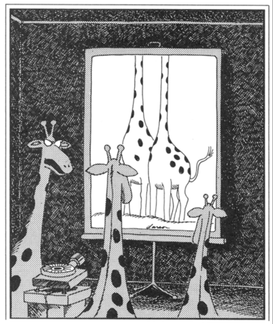
The Graphics Pipeline: Clipping & Line Rasterization

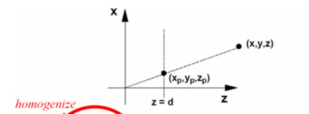


"Oh, lovely — just the hundredth time you've managed to cut everyone's head off."

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

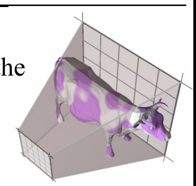


$$\begin{pmatrix} x*d/z \\ y*d/z \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

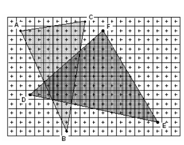
Today: Clipping & Line Rasterization

- 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

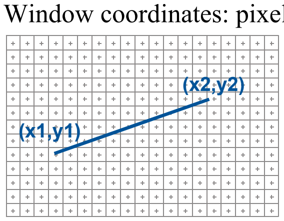


Today

• Why Clip?
• Line Clipping
• Polygon clipping
• Line Rasterization

Framebuffer Model

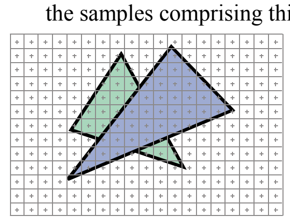
• Raster Display: 2D array of picture elements (pixels)
• Pixels individually set/cleared (greyscale, color)
• Window coordinates: pixels centered at integers

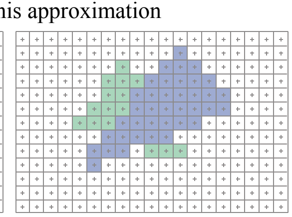



```
glBegin(GL_LINES)
glVertex3f(...)
glVertex3f(...)
glEnd();
```


2D Scan Conversion

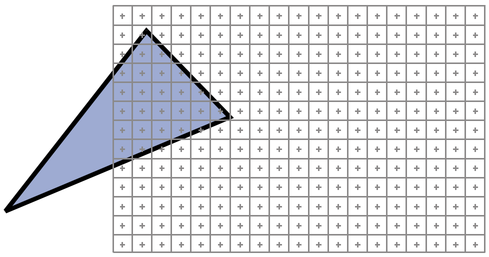
• Geometric primitives
  (point, line, polygon, circle, polyhedron, sphere...)
• Primitives are continuous; screen is discrete
• Scan Conversion: algorithms for *efficient* generation of the samples comprising this approximation





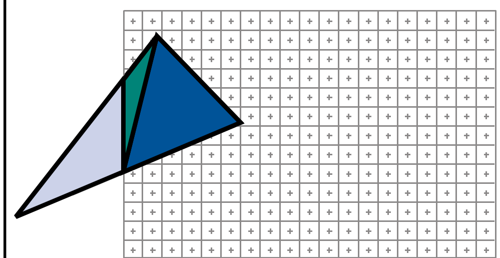
Clipping problem

- How do we clip parts outside window?

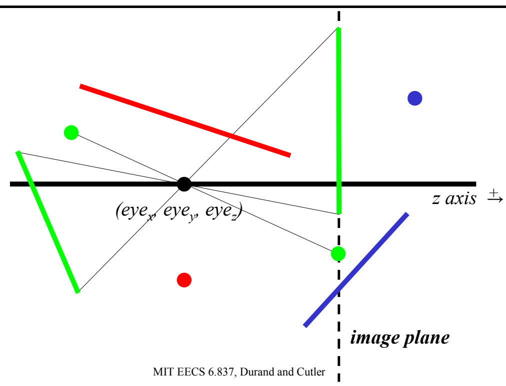


Clipping problem

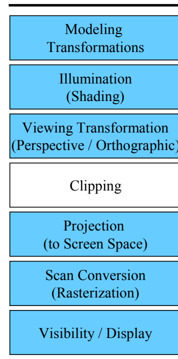
- How do we clip parts outside window?
- Create two triangles or more. Quite annoying.



Also, what if the p_z is < eye_z?

