Clipping and other geometric algorithms

MIT EECS 6.837

Frédo Durand
and Barb Cutler

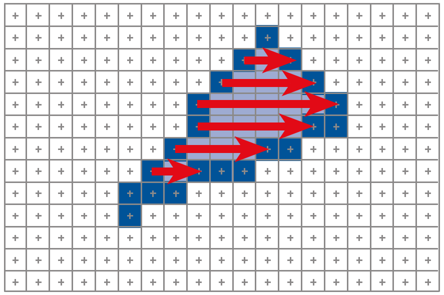
Final projects

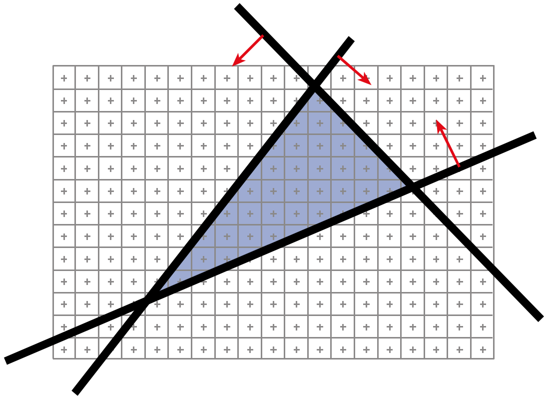
- Rest of semester
  - Weekly meetings with TAs
  - Office hours on appointment
- This week, with TAs
  - Refine timeline
  - Define high-level architecture
- Project should be a whole, but subparts should be identified with regular merging of code

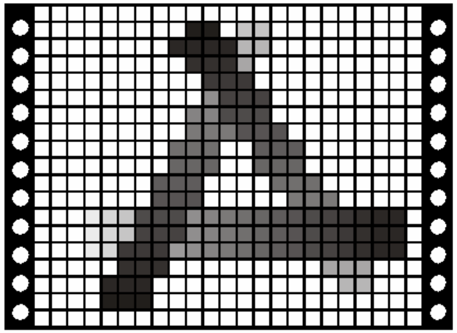
Review of last time?

Last time

- Polygon scan conversion
  - Smart
    - Take advantage of coherence
    - Good for big triangles
  - back to brute force
    - Incremental edge equation
    - Good for small triangles
    - Simpler clipping
- Visibility
  - Painer: complex ordering
  - Z buffer: simple, memory cost
    - Hyperbolic z interpolation

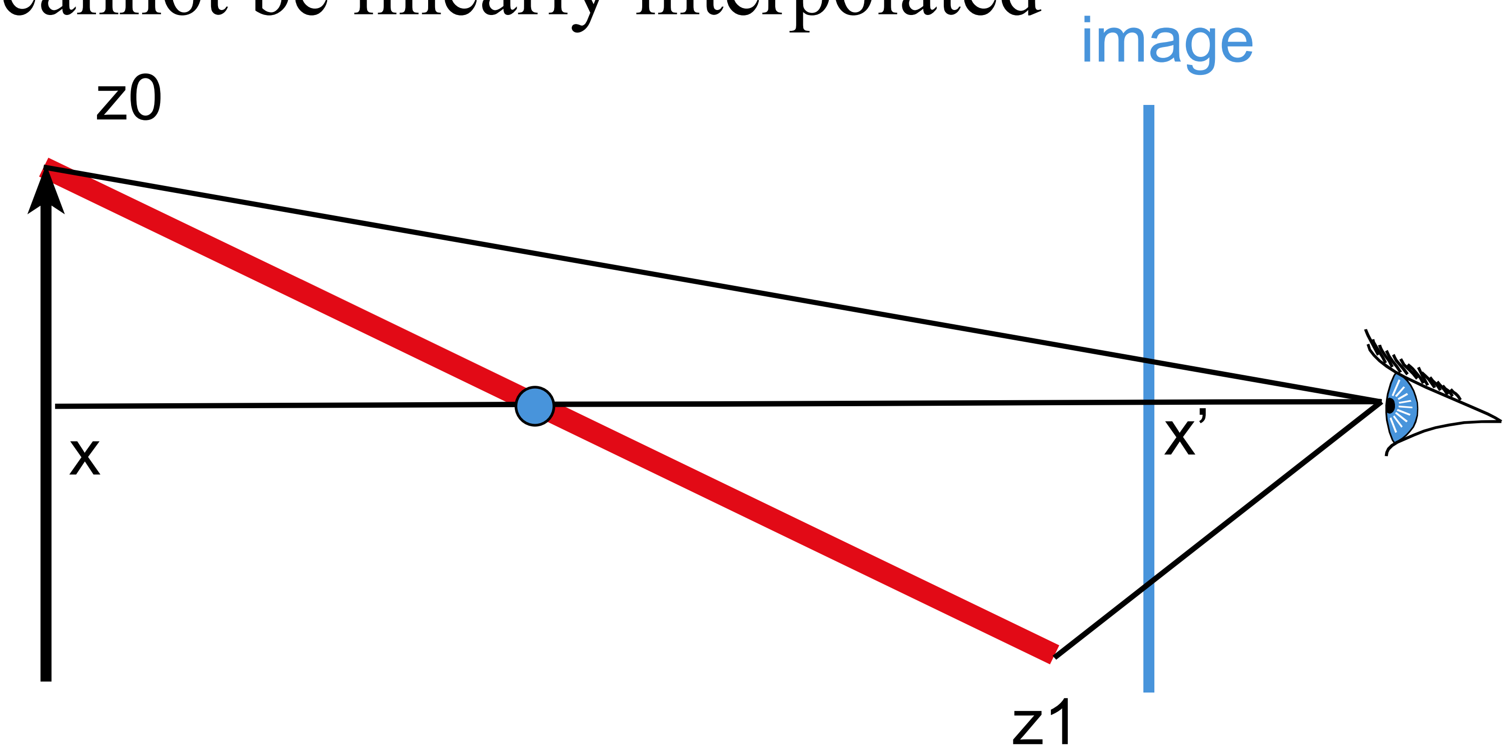






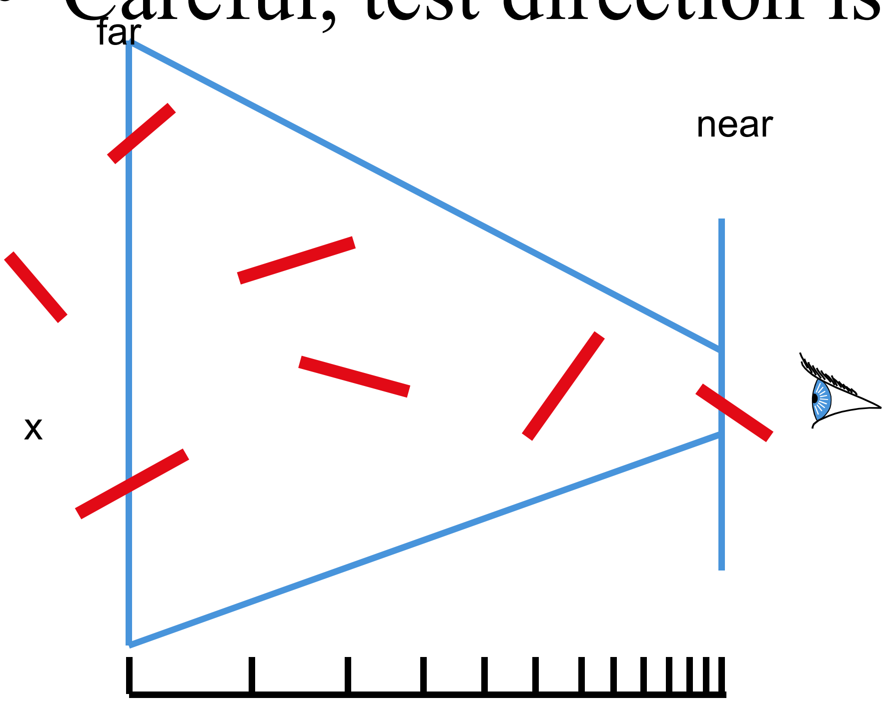
Z interpolation

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



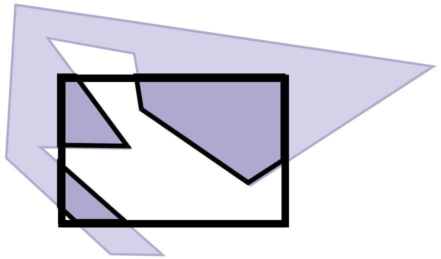
Integer z-buffer

- 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

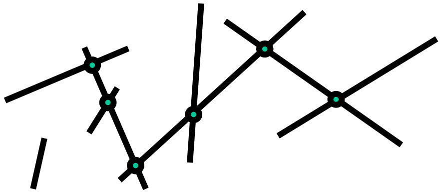


Plan

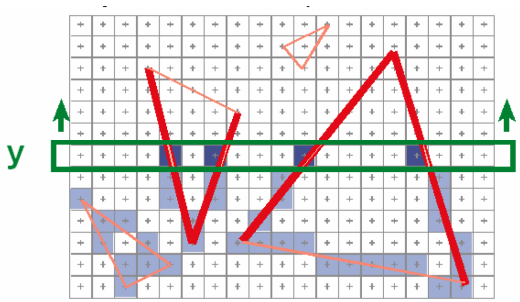
- Review of rendering pipeline
- 2D polygon clipping



- Segment intersection



- Scanline rendering overview



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

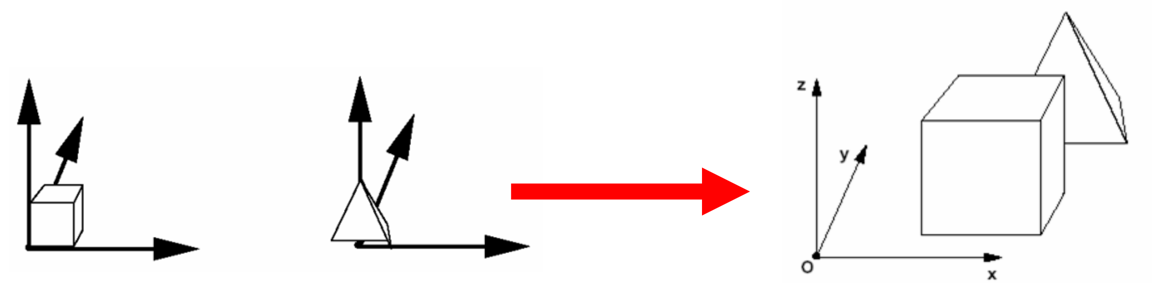
Raster Viewport:
Pixel grid onto which image plane is mapped

Output:

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

Modeling Transformations

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



Object space World space

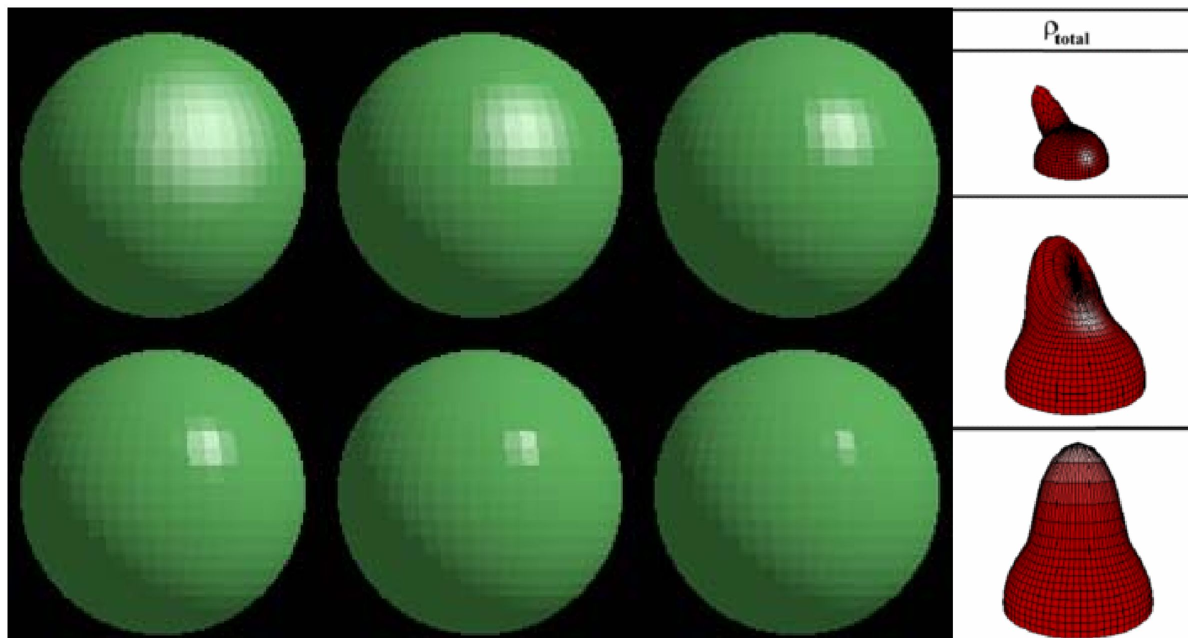
$$\begin{pmatrix} x' \\ y' \\ z' \\ \color{red}{1} \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ \color{red}{0} & \color{red}{0} & \color{red}{0} & \color{red}{1} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ \color{red}{1} \end{pmatrix}$$

Illumination (Shading) (Lighting)

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

- Vertices lit (shaded) according to material properties, surface properties (normal) and light
- Local lighting model (Diffuse, Ambient, Phong, etc.)

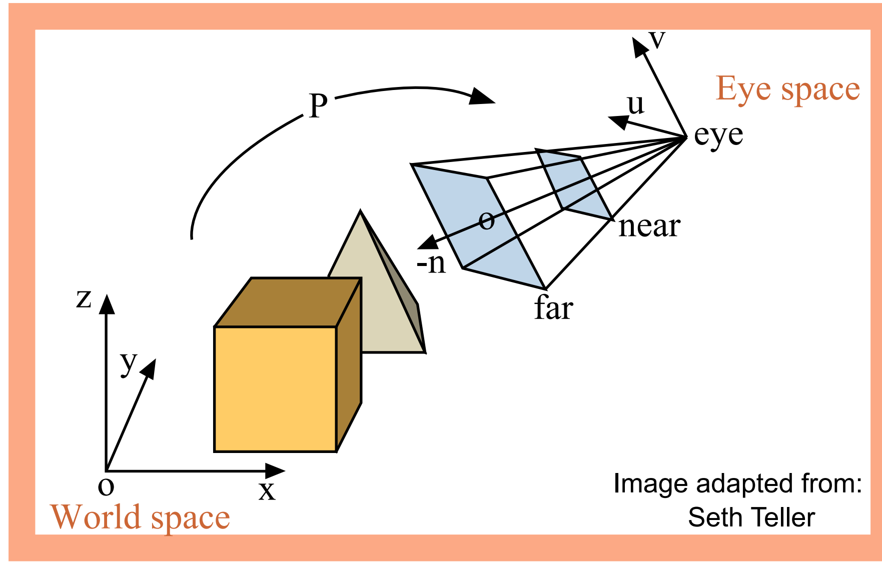
$$L(\omega_r) = k_a + \left(k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right) \frac{\Phi_s}{4\pi\, d^2}$$



Viewing Transformation

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

- Viewing position is transformed to origin & direction is oriented along some axis (usually *z*)



Yet another 4x4 matrix

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Clipping

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

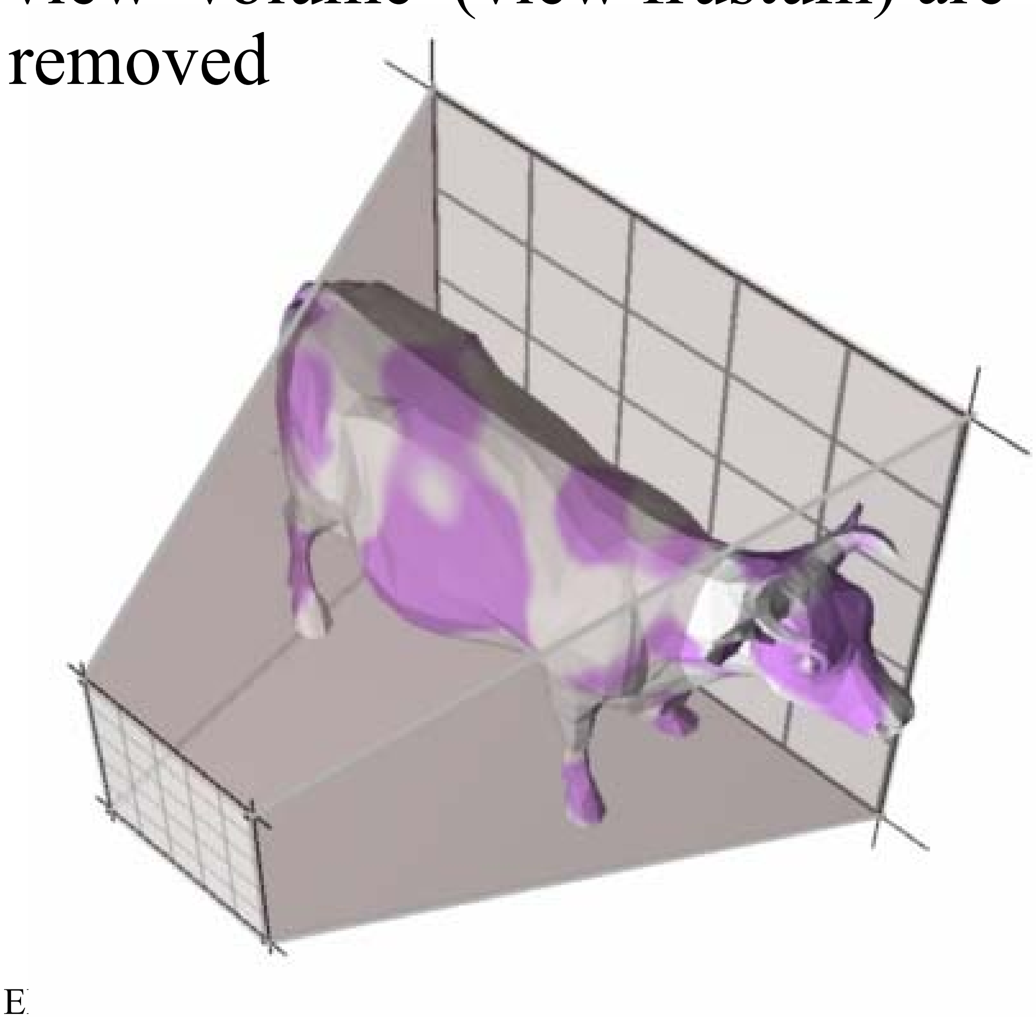
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Portions of the object outside the view volume (view frustum) are removed



