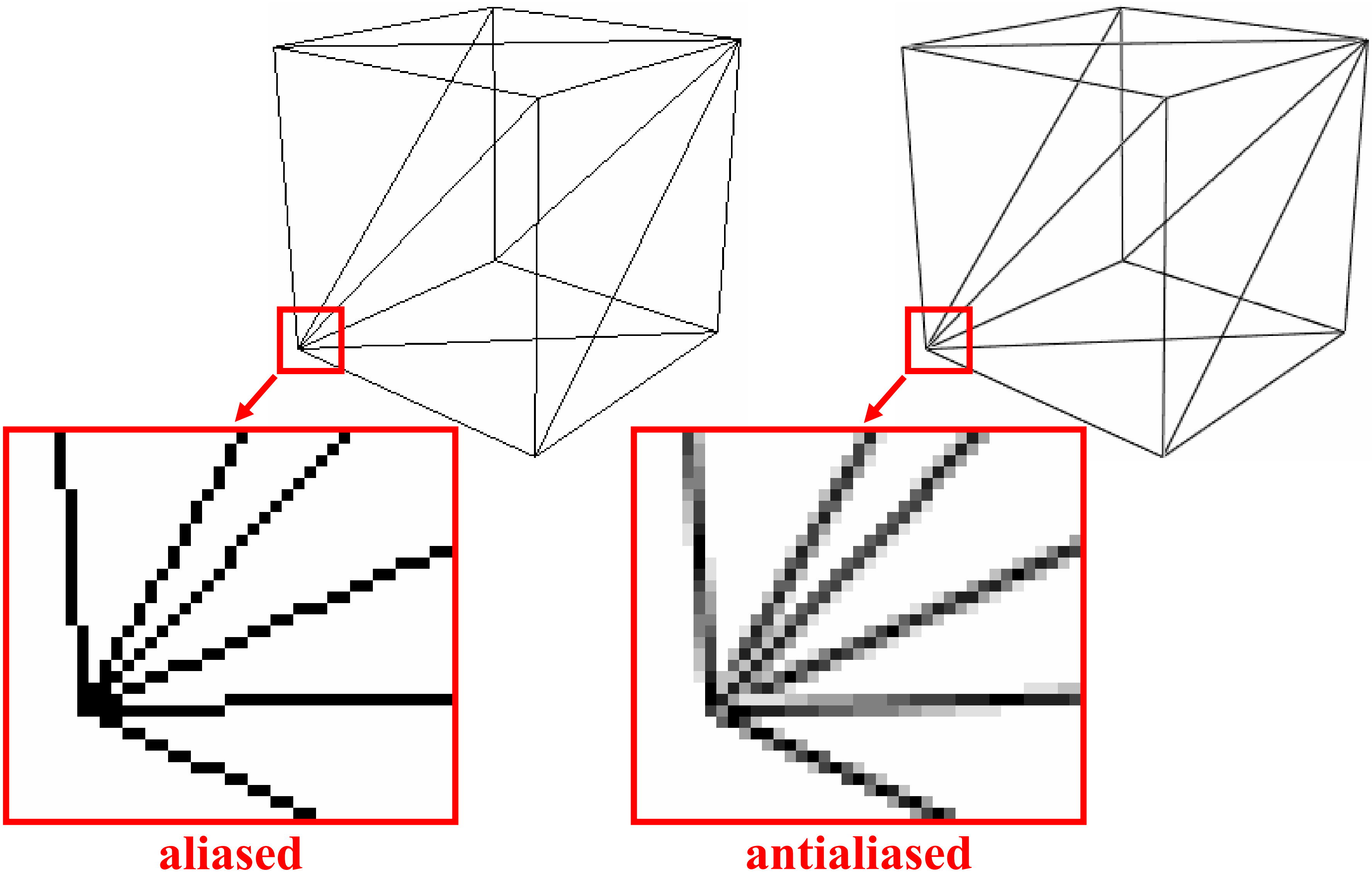


Questions?

Today

- Why Clip?
- Line Clipping
- Overview of Rasterization
- Line Rasterization
  – naive method
  – Bresenham's (DDA)
- Circle Rasterization
- Antialiased Lines

Antialiased Line Rasterization



Next Week:

Polygon Rasterization
& Polygon Clipping