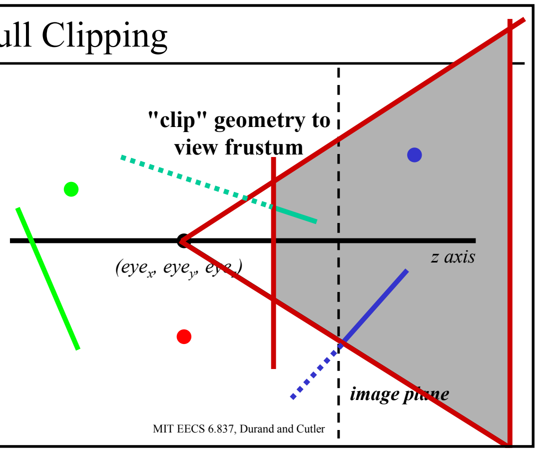


The Graphics Pipeline

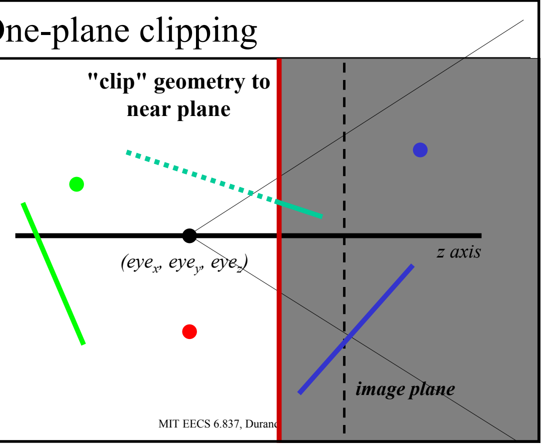


- Former hardware relied on full clipping
- Modern hardware mostly avoids clipping
  - Only with respect to plane z=0
- In general, it is useful to learn clipping because it is similar to many geometric algorithms

Full Clipping

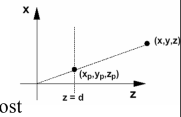


One-plane clipping



When to clip?

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

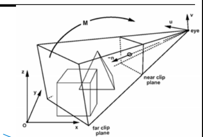


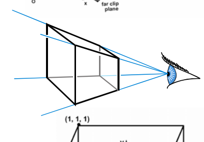
homogenize

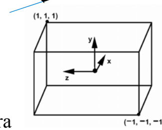
$$\begin{pmatrix} x*d/z \\ y*d/z \\ d/z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z/d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

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







Working in homogeneous coordinates

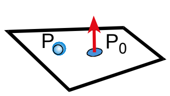
- In general, many algorithms are simpler in homogeneous coordinates before division
  - Clipping
  - Rasterization

Today

- Why Clip?
- Line Clipping
- Polygon clipping
- Line Rasterization

Implicit 3D Plane Equation

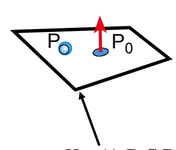
- Plane defined by:
  point *p* & normal *n* OR
  normal *n* & offset *d* OR
  3 points



- Implicit plane equation
  A*x*+B*y*+C*z*+D = 0

Homogeneous Coordinates

- Homogenous point: (x,y,z,w)
  infinite number of equivalent homogenous coordinates:
  (sx, sy, sz, sw)

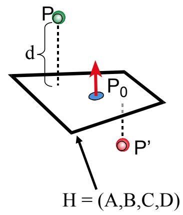


H = (A,B,C,D)

- Homogenous Plane Equation:
  A*x*+B*y*+C*z*+D = 0 → H = (A,B,C,D)
  Infinite number of equivalent plane expressions:
  sA*x*+sB*y*+sC*z*+sD = 0 → H = (sA,sB,sC,sD)

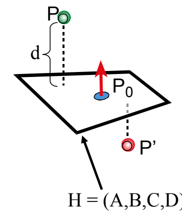
Point-to-Plane Distance

- If (A,B,C) is normalized:
  - d = H•p = $H^T p$
  - (the dot product in homogeneous coordinates)
- d is a *signed distance*
  - positive = "inside"
  - negative = "outside"



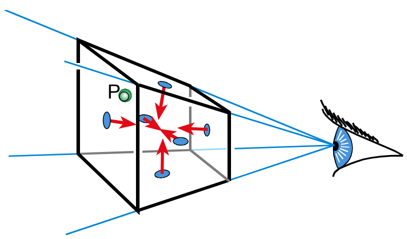
Clipping a Point with respect to a Plane

- If d = H•p ≥ 0
  Pass through
- If d = H•p < 0:
  Clip (or cull or reject)

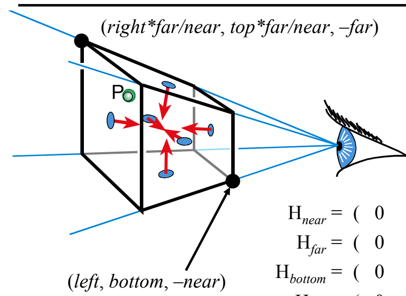


Clipping with respect to View Frustum

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



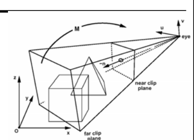
What are the View Frustum Planes?