Clipping – modern hardware

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

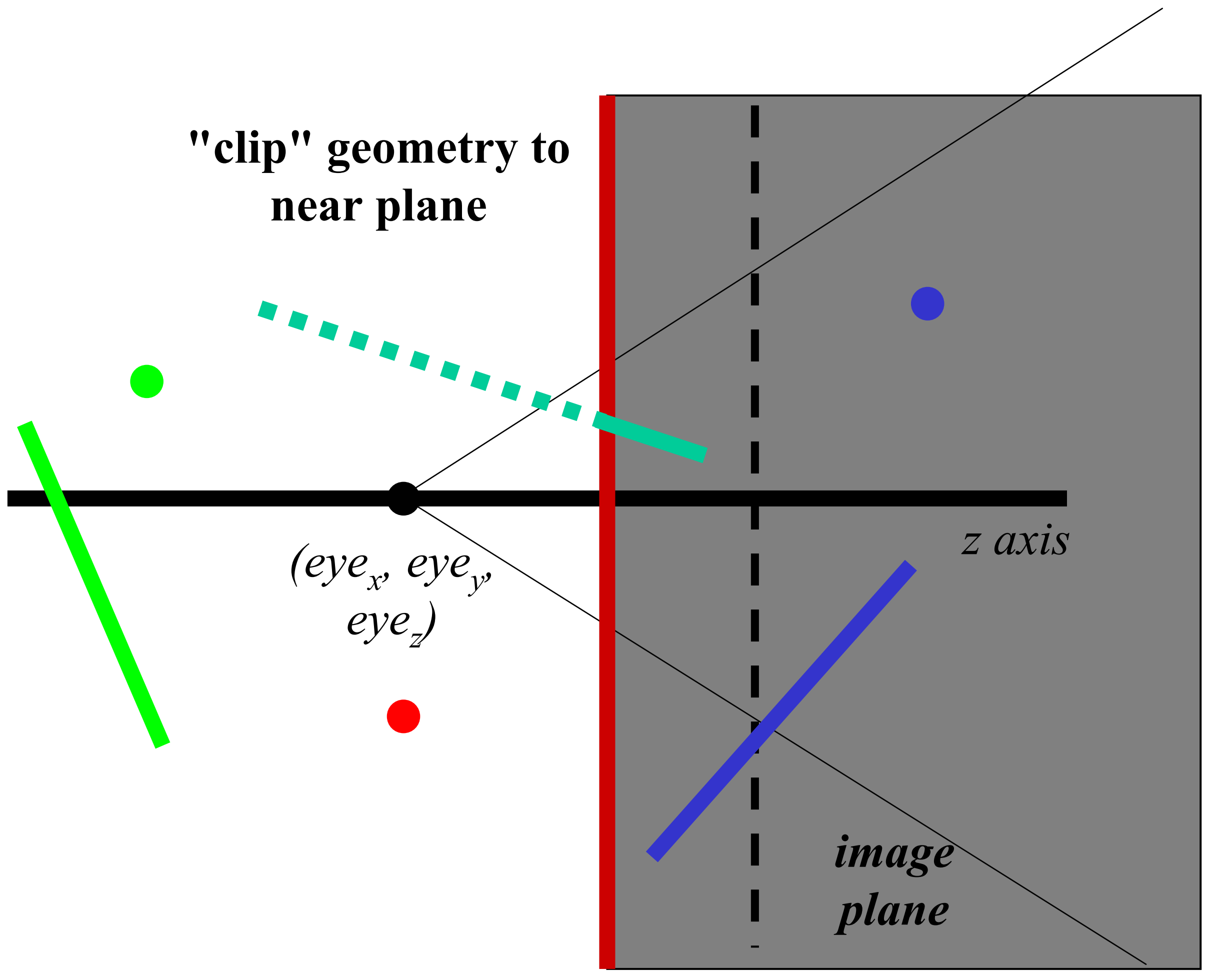
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Only to the near plane



Projection

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

- Projective transform

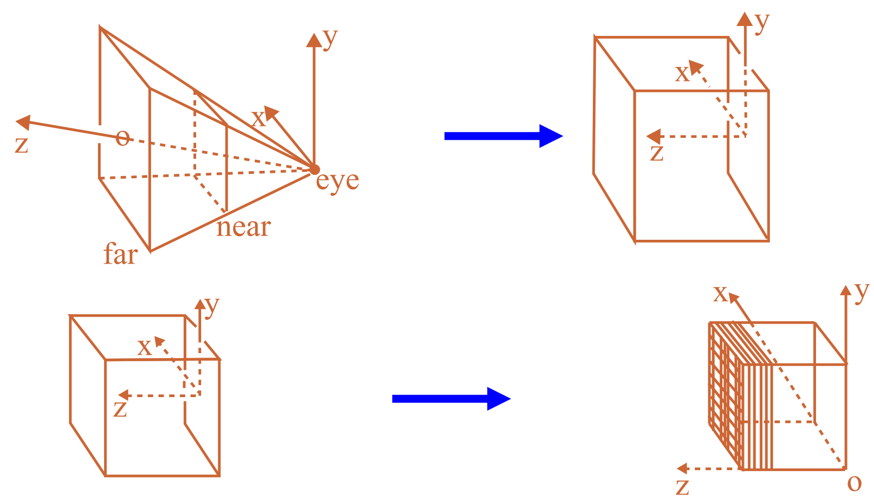
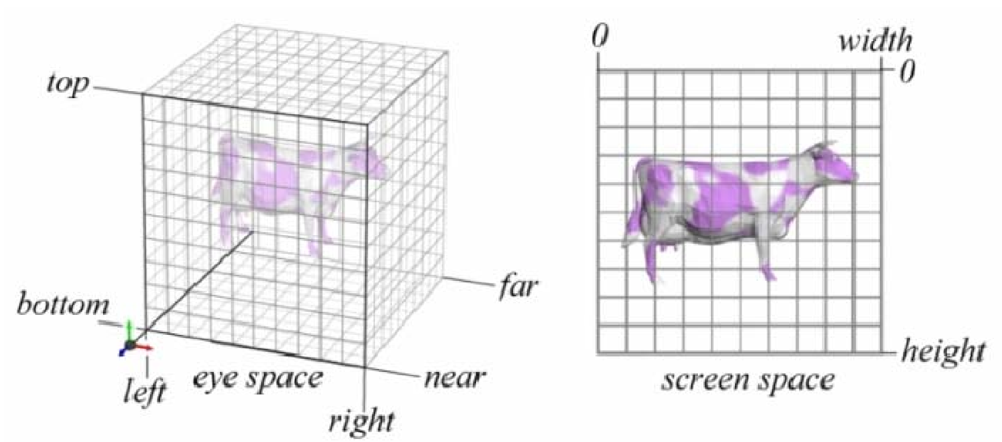
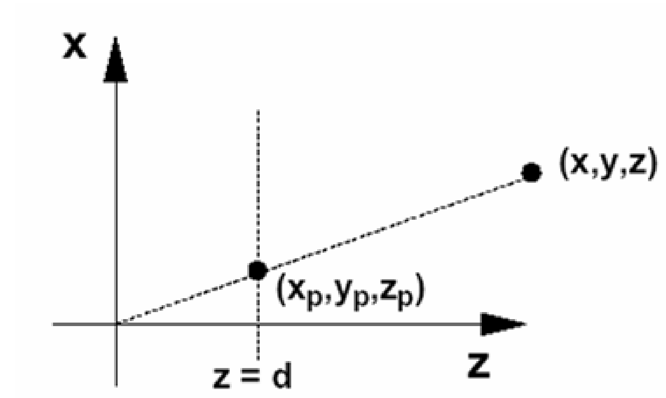


Image adapted from:
Seth Teller



Perspective Projection

- 2 conceptual steps:
  - 4x4 matrix
  - Homogenize
    - In fact not always needed
    - Mordern graphics hardware performs most operations in 2D homogeneous coordinates

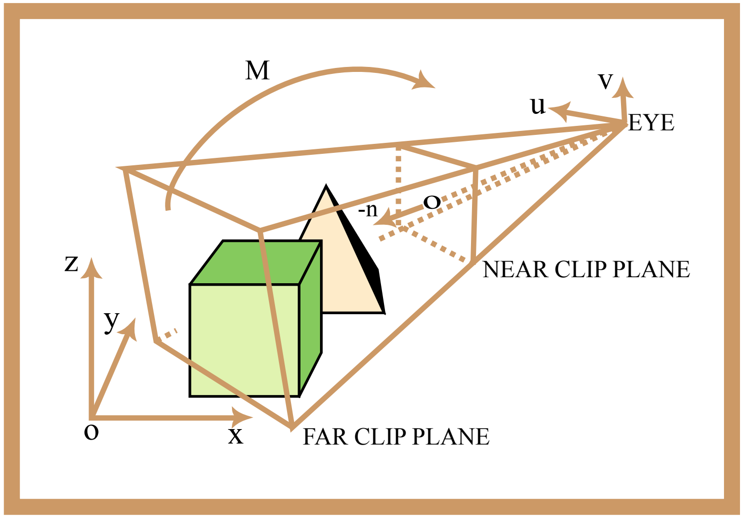


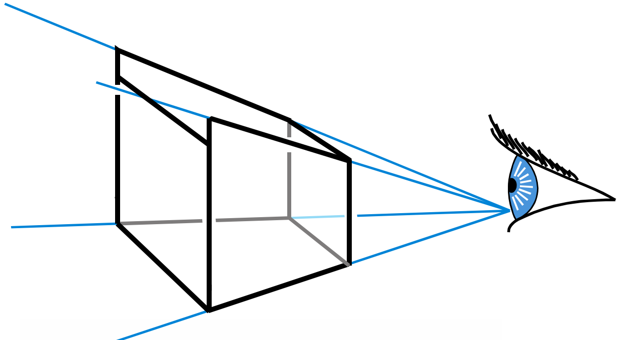
homogenize

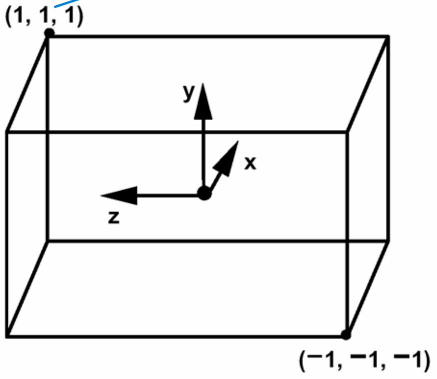
$$\begin{pmatrix} x * d / z \\ y * d / z \\ d/z \\ 1 \end{pmatrix} \xleftarrow{homogenize} \begin{pmatrix} x \\ y \\ 1 \\ z / d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

When to clip?

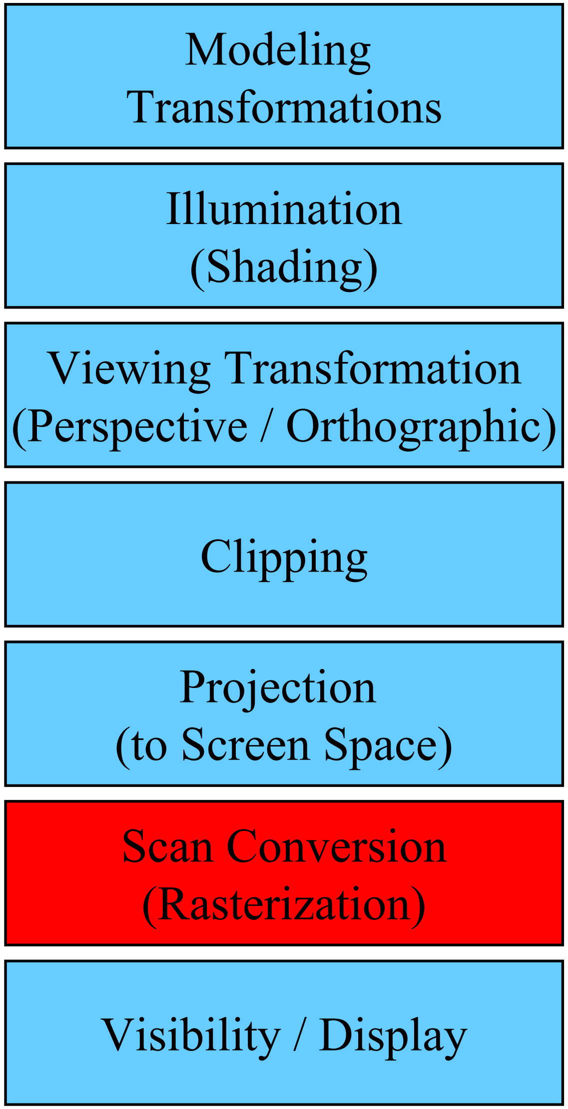
- Before perspective transform in 3D space
  - Use the equation of 6 planes
  - Natural, not too degenerate
- In homogeneous coordinates after perspective transform (Clip space)
  - Before perspective divide (4D space, weird *w* values)
  - Canonical,independent of camera
  - The simplest to implement in fact
- In the transformed 3D screen space after perspective division
  - Problem: objects in the plane of the camera



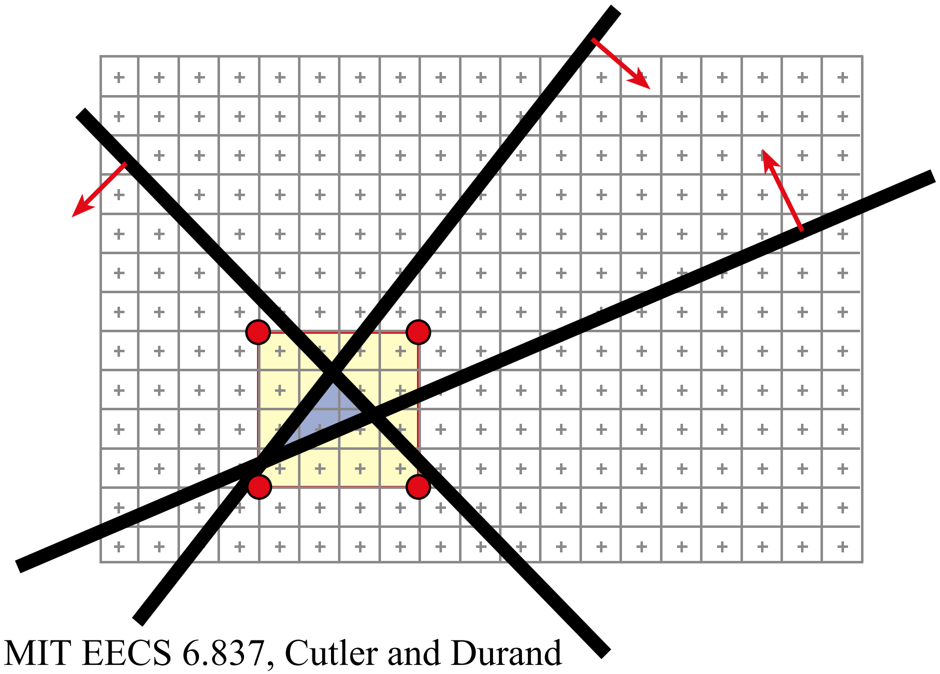




Scan Conversion (Rasterization)



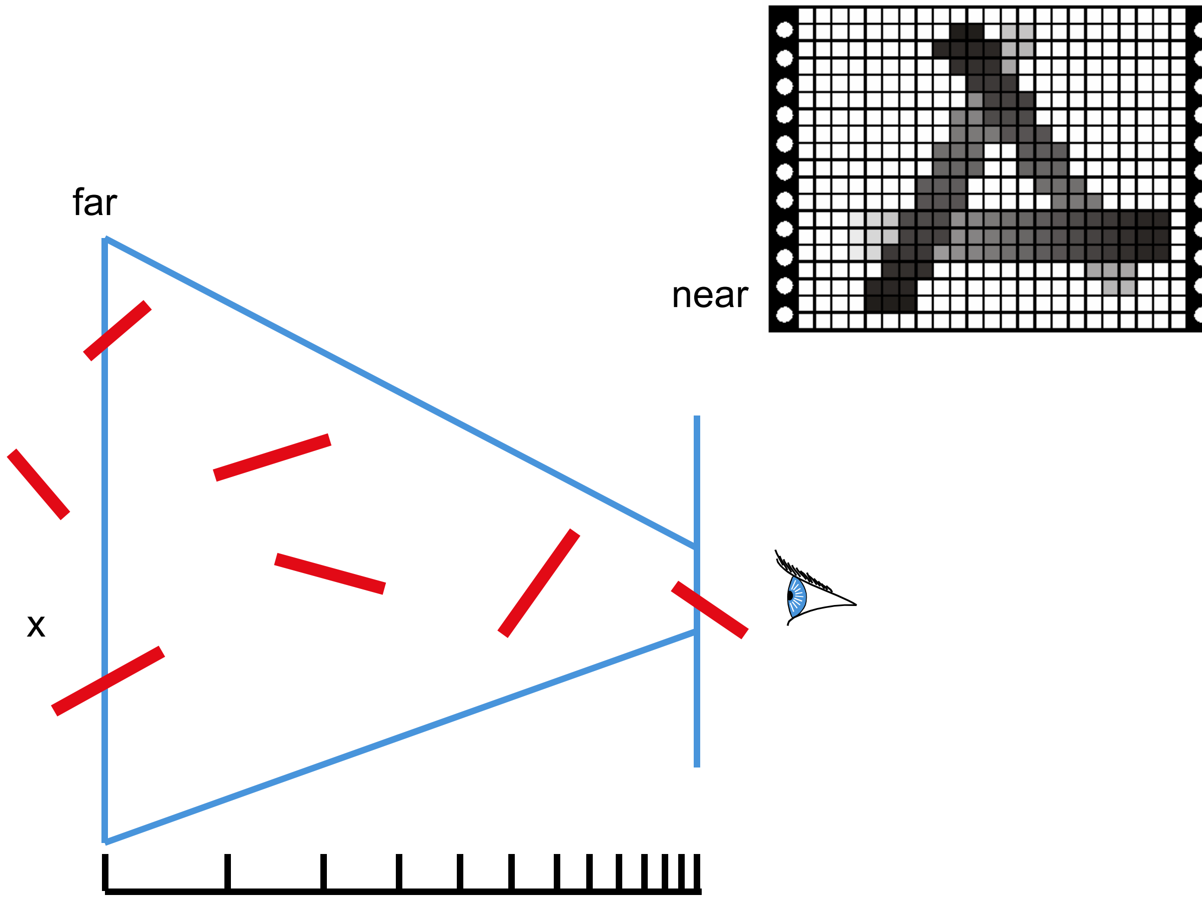
- Incremental edge equations
- Interpolate values as we go (color, depth, etc.)
- Screen-space bbox clipping



Visibility / Display

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

- Each pixel remembers the closest object (depth buffer)



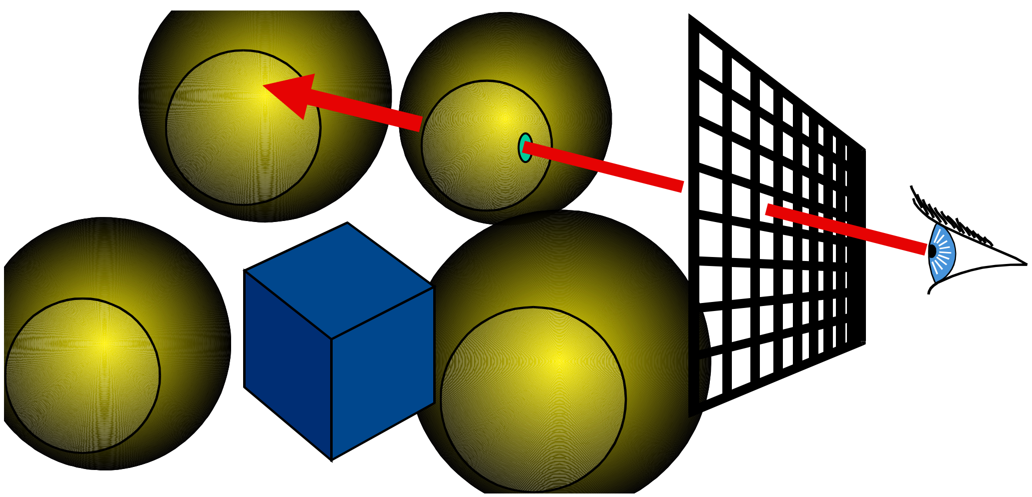
Rendering Pipeline vs. ray casting

Ray Casting

```
For each pixel
    For each object
```

Send pixels to the scene

Discretize first

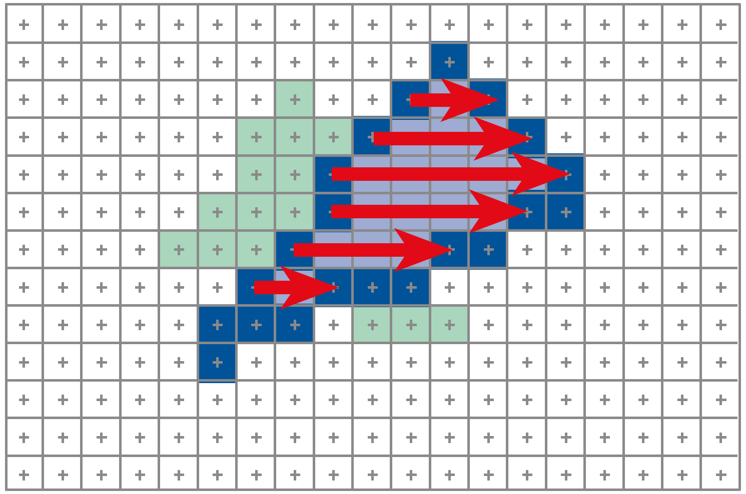


Rendering Pipeline

```
For each triangle
    For each pixel
```

Project scene to the pixels

Discretize last

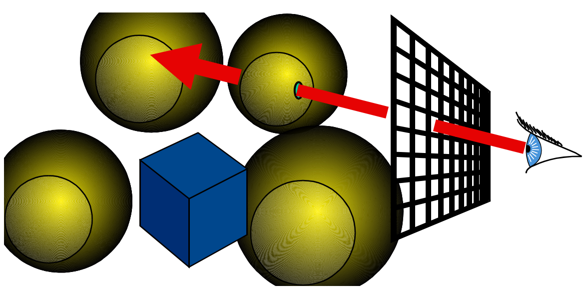


Rendering Pipeline vs. ray casting

Ray Casting

```
For each pixel
    For each object
```

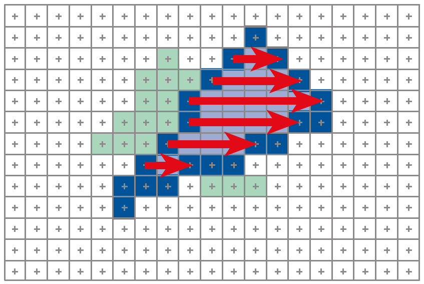
- Depth complexity: no calculation for hidden part
- Whole scene must be in memory
- Very atomic computation
- More general, more flexible
  - Primitive, lighting effects, adaptive antialiasing



Rendering Pipeline

```
For each triangle
    For each pixel
```

- Coherence: geometric transforms for vertices only
- Arithmetic intensity: the amount of computation increases in the depth of the pipeline
  - Good bandwidth/computation ratio
- Harder to get global illumination (shadows, interreflection, etc.)

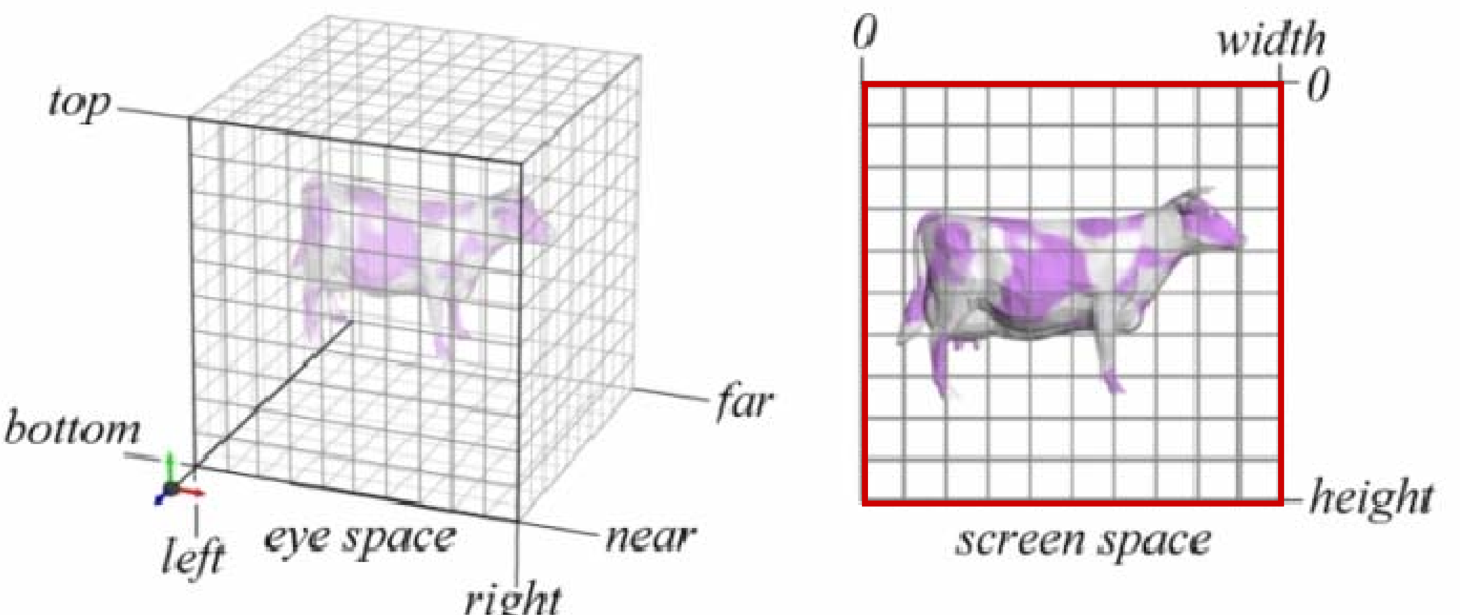


What they use

- Games: pipeline
- Flight simulation: pipeline
- Movies: Both pipeline and ray tracing
- CAD-CAM & design: pipeline during design, anything for final image
- Architecture: ray-tracing, pipeline, but do complex lighting simulation (cf. later lectures)
- Virtual reality: pipeline
- Visualization: mostly pipeline, ray-tracing for high-quality eye candy, interactive ray-tracing is starting

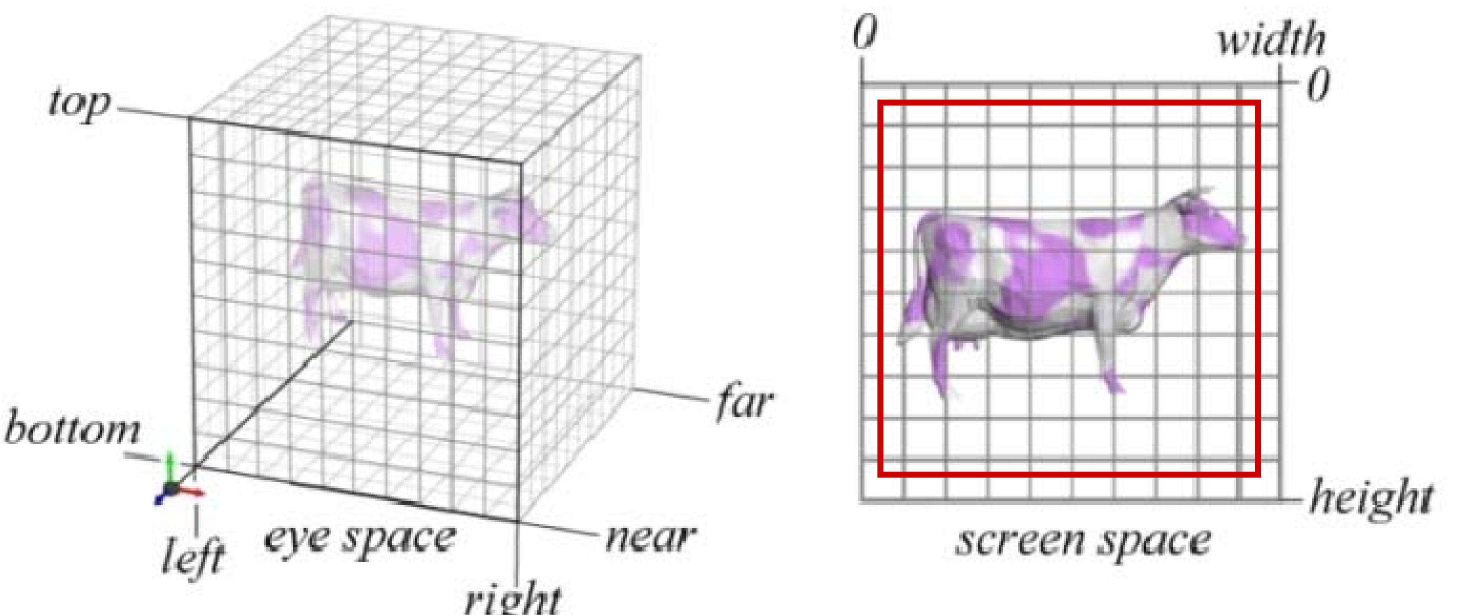
The infamous half pixel

- 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?



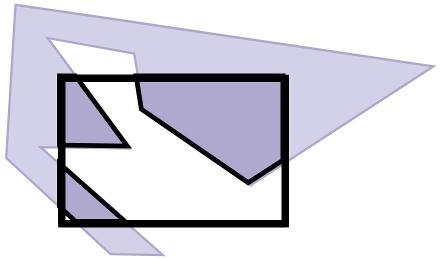
The infamous half pixel

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

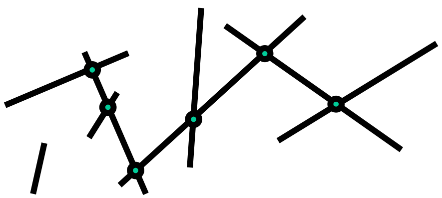


Plan

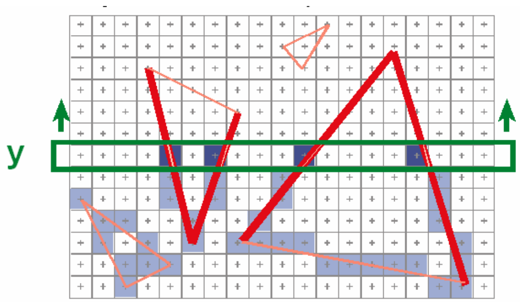
- Review of rendering pipeline
- 2D polygon clipping



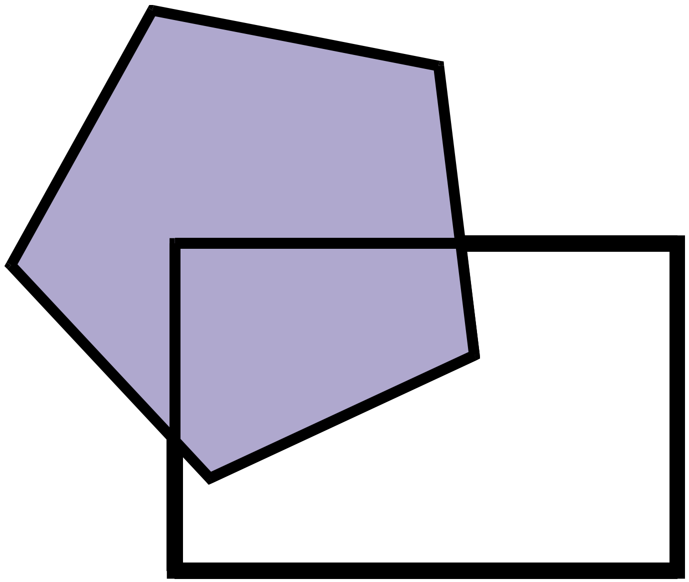
- Segment intersection



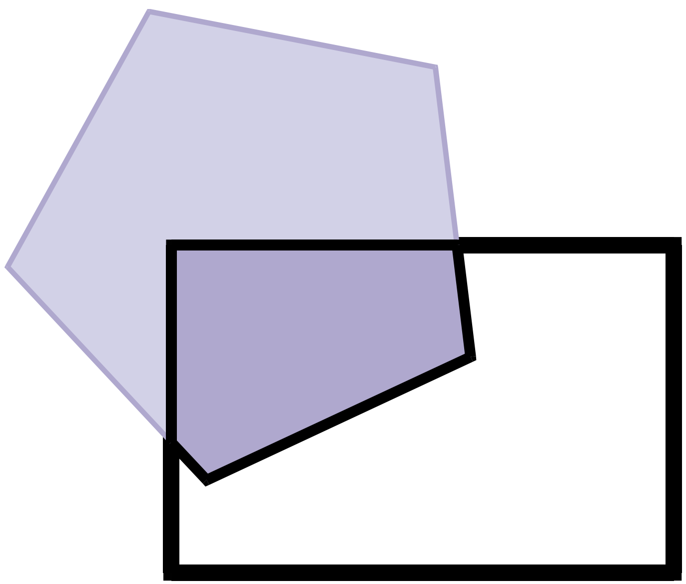
- Scanline rendering overview



Polygon clipping

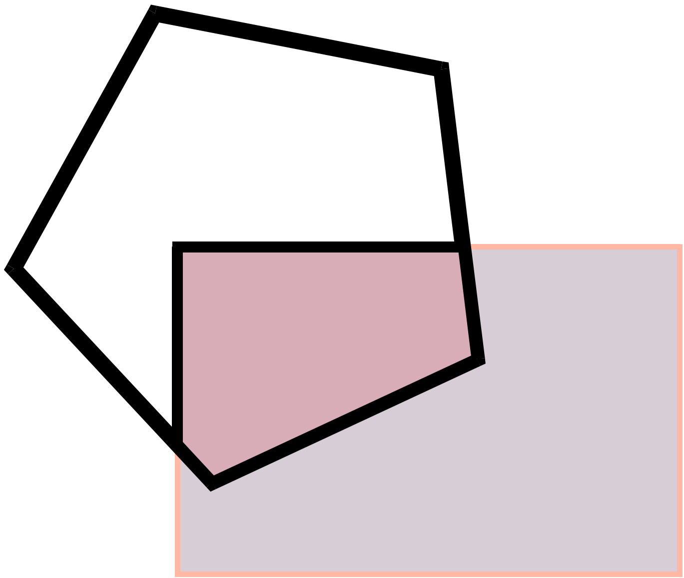


Polygon clipping



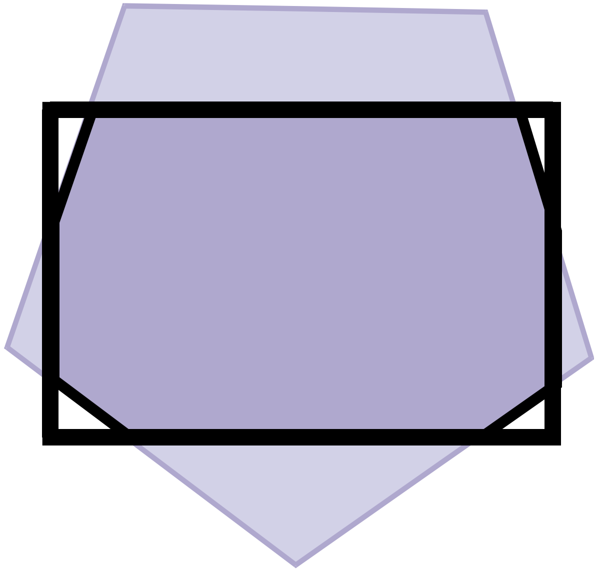
Polygon clipping

- Clipping is symmetric



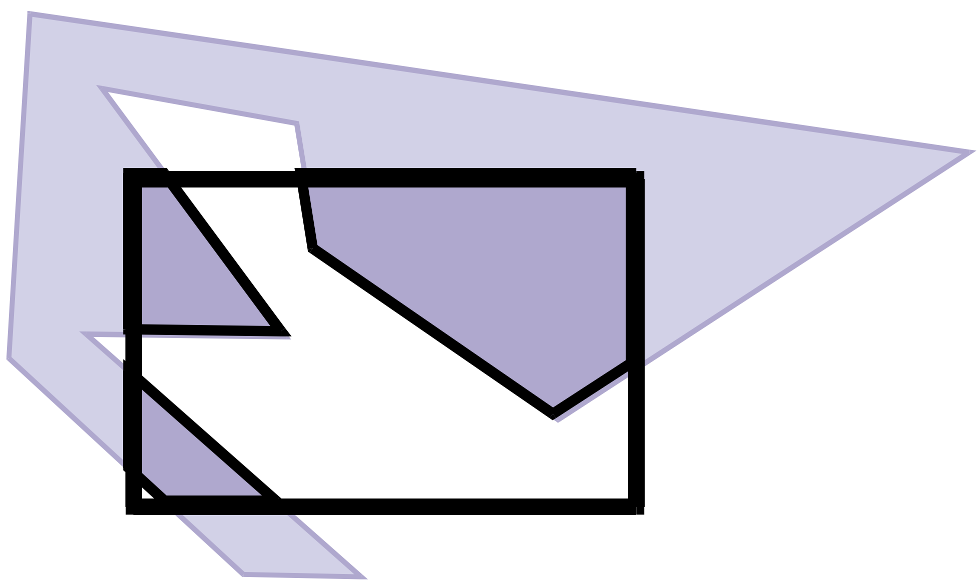
Polygon clipping is complex

- Even when the polygons are convex



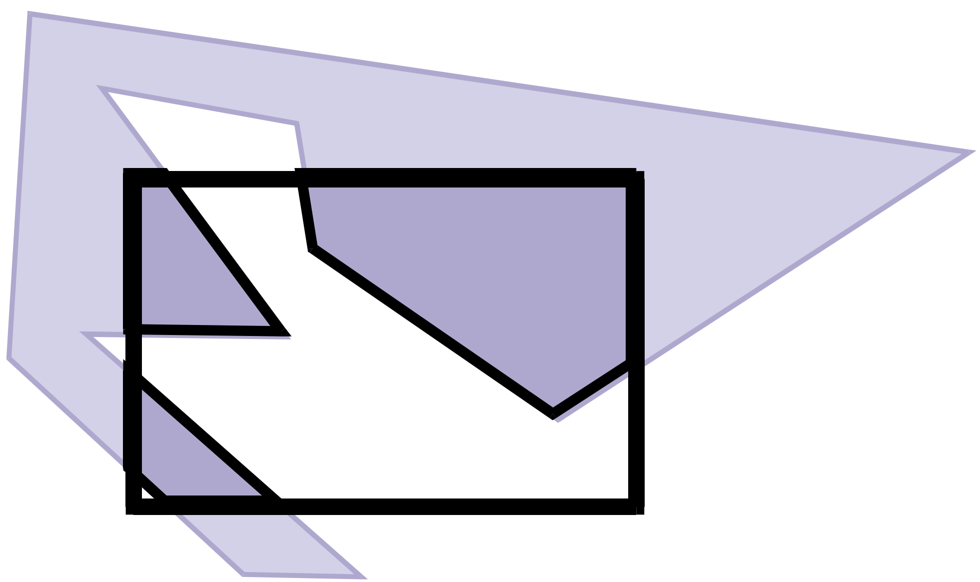
Polygon clipping is nasty

- When the polygons are concave



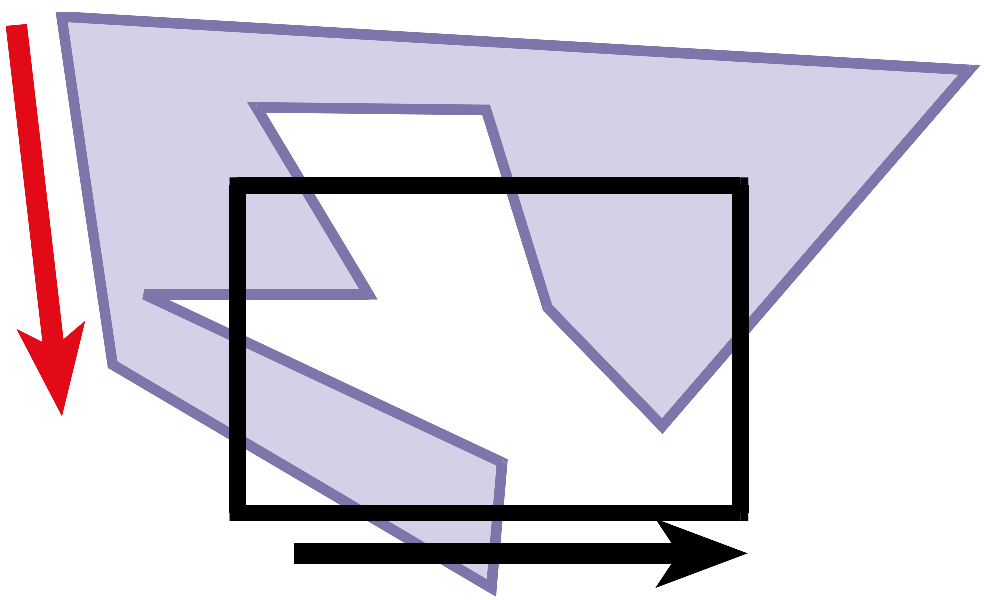
Naïve polygon clipping?