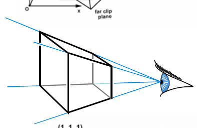


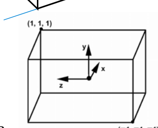
H_{near} = (0 0 –1 –*near*)
H_{far} = (0 0 1 *far*)
H_{bottom} = (0 *near* *bottom* 0)
H_{top} = (0 –*near* –*top* 0)
H_{left} = (*left* *near* 0 0)
H_{right} = (–*right* –*near* 0 0)

Recall: 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





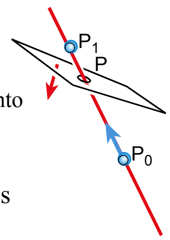


Questions?

- You are now supposed to be able to clip points wrt view frustum
- Using homogeneous coordinates

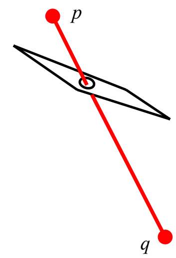
Line – Plane Intersection

- 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.)



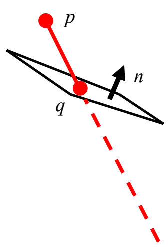
Segment Clipping

- 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



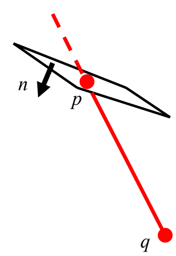
Segment Clipping

- 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



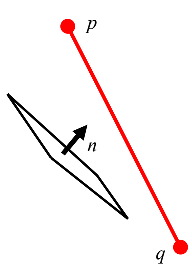
Segment Clipping

- 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



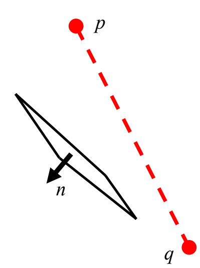
Segment Clipping

- 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



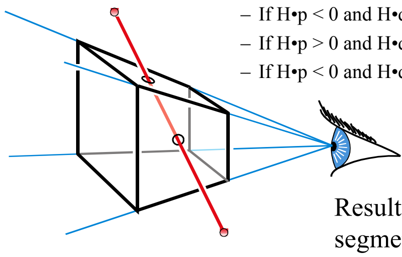
Segment Clipping

- 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



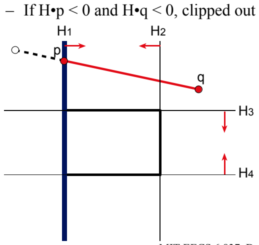
Result is a single segment. Why?

Questions?

- You are now supposed to be able to clip segments wrt view frustum

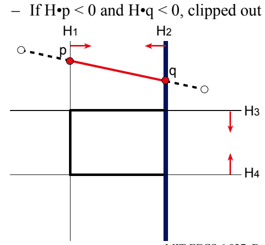
Is this Clipping Efficient?

- 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



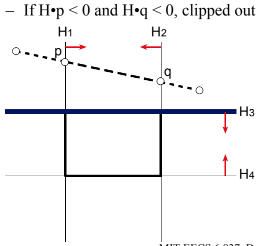
Is this Clipping Efficient?

- 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



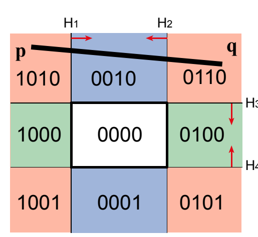
What is the problem?

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

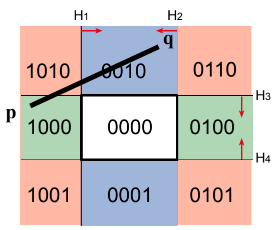
Outcode of q : 0110

Outcode of [pq] : 0010

Clipped because there is a 1

Improving Efficiency: Outcodes

- When do we fail to save computation?