- N*m intersections
- Then must link all segment
- Not efficient and not even easy



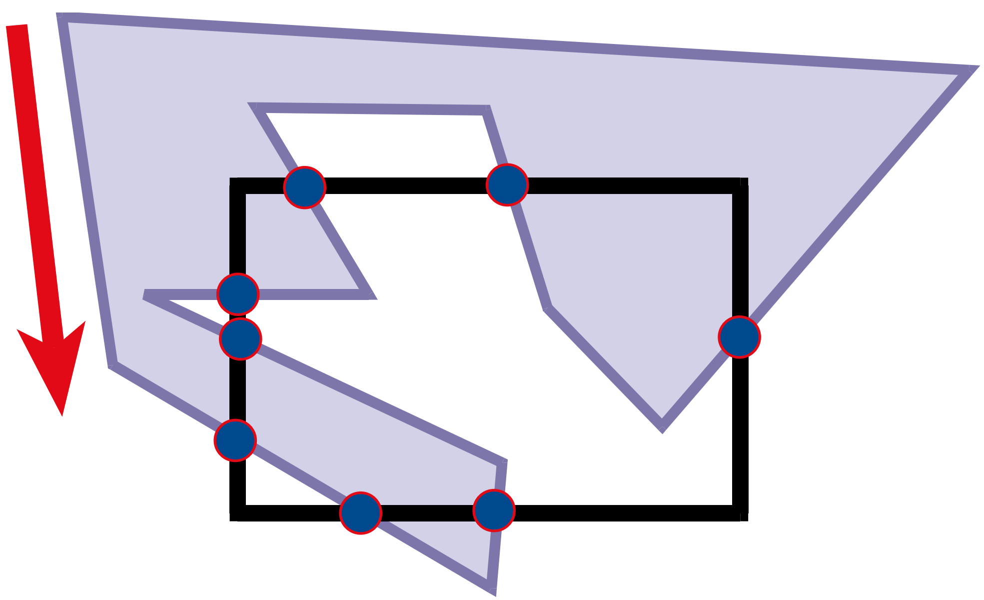
Weiler-Atherton Clipping

- Strategy: "Walk" polygon/window boundary
- Polygons are oriented (CCW)



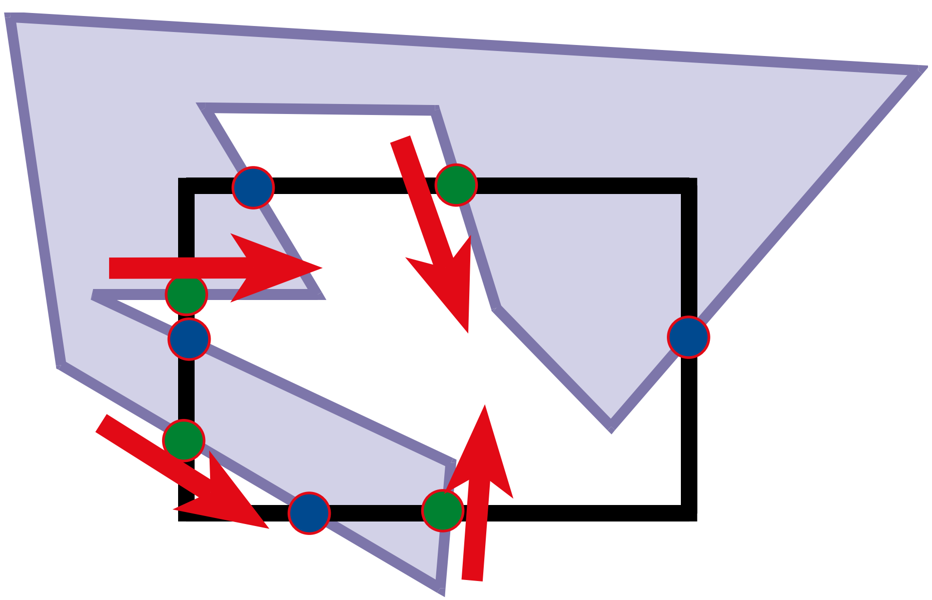
Weiler-Atherton Clipping

- Compute intersection points



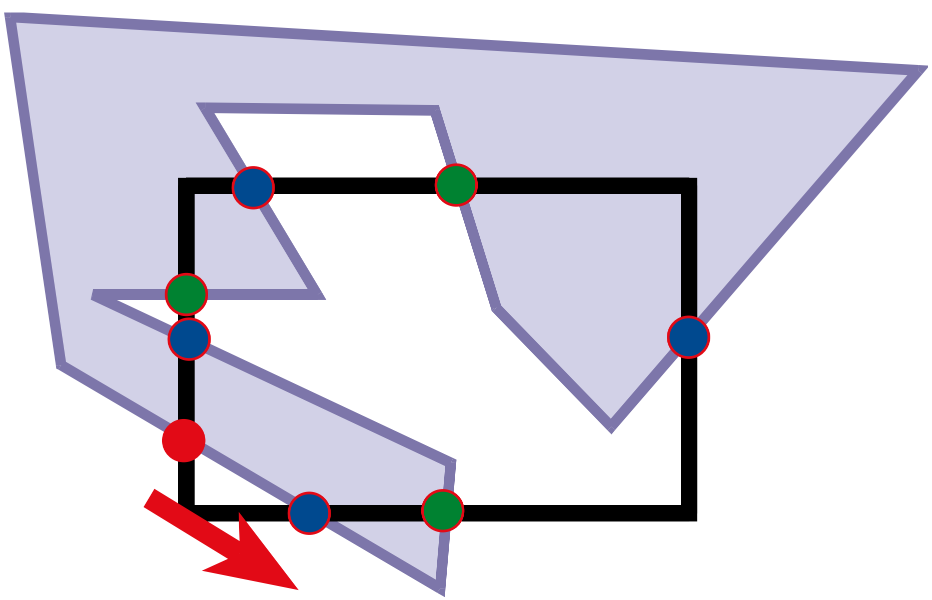
Weiler-Atherton Clipping

- Compute intersection points
- Mark points where polygons enters clipping window (green here)



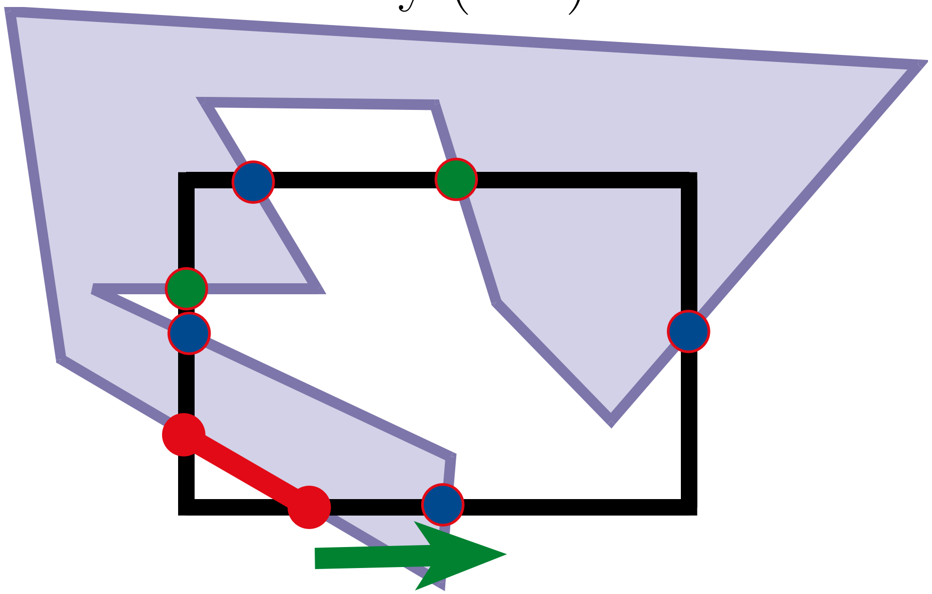
Clipping

While there is still an unprocessed entering intersection
Walk” polygon/window boundary



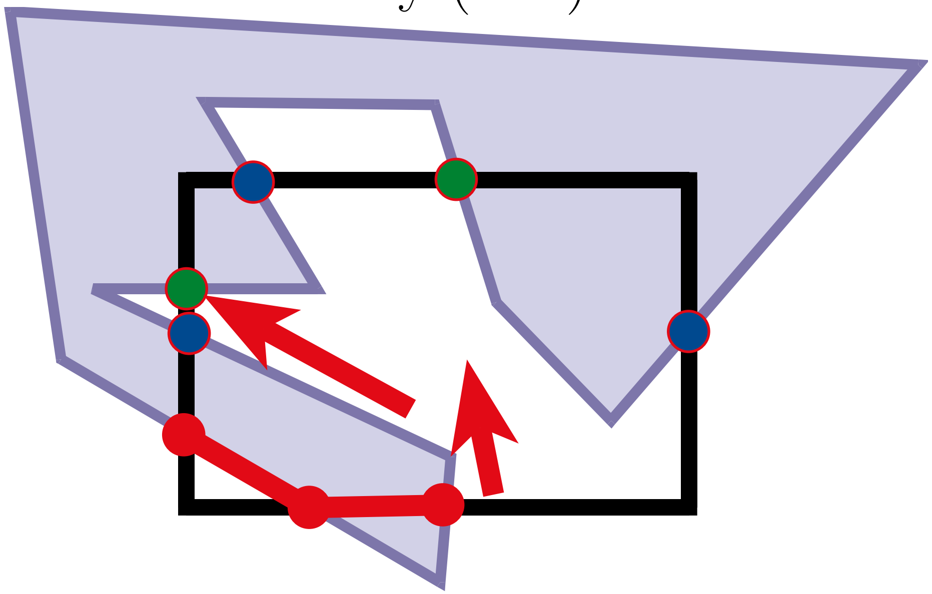
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



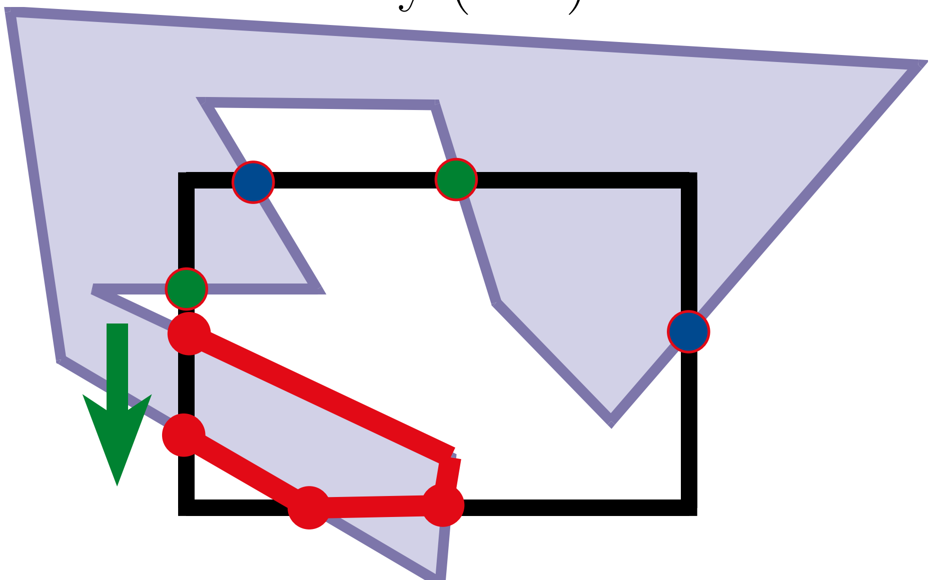
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



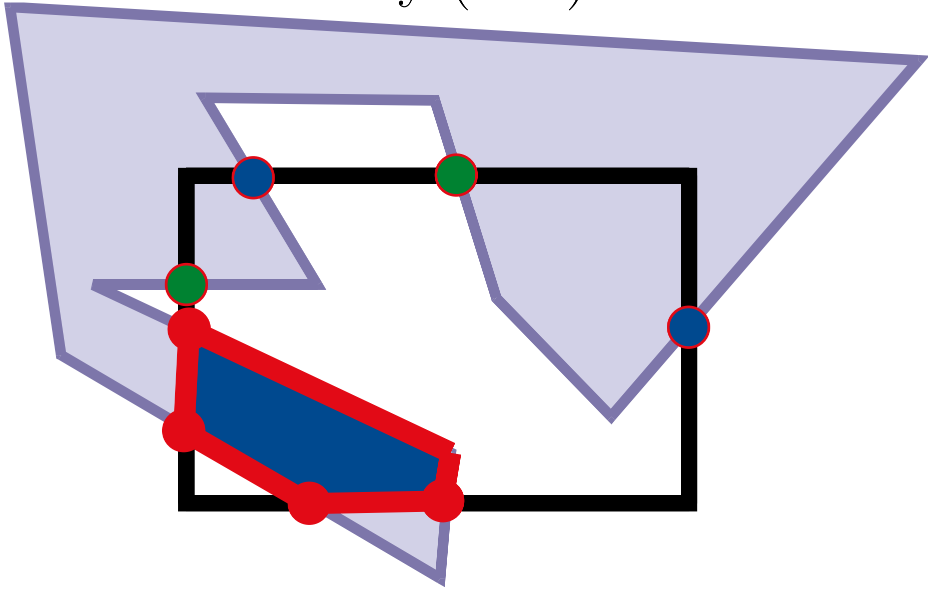
Walking rules

- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



Walking rules

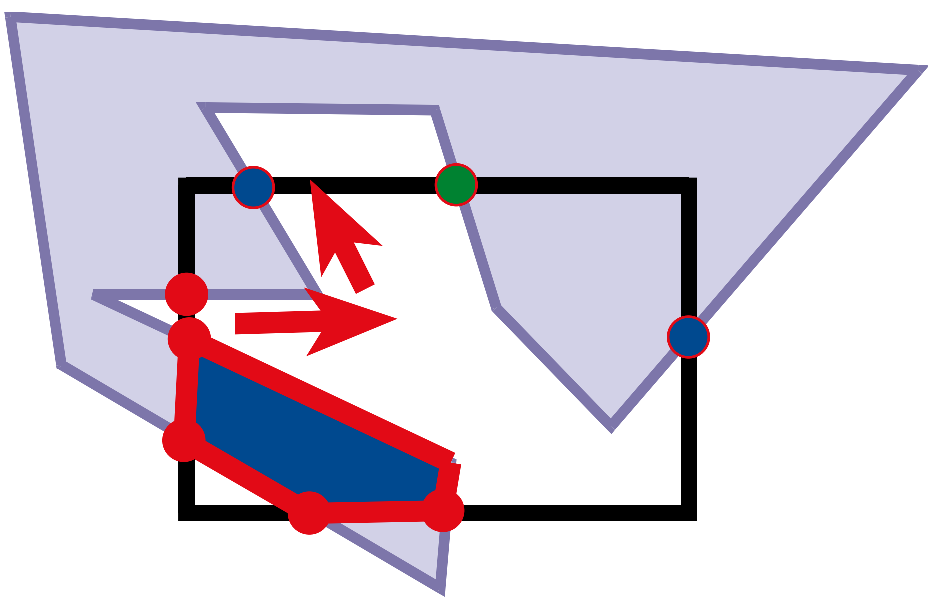
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



Walking rules

While there is still an unprocessed entering intersection

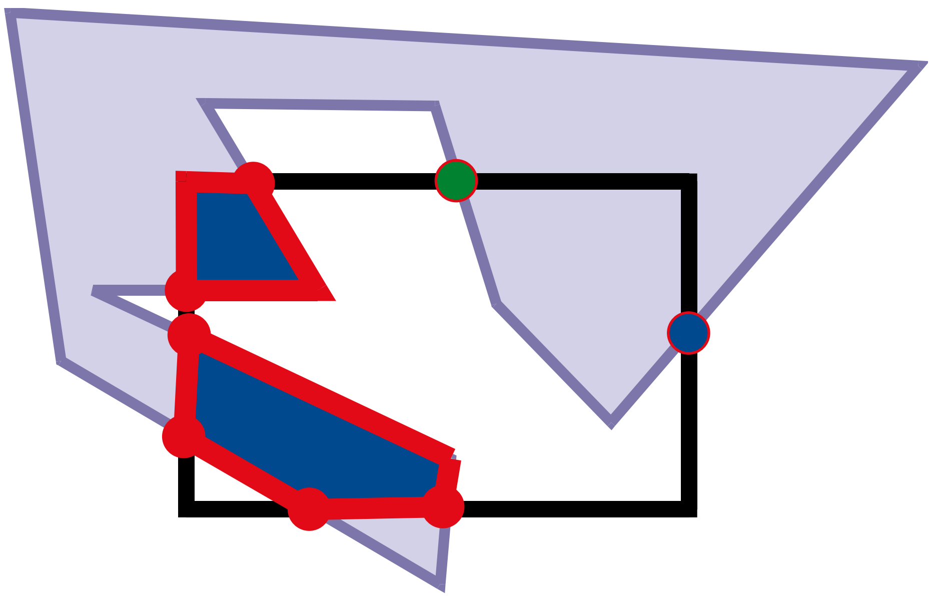
Walk" polygon/window boundary



Walking rules

While there is still an unprocessed entering intersection

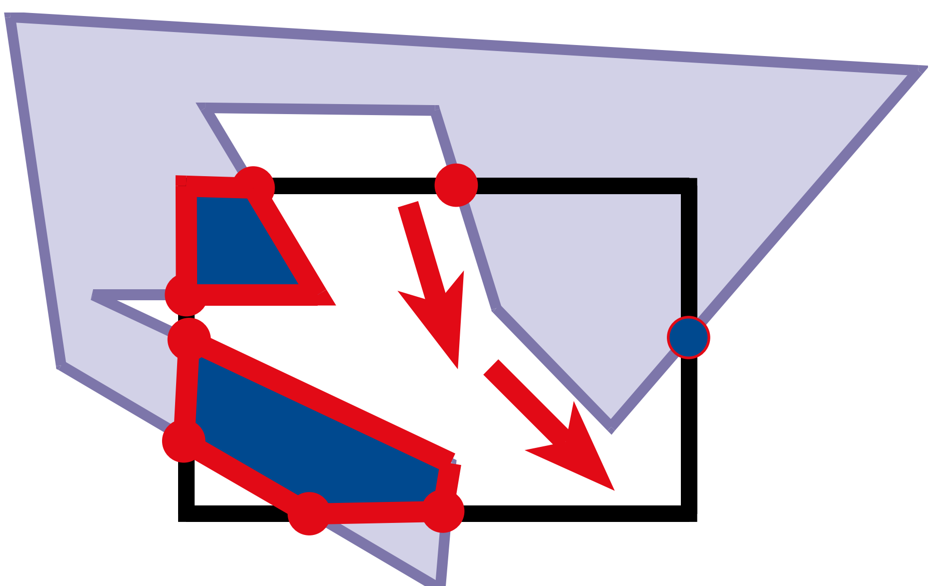
Walk" polygon/window boundary



Walking rules

While there is still an unprocessed entering intersection

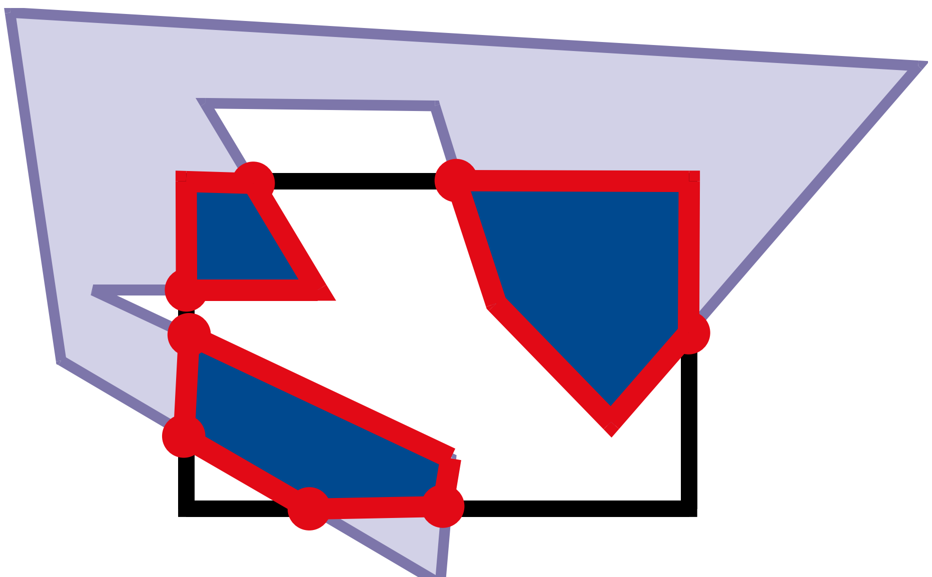
Walk" polygon/window boundary



Walking rules

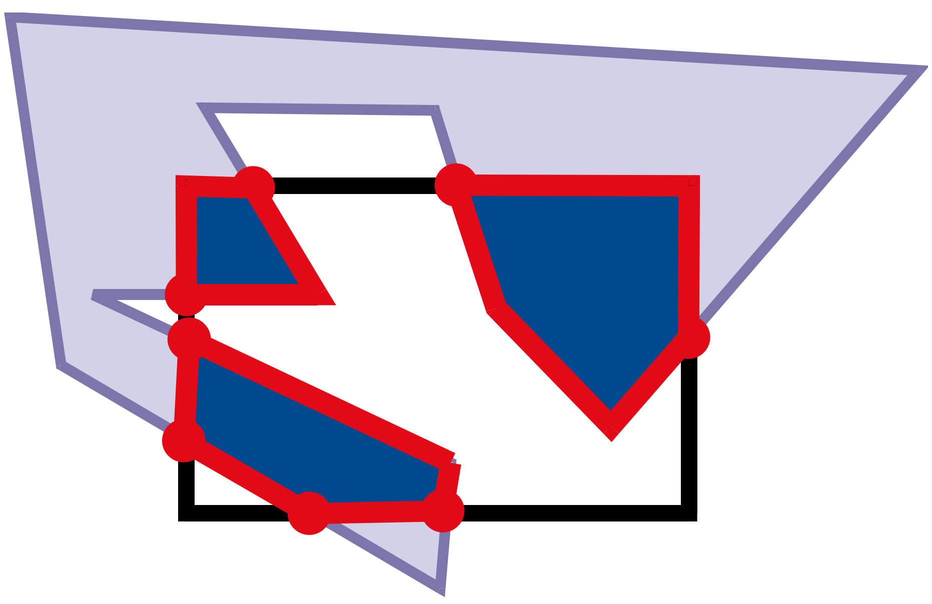
While there is still an unprocessed entering intersection

Walk” polygon/window boundary



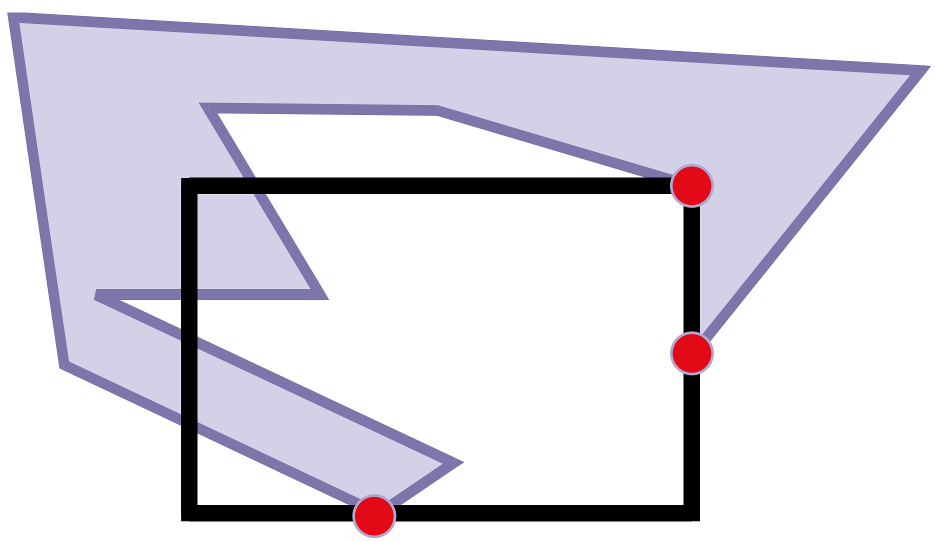
Weiler-Atherton Clipping

- Importance of good adjacency data structure (here simply list of oriented edges)



Robustness, precision, degeneracies

- What if a vertex is on the boundary?
- What happens if it is “almost” on the boundary?
  - Problem with floating point precision
- Welcome to the real world of geometry!

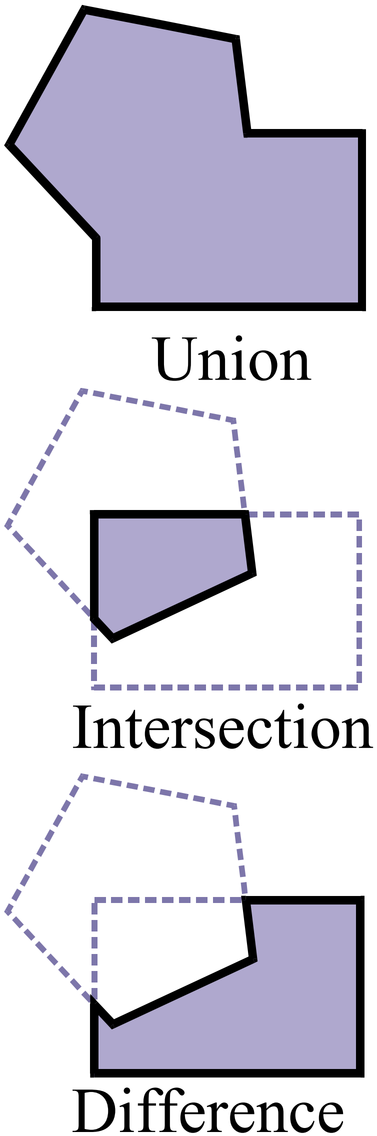


Clipping

- Many other clipping algorithms:
- Parametric, general windows, region-region, One-Plane-at-a-Time Clipping, etc.

Constructive Solid Geometry (CSG)

- Sort of generalized clipping
- Boolean operations
- Very popular in CAD/CAM
- CSG tree

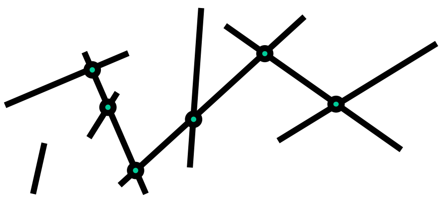


Ari Rappoport, Steven Spitz 97

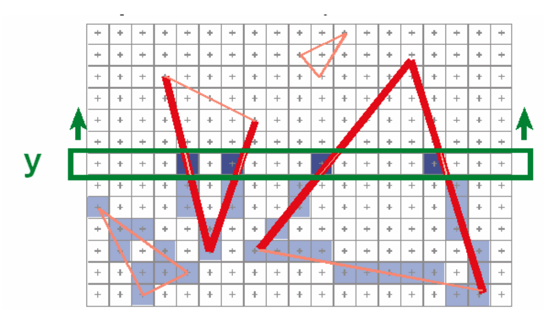
Plan

- Review of rendering pipeline
- 2D polygon clipping

- Segment intersection

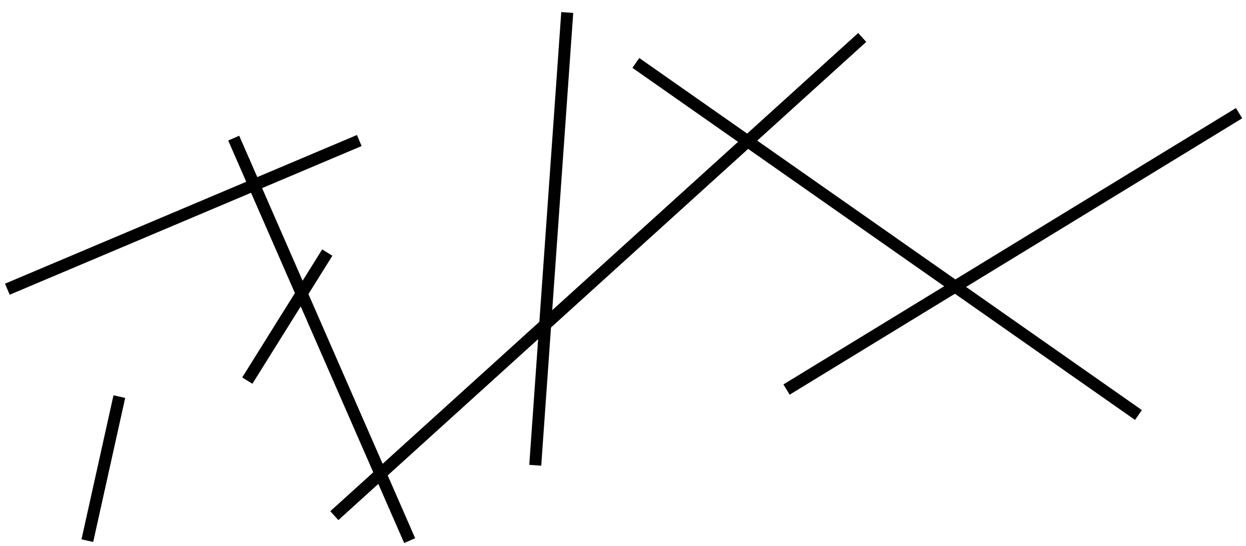


- Scanline rendering overview



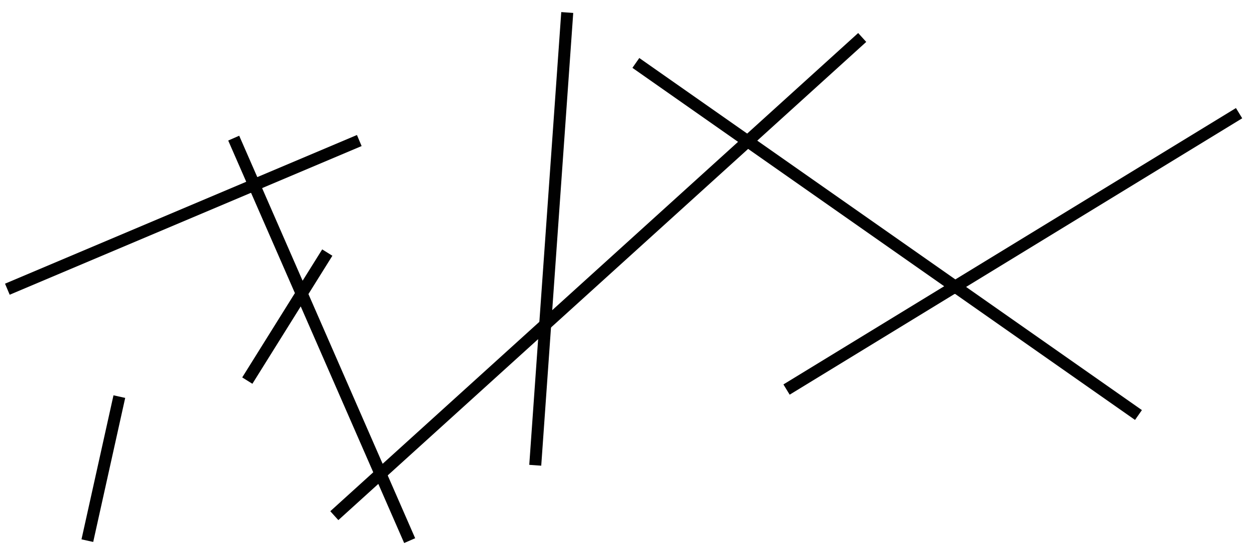
Line segment intersection

- N segments in the plane
- Find all intersections



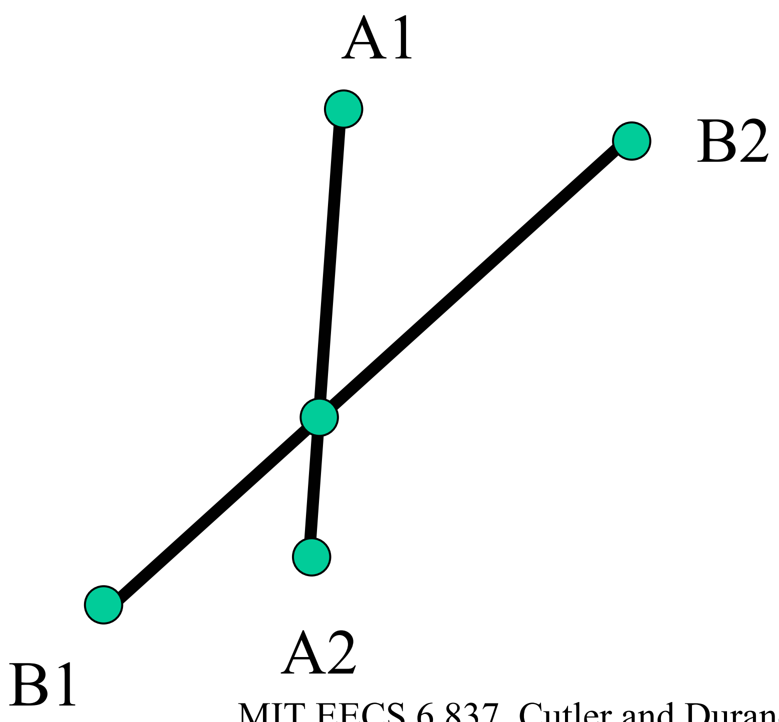
Maximum complexity?

- N^2
- (always N^2 if we take full lines)



Intersection between 2 segments

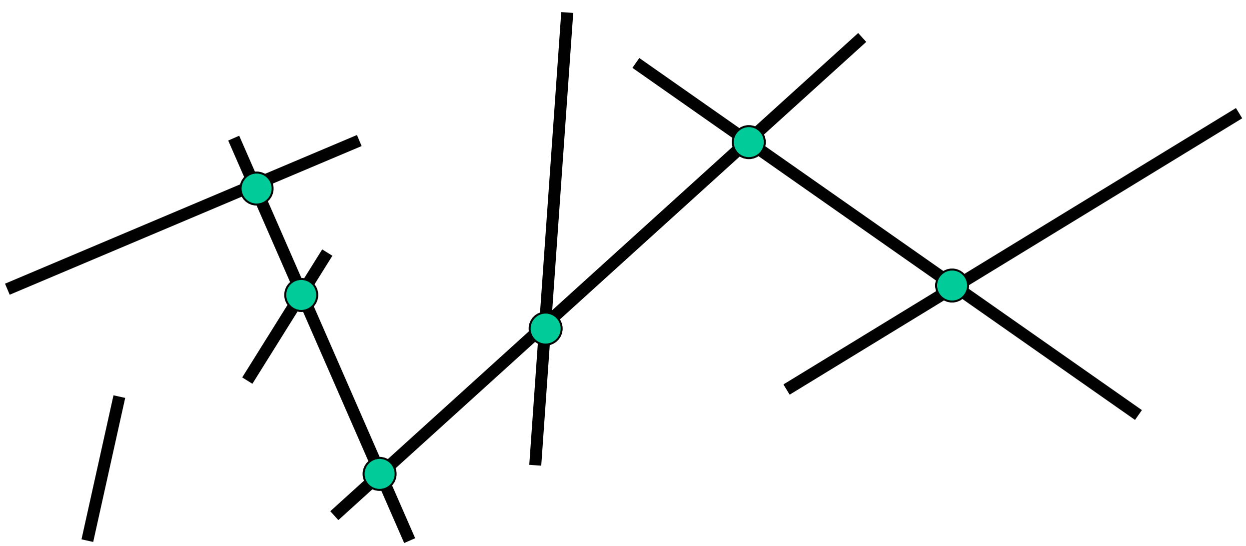
- Compute line equation for the 4 vertices
- If different signs
- Line intersection