Outcode of p : 1000

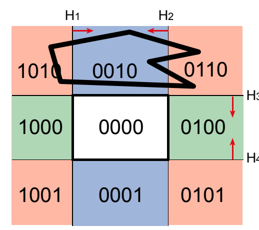
Outcode of q : 0010

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

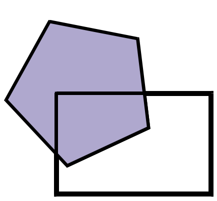
Questions?

- You are now supposed to be able to make clipping efficient using outcodes

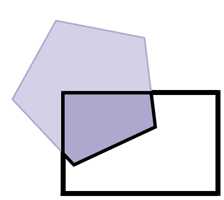
Today

- Why Clip?
- Line Clipping
- Polygon clipping
- Line Rasterization

Polygon clipping

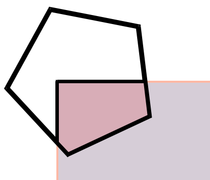


Polygon clipping



Polygon clipping

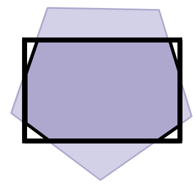
- Clipping is symmetric



MIT EECS 6.837, Durand and Cutler

Polygon clipping is complex

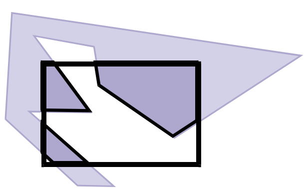
- Even when the polygons are convex



MIT EECS 6.837, Durand and Cutler

Polygon clipping is nasty

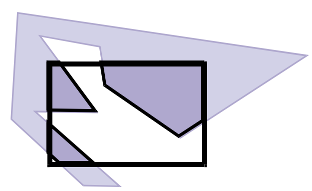
- When the polygons are concave



MIT EECS 6.837, Durand and Cutler

Naïve polygon clipping?

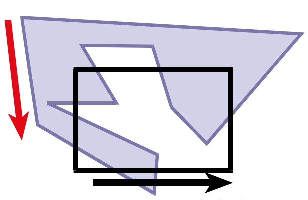
- N*m intersections
- Then must link all segment
- Not efficient and not even easy



MIT EECS 6.837, Durand and Cutler

Weiler-Atherton Clipping

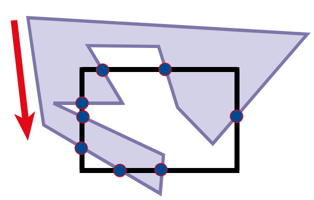
- Strategy: "Walk" polygon/window boundary
- Polygons are oriented (CCW)



MIT EECS 6.837, Durand and Cutler

Weiler-Atherton Clipping

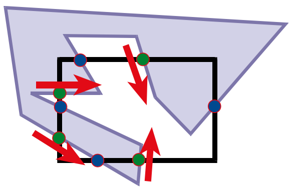
- Compute intersection points



MIT EECS 6.837, Durand and Cutler

Weiler-Atherton Clipping

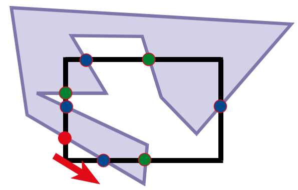
- Compute intersection points
- Mark points where polygons enters clipping window (green here)



MIT EECS 6.837, Durand and Cutler

Clipping

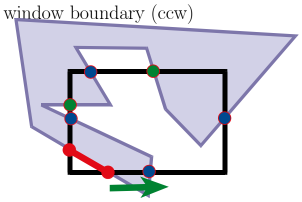
While there is still an unprocessed entering intersection
Walk" polygon/window boundary



MIT EECS 6.837, Durand and Cutler

Walking rules

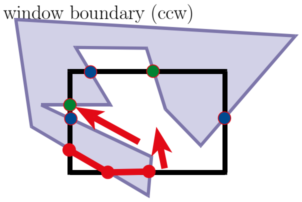
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



MIT EECS 6.837, Durand and Cutler

Walking rules

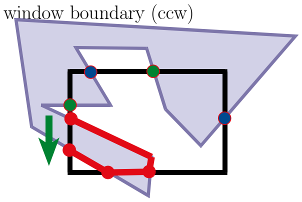
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



MIT EECS 6.837, Durand and Cutler

Walking rules

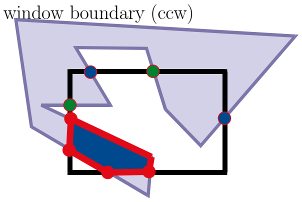
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