Naïve algorithm

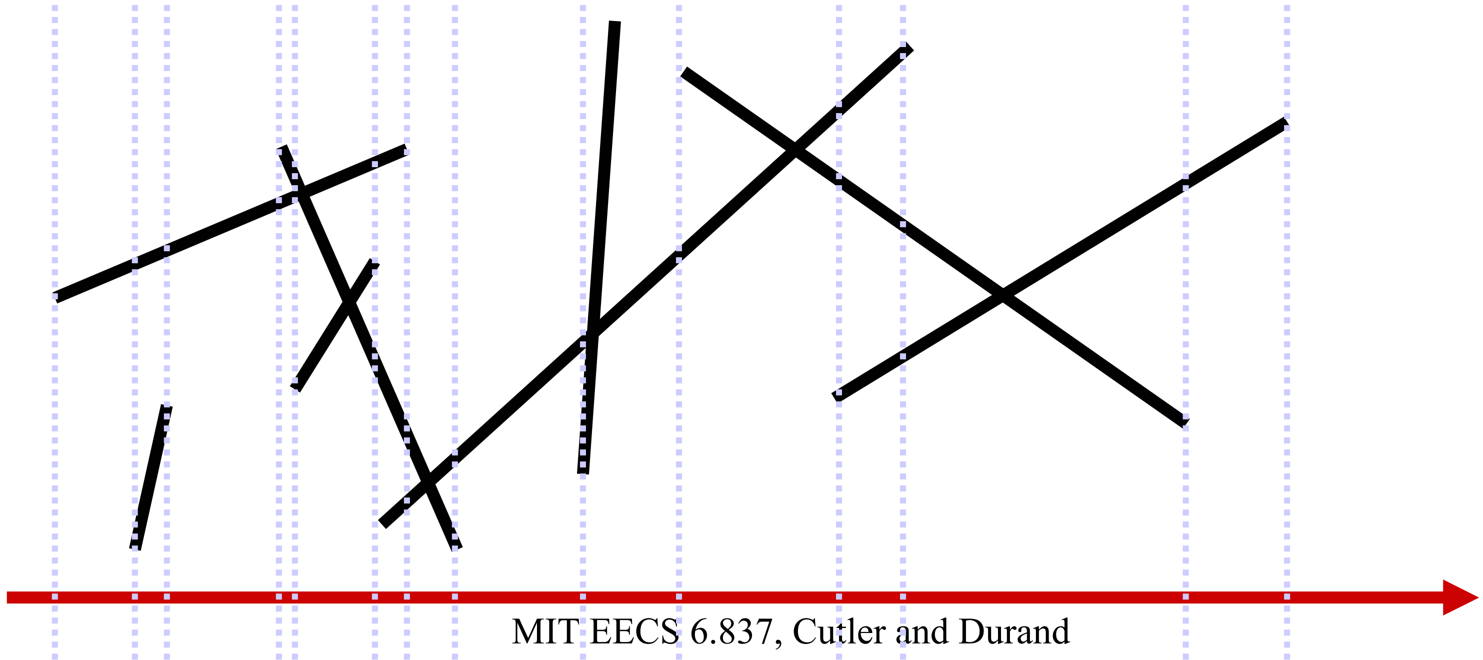
- N^2 intersection:

```
For (I=0; I<N; I++)
   For (J=I+1; J<N; J++)
      Compute intersection segments I and J
```



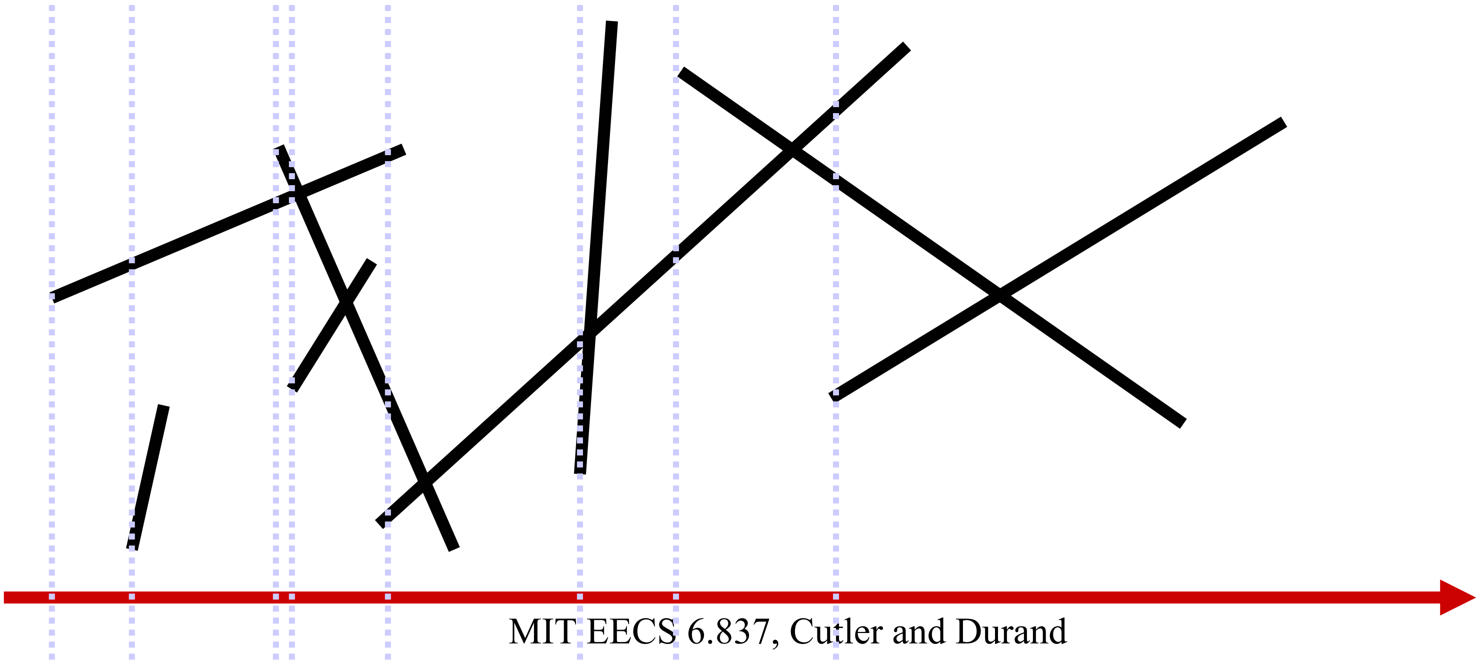
Taking advantage of coherence 1

- Sort in x
- Test only overlapping segments



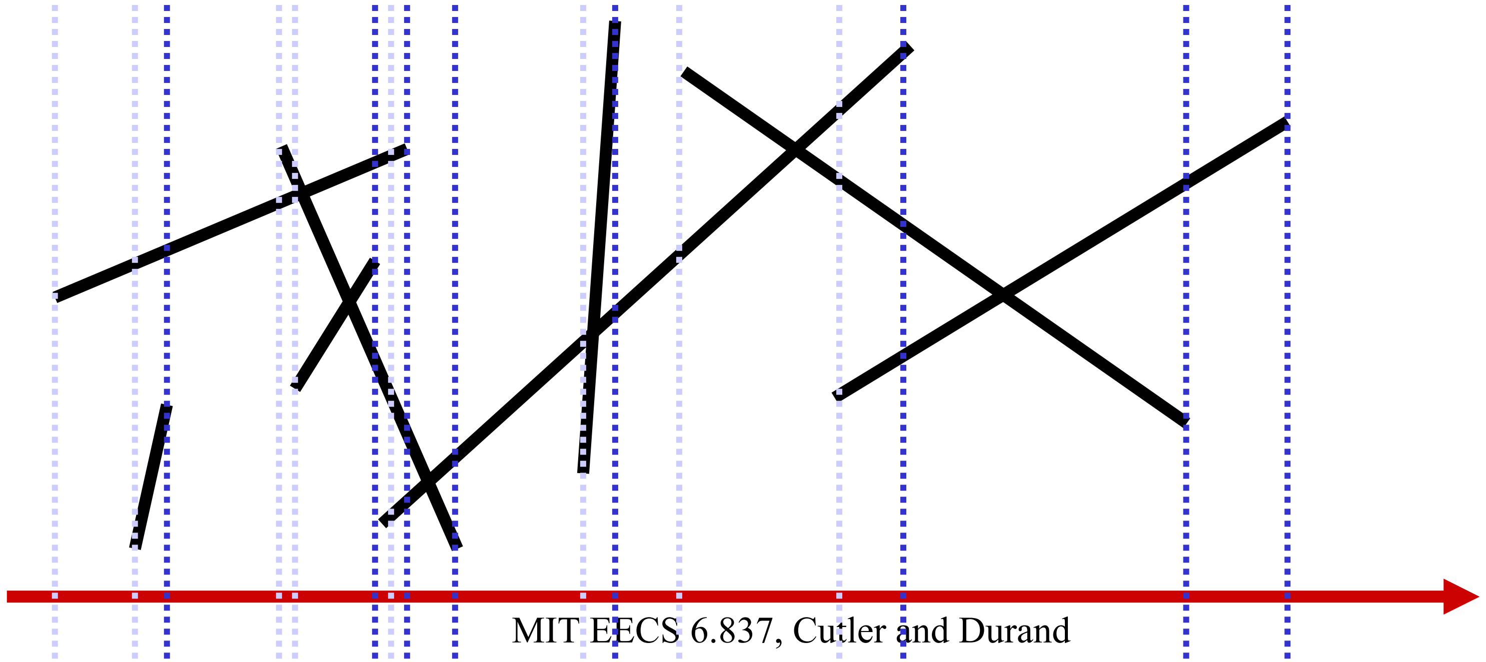
Taking advantage of coherence 1

```
Sort segments by xmin into queue Q
List ActiveSegments =empty
While Q not empty
    L= Q.next()  //pick next segment
    ActiveSegment->removeSegmentsBefore(L.xmin)
    For all segments Li in Active segments
        Compute Intersection between L and Li
    ActiveSegments->insert(L)
```



Taking advantage of coherence 1

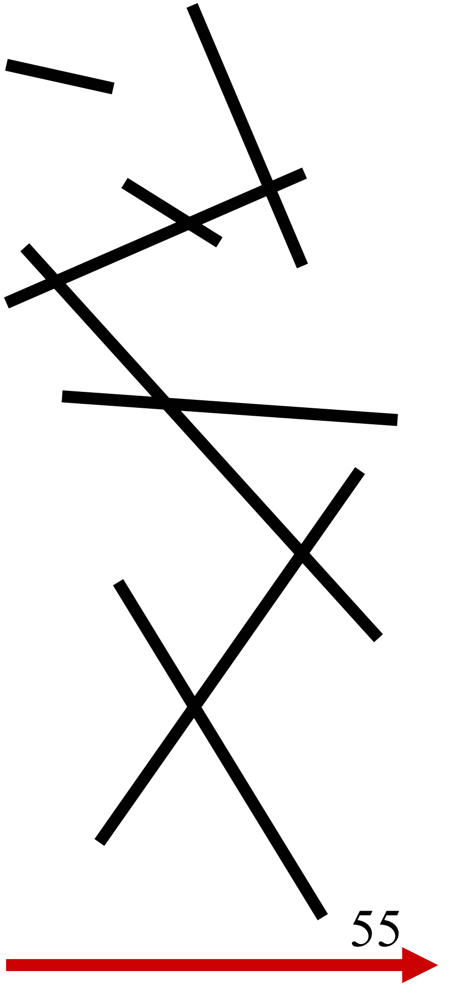
```
Sort segments by xmin into queue Q
List ActiveSegments =empty
While Q not empty
    L= Q.next()  //pick next segment
    ActiveSegment->removeSegmentsBefore(L.xmin)  //easier if sorted
    For all segments Li in Active segments
        Compute Intersection between L and Li
    ActiveSegments->insert(L)  //keep sorted by xmax
```



What have we achieved?

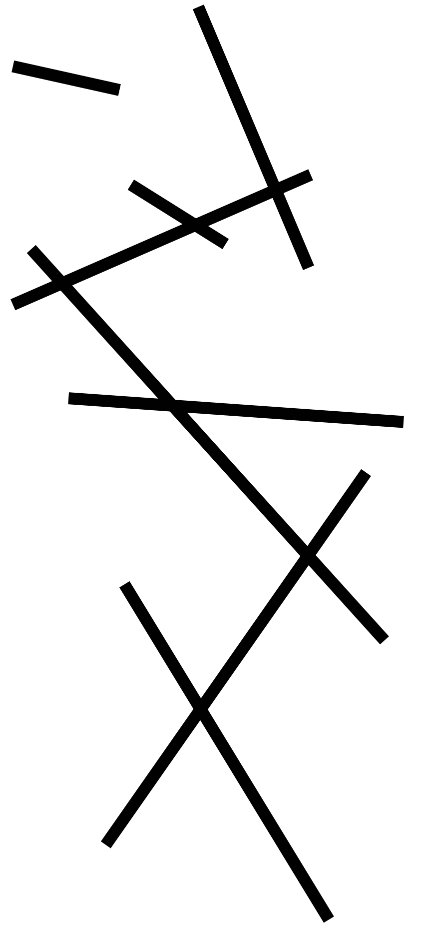
- Take advantage of locality and coherence
- Maintain working set
- Still $O(n^2)$
- But much better on average

- Can we do better?



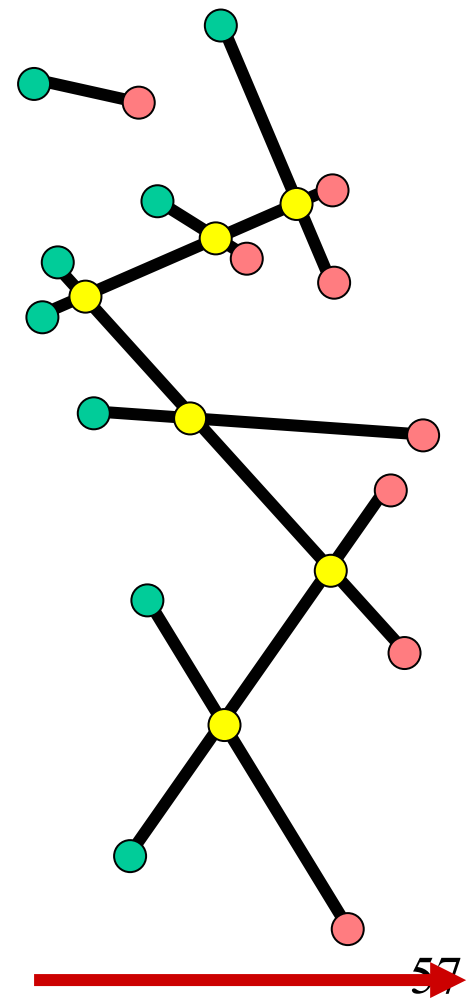
Can we do better?

- We have taken advantage of the coherence in x
- We have maintained a local view of the world at discrete events in x
- Do the same in y as well



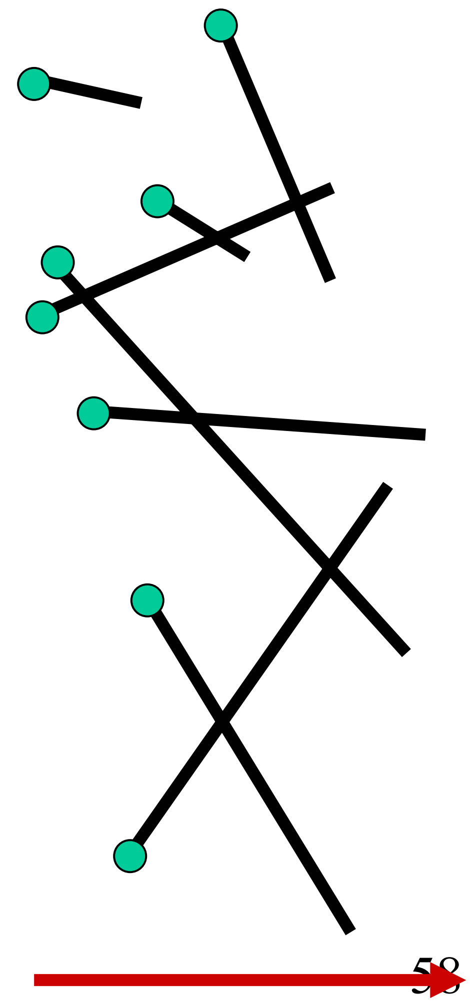
Maintain segments sorted in y

- Events
  - New segment
  - End of segment
  - Change of y sorting



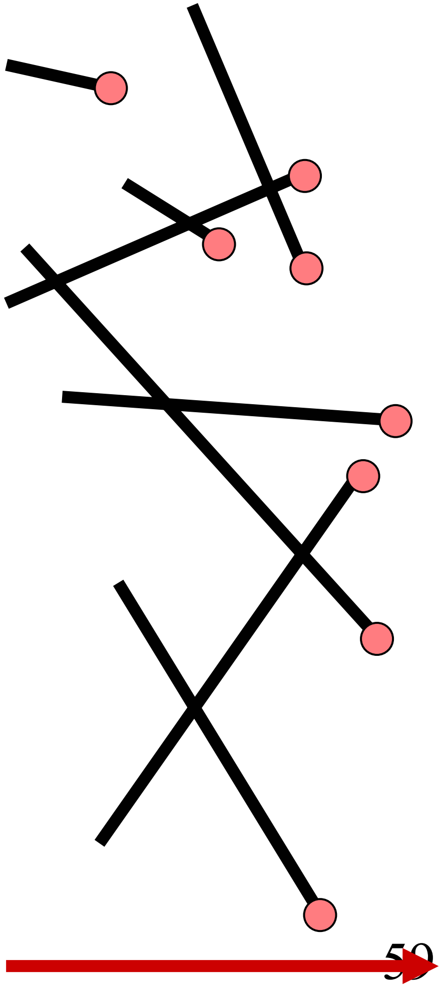
New segment

- Just insert at y1
- Use balanced binary trees



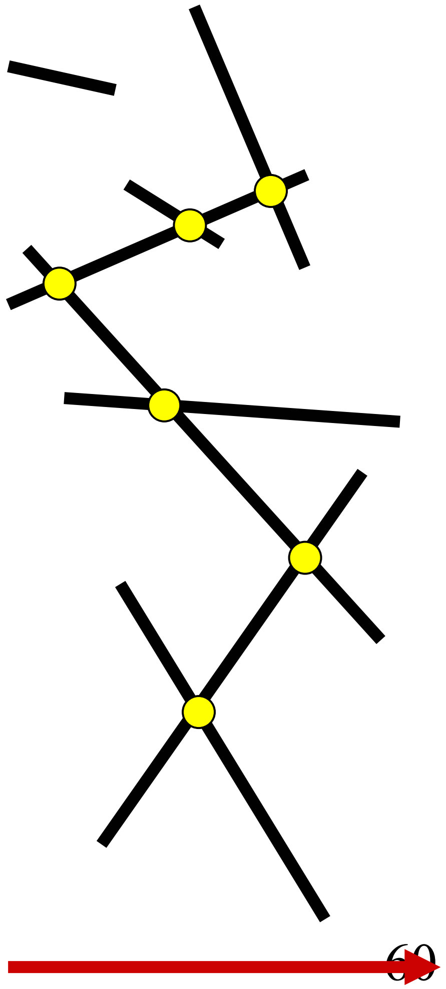
End of segment

- Just remove
- Potentially re-balance the tree



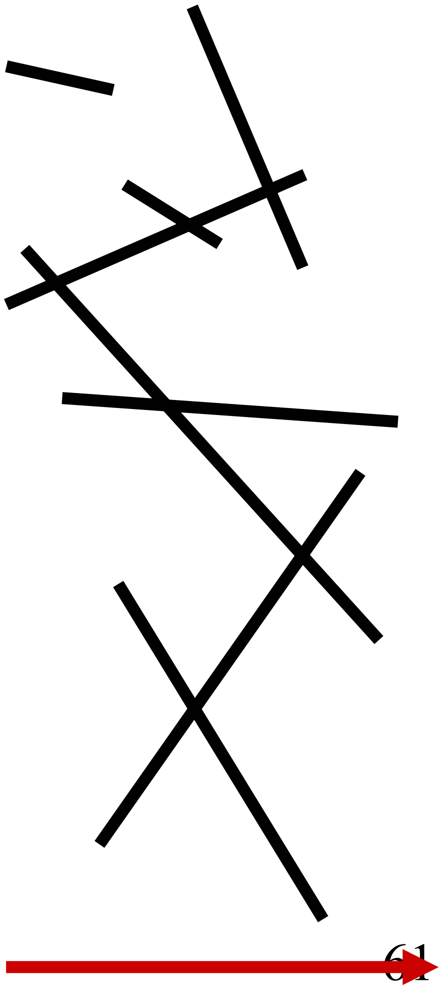
Intersection

- Where can intersection occur?
- Intersection must be between segments adjacents in y
- Fort each pair of adjacent segments, always maintain next intersection



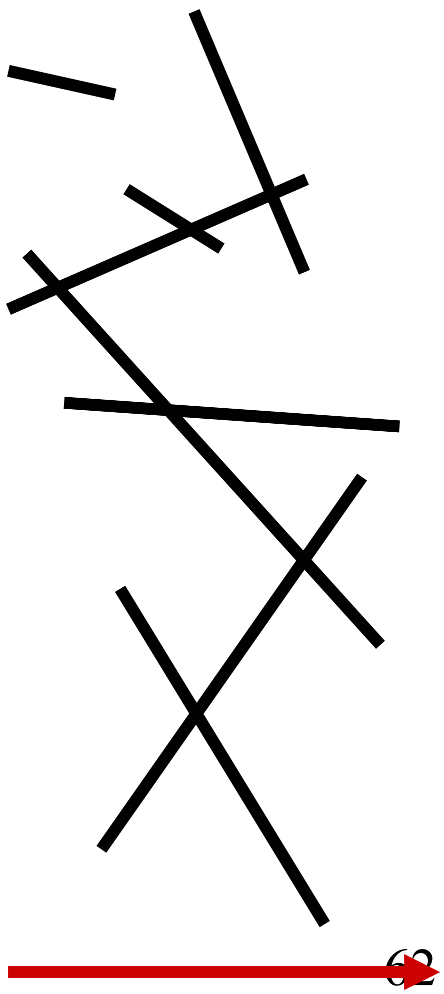
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



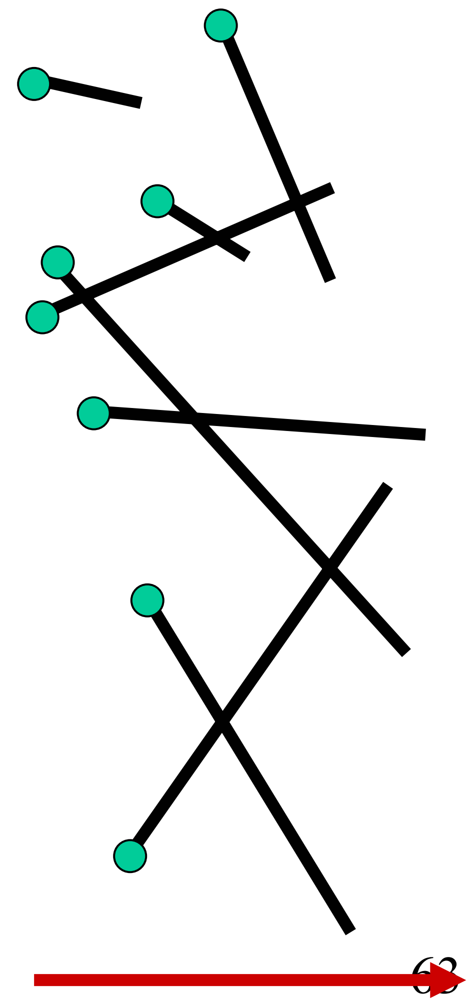
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



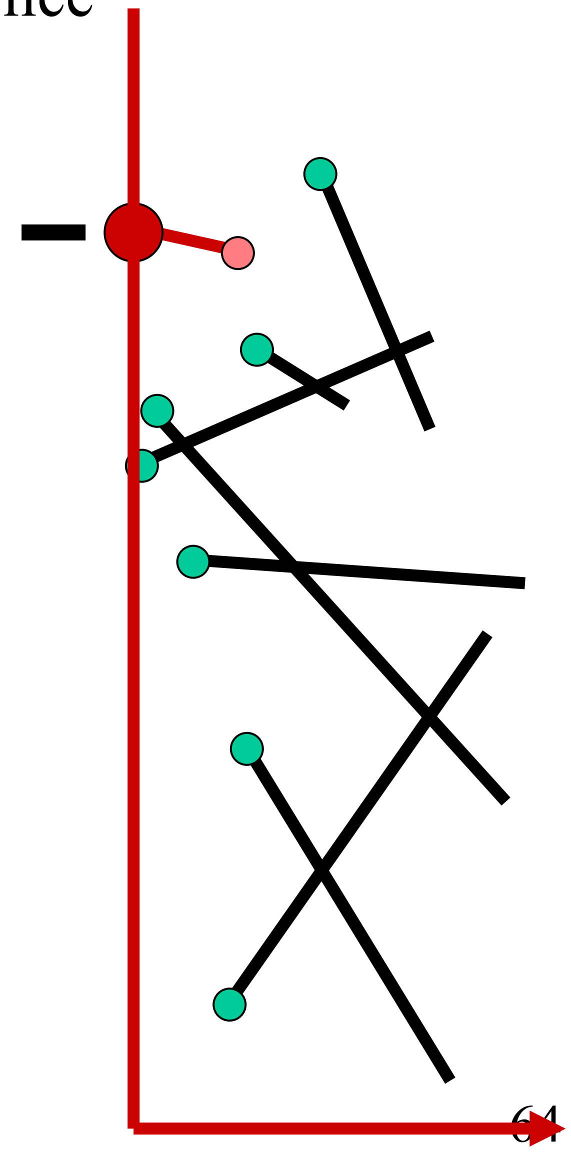
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



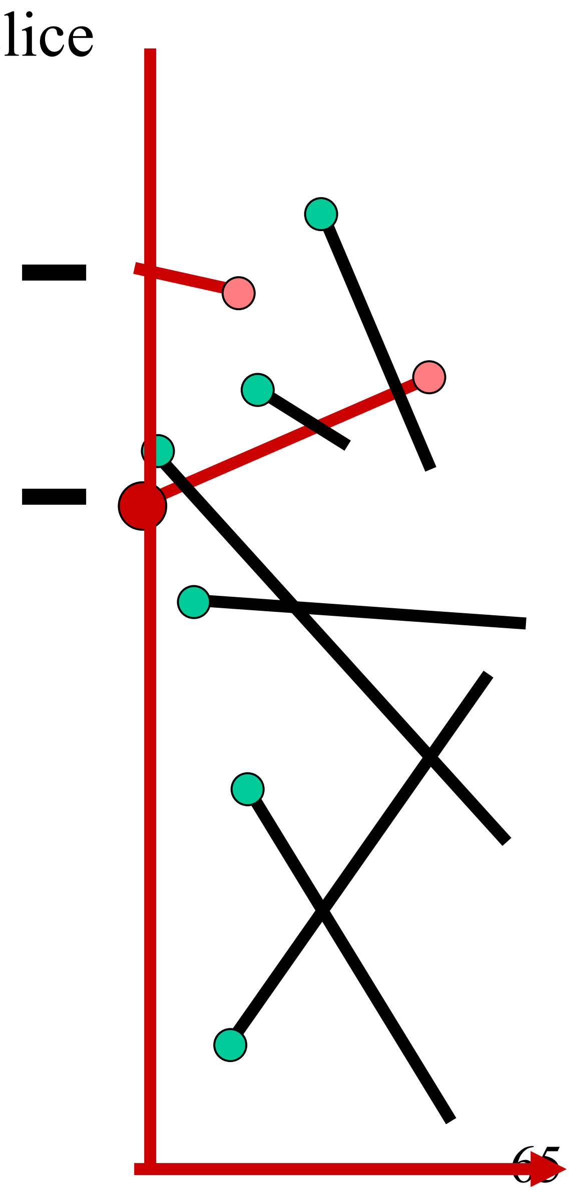
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



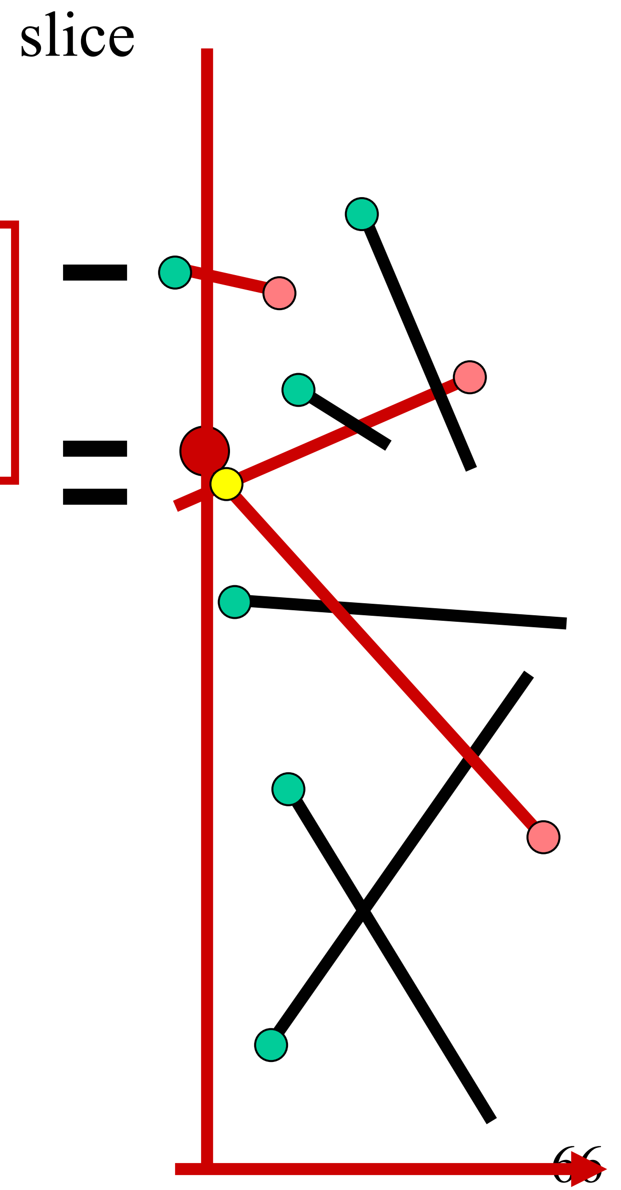
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



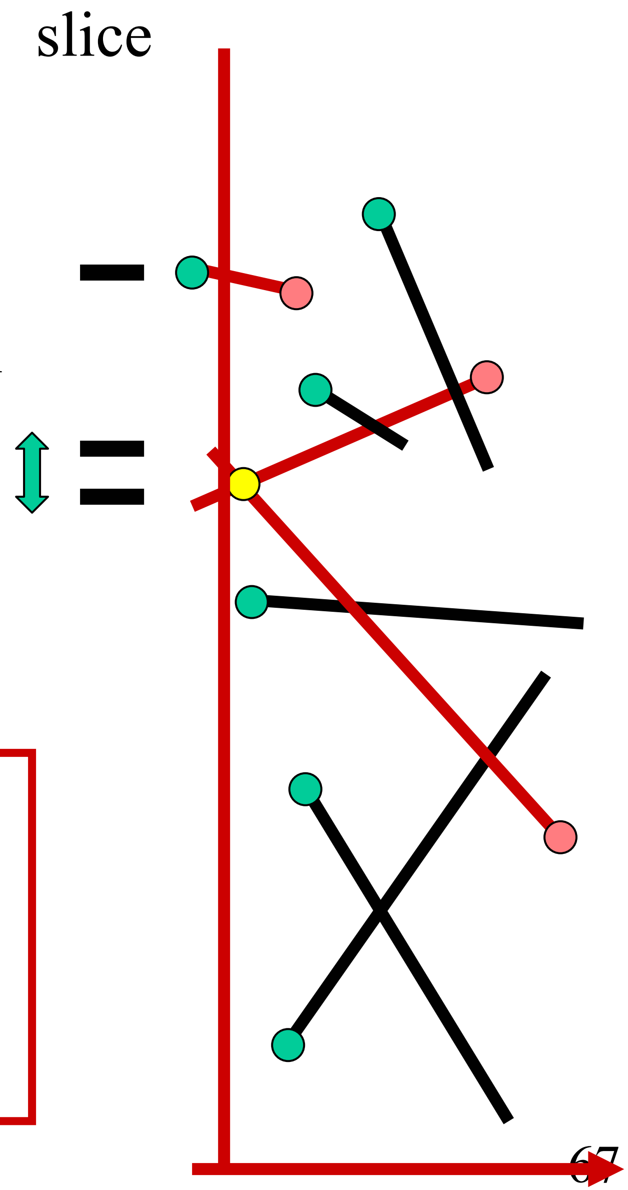
Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



Sweep algorithm

- Maintain event queue
  - New segment for each x1
    - Insert in binary tree
    - Compute potential new intersection
    - Add ending event
  - End of segment
    - simply remove
    - compute new intersections
  - Change of y sorting
    - report intersection
    - swap two segments
    - compute new intersections



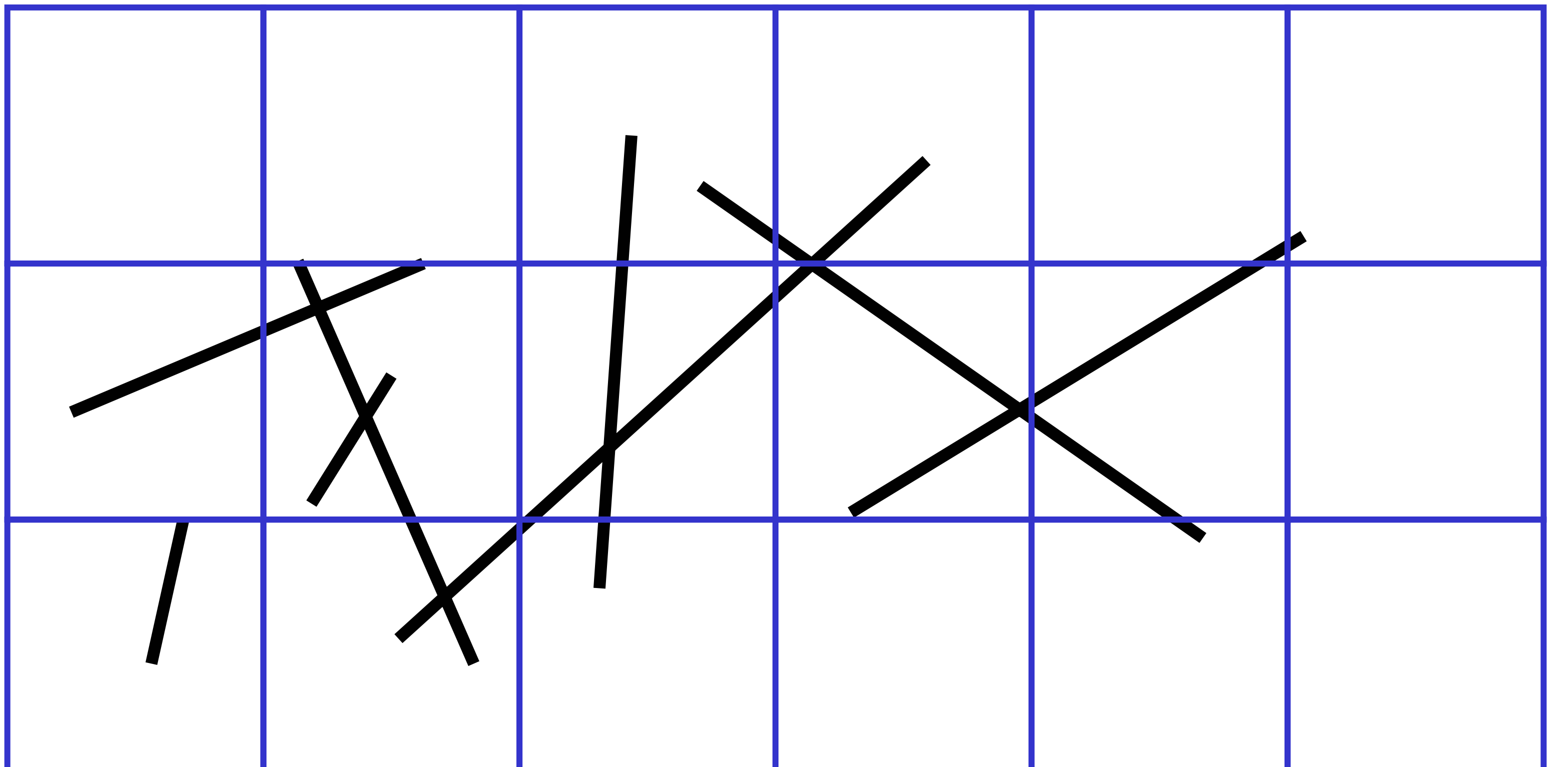
Output sensitive

- The running time depends on the output
- Hopefully linear in the output
  + smaller complexity in the input
- In our case time $O(n \log n + k \log n)$
  - Where k is the number of intersections
- Space: $O(n)$

- The optimal bound is time $O(n \log n + k)$

Other strategy?

- Grid!

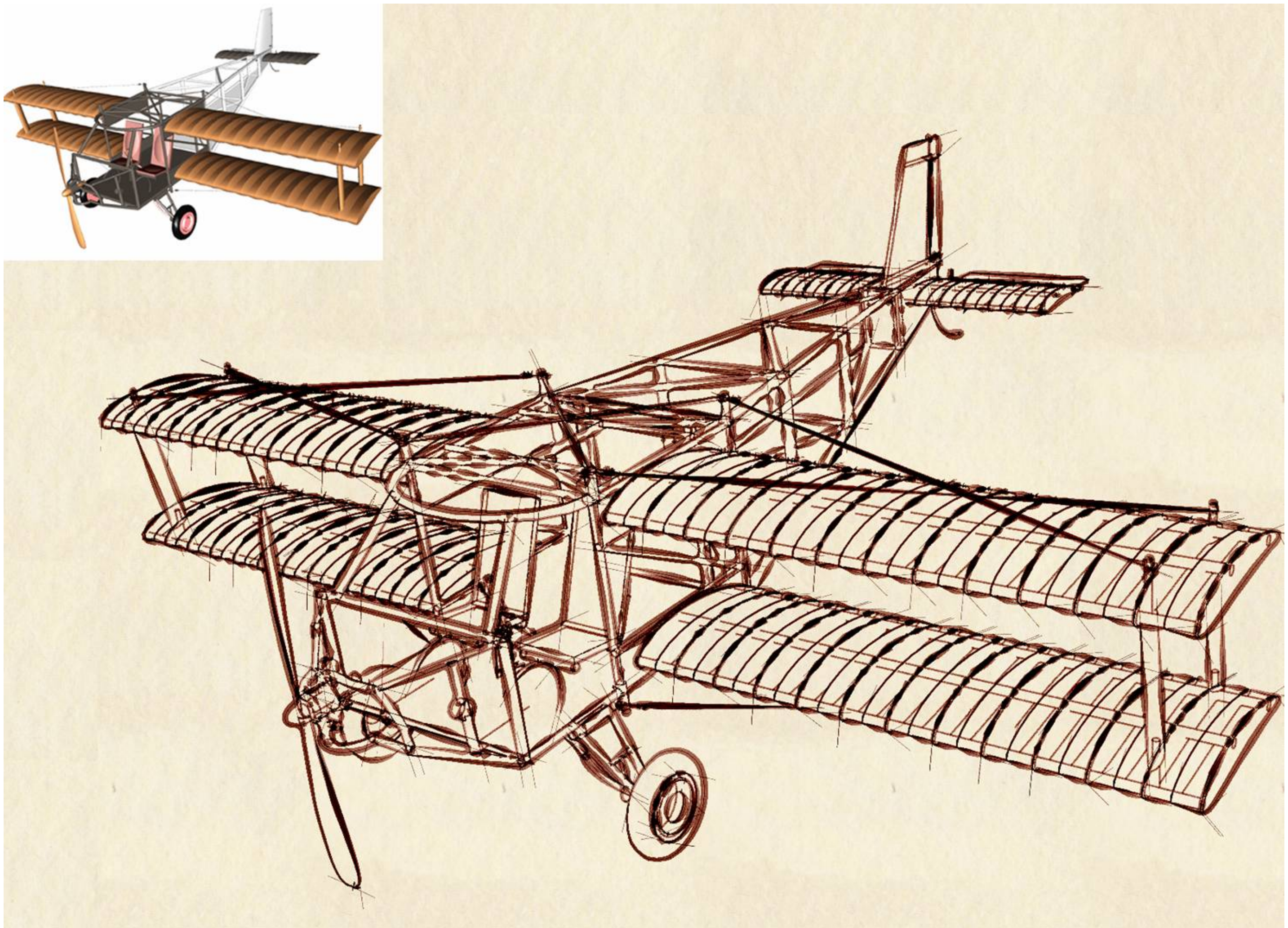


Ref

- De Berg, M. M. van Kreveld, M. Overmars. O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Ed. 2. Springer
- O'Rourke, Joseph. *Computational Geometry in C*. Ed. 2.