MIT EECS 6.837, Durand and Cutler

Walking rules

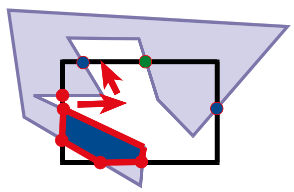
- Out-to-in pair:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out pair:
  - Record clipped point
  - Follow window boundary (ccw)



MIT EECS 6.837, Durand and Cutler

Walking rules

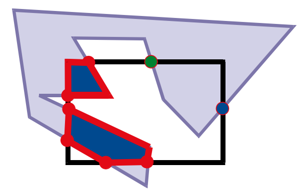
While there is still an unprocessed entering intersection
Walk” polygon/window boundary



MIT EECS 6.837, Durand and Cutler

Walking rules

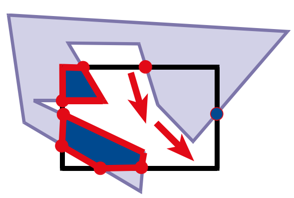
While there is still an unprocessed entering intersection
Walk” polygon/window boundary



MIT EECS 6.837, Durand and Cutler

Walking rules

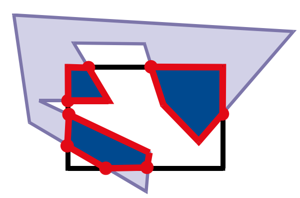
While there is still an unprocessed entering intersection
Walk” polygon/window boundary



MIT EECS 6.837, Durand and Cutler

Walking rules

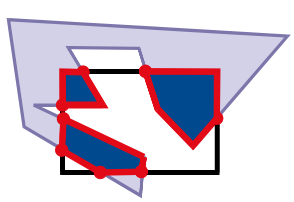
While there is still an unprocessed entering intersection
Walk” polygon/window boundary



MIT EECS 6.837, Durand and Cutler

Weiler-Atherton Clipping

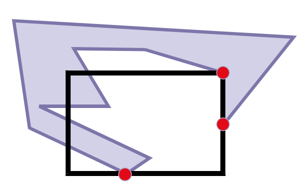
- Importance of good adjacency data structure (here simply list of oriented edges)



MIT EECS 6.837, Durand and Cutler

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!



MIT EECS 6.837, Durand and Cutler

Clipping

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

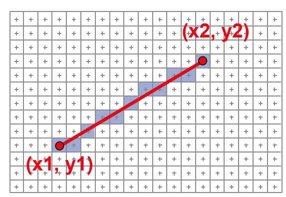
Questions?

Today

- Why Clip?
- Line Clipping
- Polygon clipping
- Line Rasterization

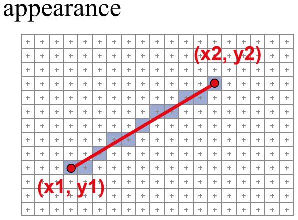
Scan Converting 2D Line Segments

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



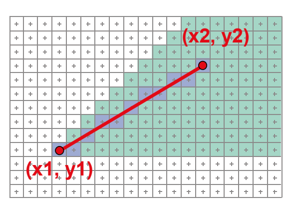
Line Rasterization Requirements

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



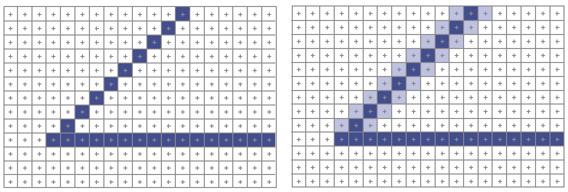
Algorithm Design Choices

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



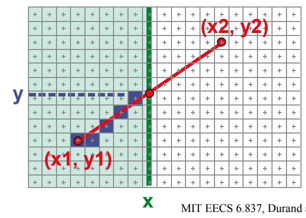
Algorithm Design Choices

- 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)))



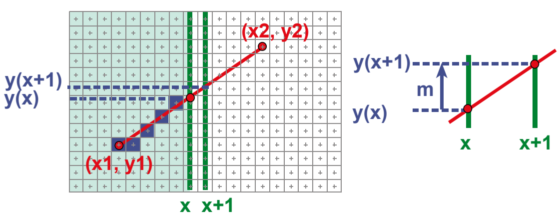
$$y = y1 + \frac{x - x1}{x2 - x1}(y2 - y1)$$
$$= y1 + m(x - x1)$$
$$m = \frac{dy}{dx}$$