Questions?

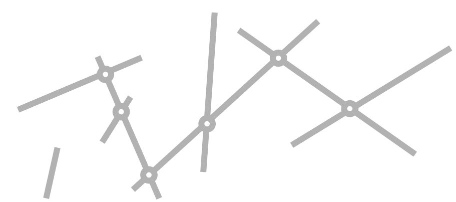
- Rendering this line drawing involved the intersection of all stroke segments



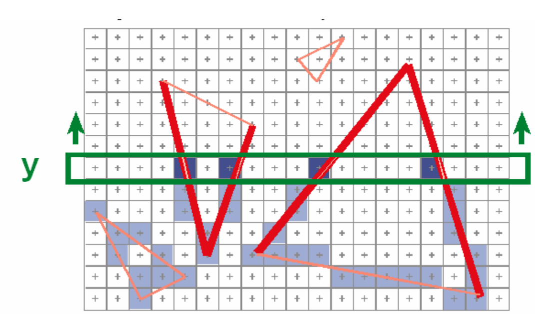
Plan

- Review of rendering pipeline
- 2D polygon clipping

- Segment intersection

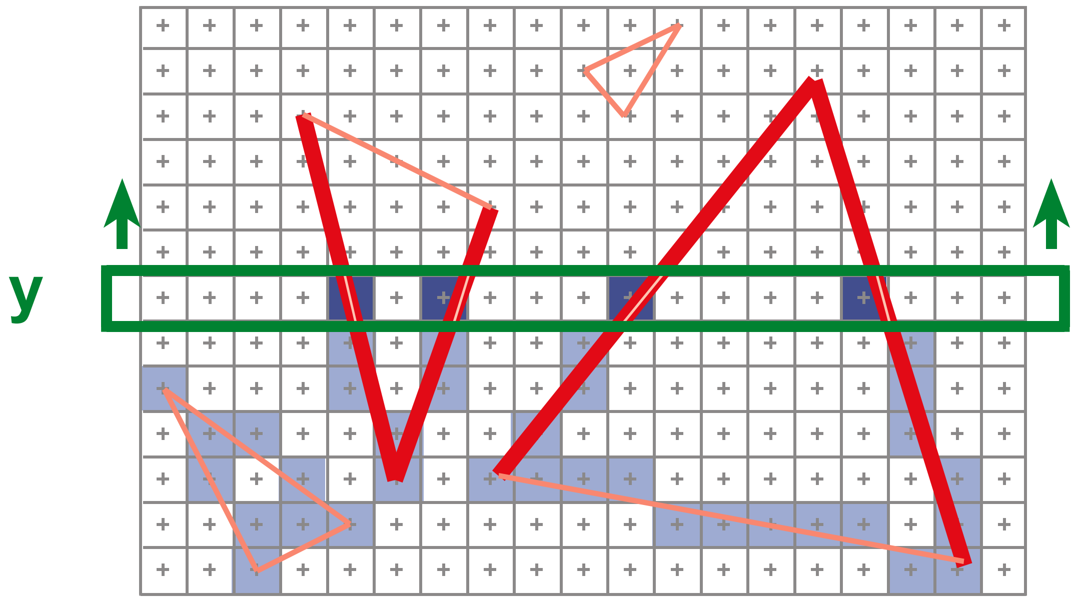


- Scanline rendering overview



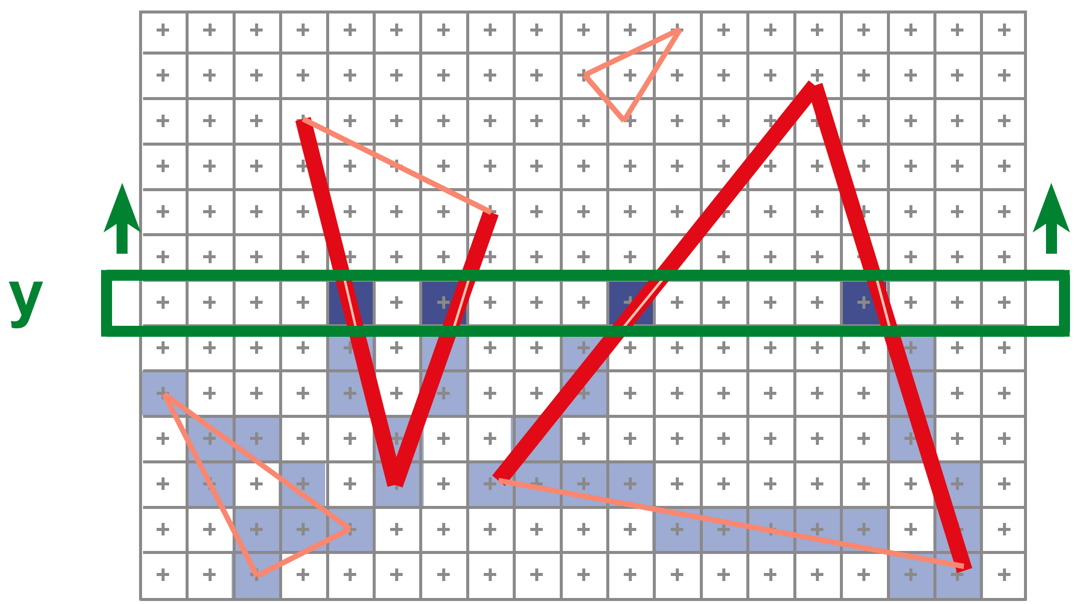
Scan Line rasterization

- Draw one scanline at a time
- Maintain ordered slices of triangles
- Advantage, does not require whole model and whole image in memory



Scan Line : Principle

- Proceed row by row
- Maintain Active Edge List (AEL) (EdgeRecList)
- Edge Table (ET) for new edges at y (EdgeRecTable)



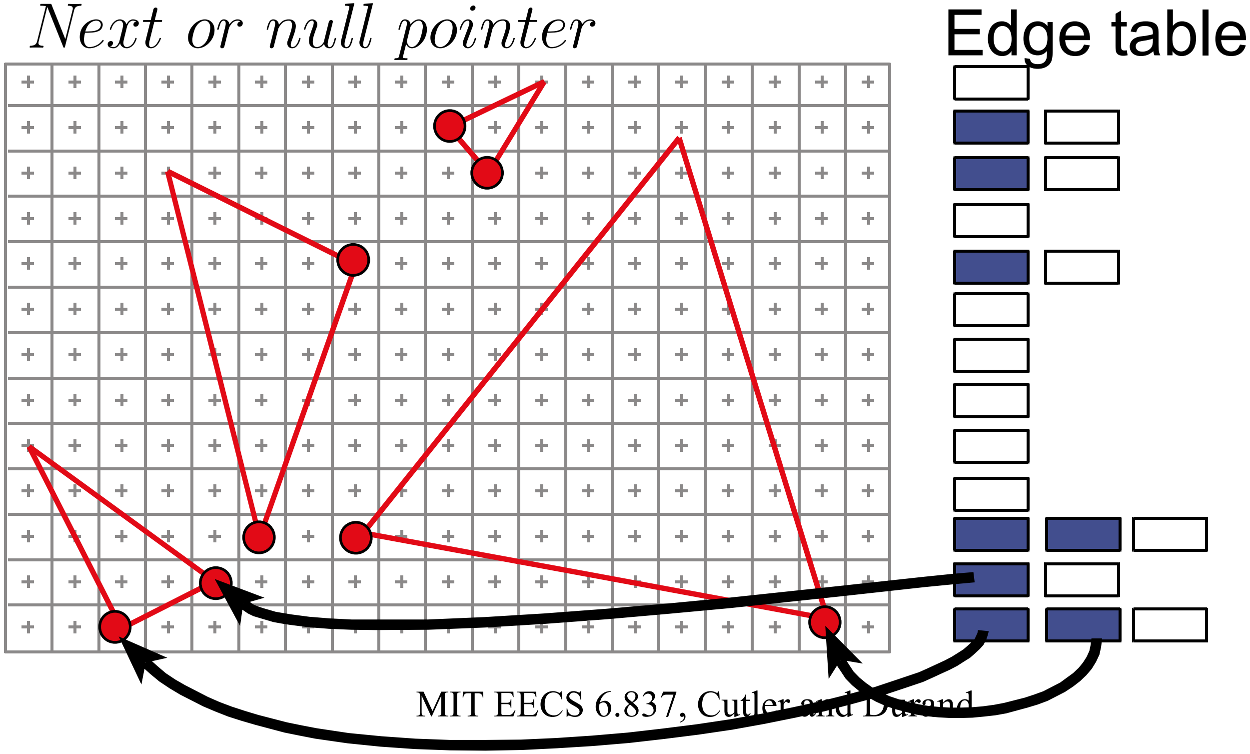
Precompute: Edge Table

- One entry per scan line (where edge begins)
- Each entry is a linked list of **`Edges,`** sorted by x

  y**end**: y of top edge endpoint

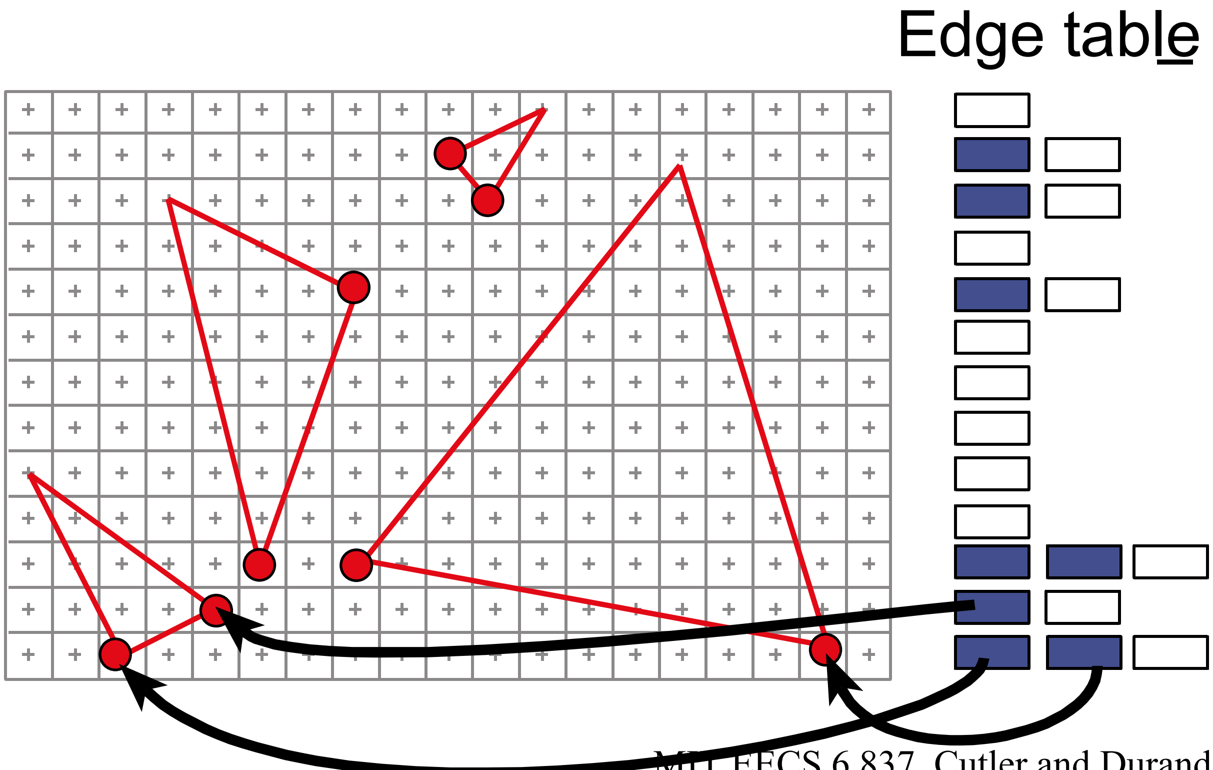
  x**curr**, **x**: current x intersection, delta wrt y

  Next or null pointer



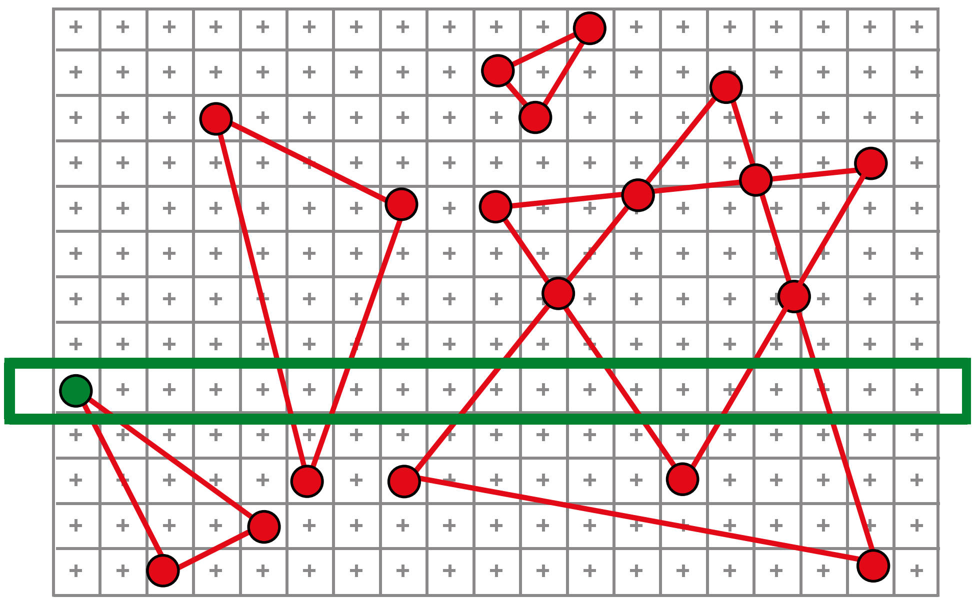
Initialization: events

- **Edge Table**
  - List of `Edges`, sorted by x
    - $y$end
    - $x$curr, delta wrt y
- **Active edge list (AEL)**
  - Will be maintained
  - Store all active edges intersecting scanline
  - Ordered by x



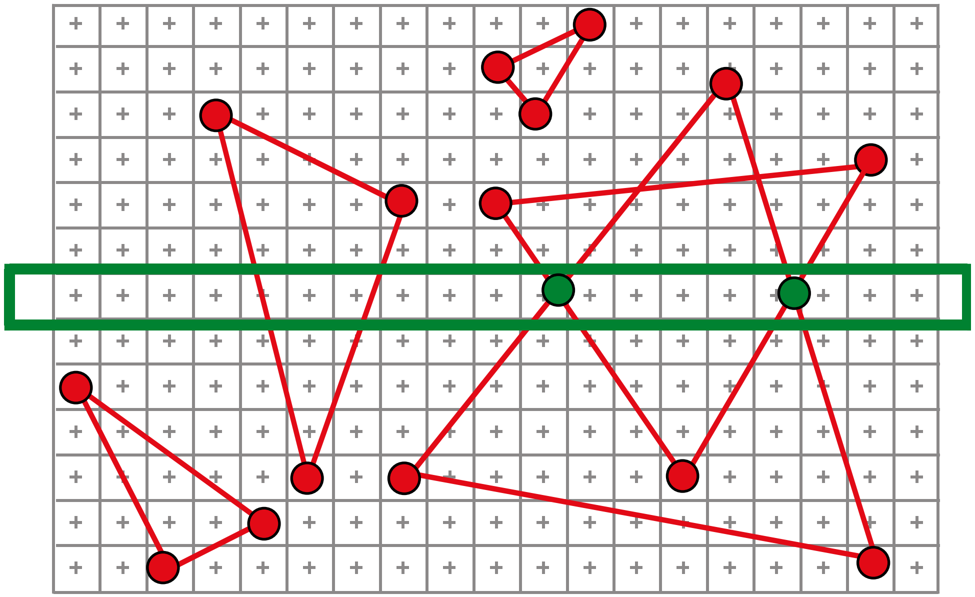
When Does AEL Change State?

- When a vertex is encountered
  - When an edge begins
    - All such events pre-stored in Edge Table
  - When and edge ends
    - Can be deduced from current Active Edge List



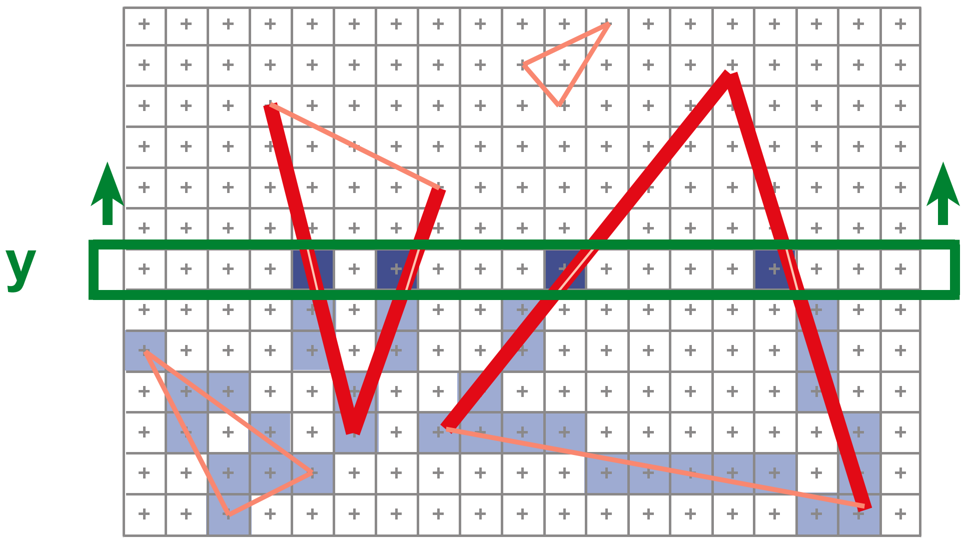
When Does AEL Change State?

- When a vertex is encountered
- When two edges change order along a scanline
  - I.e., when edges cross each other!
  - How to detect this efficiently?



Scanline algorithm summary

- Initialize Raster, Polygons, **Edge Table, AEL**
- For each scanline y
  - Update Active Edge List (insert edges from EdgeTable[y])
  - Assign raster of pixels from AEL
  - Update AEL (delete, increment, resort)



Other sweep algorithms

- Sweep is a very general principle:
  - Maintain a slice
  - Update at events
  - Works well if events are predictable locally in the slice (regular)
- Applied to many problems
  - E.g. construction of weird visibility data structures in 4.5D