Efficiency

- Computing y value is expensive

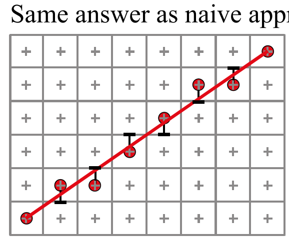
$$y = y1 + m(x - x1)$$

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



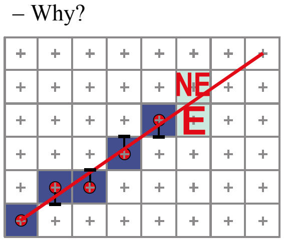
Bresenham's Algorithm (DDA)

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



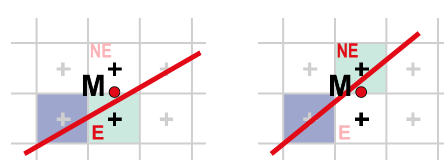
Bresenham's Algorithm (DDA)

- 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

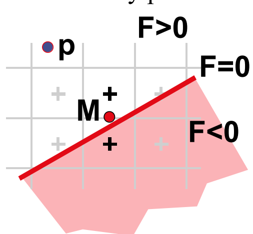


Bresenham Step

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

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

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

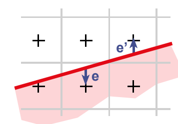


D(p_x,p_y) = vertical distance from point to line

Bresenham's Algorithm (DDA)

- Decision Function:

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



- Initialize:

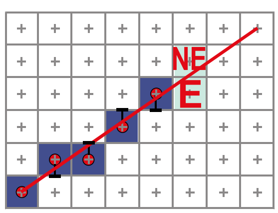
  error term e = –D(x,y)

- 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

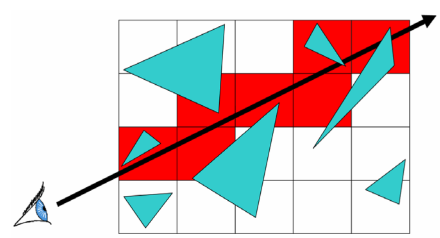
- initialize x, y, e
- for (x = x1; $x \leq$ x2; x++)
  - plot (x,y)
  - update x, y, e



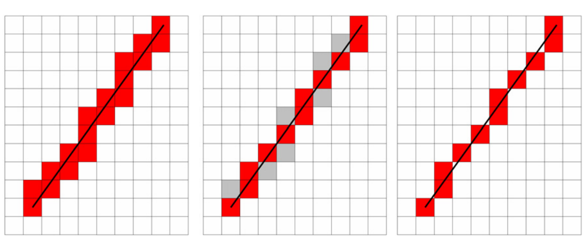
- Generalize to handle all eight octants using symmetry
- Can be modified to use only integer arithmetic

Line Rasterization

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



Grid Marching vs. Line Rasterization



Ray Acceleration:

Must examine every cell the line touches

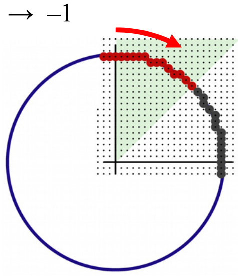
Line Rasterization:

Best discrete approximation of the line

Questions?

Circle Rasterization

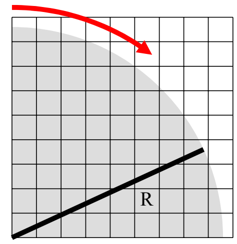
- Generate pixels for 2nd octant only
- Slope progresses from 0 → –1
- Analog of Bresenham Segment Algorithm



MIT EECS 6.837, Durand and Cutler

Circle Rasterization

- Decision Function:
  $D(x, y) = x^2 + y^2 - R^2$
- 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), $e' = e + 1$



MIT EECS 6.837, Durand and Cutler

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

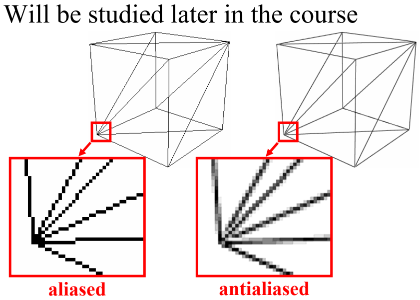
MIT EECS 6.837, Durand and Cutler

Questions?

MIT EECS 6.837, Durand and Cutler

Antialiased Line Rasterization

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



MIT EECS 6.837, Durand and Cutler

High-level concepts for 6.837

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

MIT EECS 6.837, Durand and Cutler

Thursday

Polygon Rasterization
& Visibility