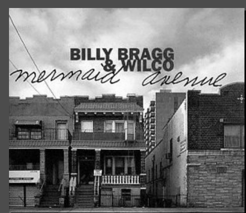
Now Playing:



California Stars
Billy Bragg & Wilco
from *Mermaid Avenue*
Released June 23, 1998

Vertex Processing: Clipping

Rick Skarbez, Instructor
COMP 575
October 2, 2007

Some slides and images courtesy Jeremy Wendt (2005)

Announcements

- Assignment 2 is out today
  - Due next Tuesday by the end of class

Last Time

- Reviewed the OpenGL pipeline
- Discussed classical viewing and presented a taxonomy of different views
- Talked about how projections and viewport transforms are used in OpenGL

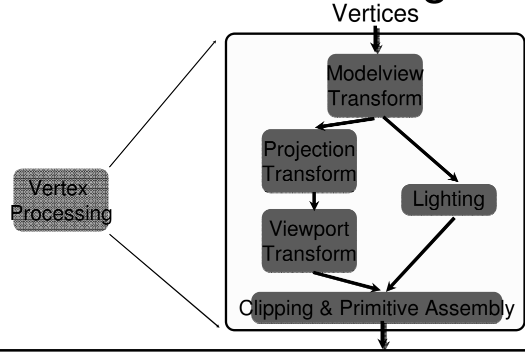
Today

- Discuss clipping
  - Points
  - Lines
  - Polygons

Rendering Pipeline

- OpenGL rendering works like an assembly line
  - Each stage of the pipeline performs a distinct function on the data flowing by
  - Each stage is applied to every vertex to determine its contribution to output pixels

Vertex Processing



Determining What's in the Viewport

- Not all primitives map to inside the viewport
  - Some are entirely outside
    - Need to *cull*
  - Some are partially inside and partially outside
    - Need to *clip*
  - There must be NO DIFFERENCE to the final rendered image

Why Clip?

- Rasterization is very expensive
  - Approximately linear with number of fragments created
  - Math and logic *per pixel*
- If we only rasterize what is actually viewable, we can save a lot
  - A few operations now can save many later

Clipping Primitives

- Different primitives can be handled in different ways
  - Points
  - Lines
  - Polygons

Point Clipping

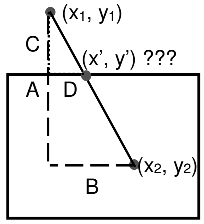
- This one is easy
- How to determine if a point (x, y, z) is in the viewing volume (x_{near}, y_{near}, z_{near}), (x_{far}, y_{far}, z_{far})?
  - Who wants to tell me how?
    - if (($x > x_{far}$ || $x < x_{near}$) ||
      ($y > y_{far}$ || $y < y_{near}$) ||
      ($z > z_{far}$ || $z < z_{near}$))
      cull the point
      else
      keep it

Line Clipping

- What happens when a line passes out of the viewing volume/plane?
  - Part is visible, part is not
- Need to find the entry/exit points, and shorten the line
  - The shortened line is what gets passed to rasterization

Line Clipping Example

- Let's do 2D first
- Clip a line against 1 edge of the viewport
- What do we know?
  - Similar triangles
    - A / B = C / D
      - B = $(x_2 - x_1)$
      - A = $(y_2 - y_1)$
      - C = $(y_1 - y_{max})$
      - ➡ D = BC / A
      - ➡ $(x', y') = (x_1 - D, y_{max})$

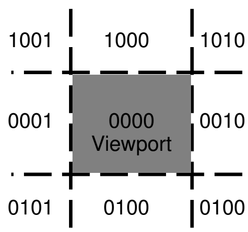


Line Clipping

- The other cases are handled similarly
- The algorithm extends easily to 3D
- The problem?
  - Too expensive! (these numbers are for 2D)
    - 3 floating point subtracts
    - 2 floating point multiplies
    - 1 floating point divide
    - 4 times! (once for each edge)
- We need to do better

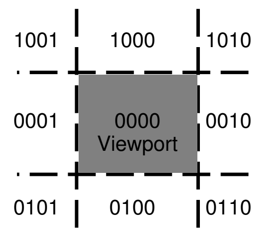
Cohen-Sutherland Line Clipping

- Split plane into 9 regions
- Assign each a 4-bit tag
  - (above, below, right, left)
- Assign each endpoint a tag



Cohen-Sutherland Line Clipping

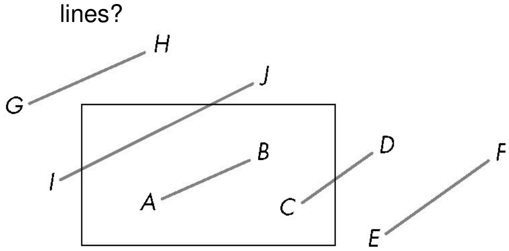
- Algorithm:
  1. if (tag_1 == tag_2 == 0000) accept the line
  2. if ((tag_1 & tag_2) != 0) reject the line
  3. Clip the line against an edge (where both bits are nonzero)
  4. Assign the new vertex a 4-bit value
  5. Return to 1



N.B.: & is the bitwise AND operator

Cohen-Sutherland Example

- What are the vertex codes for these lines?



Cohen-Sutherland Line Clipping

- Lets us eliminate many edge clips early
- Extends easily to 3D
  - 27 regions
  - 6 bits
- Similar triangles still works in 3D
  - Just have to do it for 2 sets of similar triangles

Liang-Barsky Line Clipping

- Consider the parametric definition of a line:
  - $x = x_1 + u\Delta x$
  - $y = y_1 + u\Delta y$
  - $\Delta x = (x_2 - x_1)$, $\Delta y = (y_2 - y_1)$, $0 \le (u, v) \le 1$
- What if we could find the range for u and v in which both x and y are inside the viewport?

Liang-Barsky Line Clipping

- Mathematically, this means
  - $x_{min} \le x_1 + u\Delta x \le x_{max}$
  - $y_{min} \le y_1 + u\Delta y \le y_{max}$
- Rearranging, we get
  - $-u\Delta x \le (x_1 - x_{min})$
  - $u\Delta x \le (x_{max} - x_1)$
  - $-v\Delta y \le (y_1 - y_{min})$
  - $v\Delta y \le (y_{max} - y_1)$
    - In general: $u * p_k \le q_k$

Liang-Barsky Line Clipping

- Cases:
  1. $p_k = 0$
     - Line is parallel to boundaries
       - If for the same k, $q_k < 0$, reject
       - Else, accept
  2. $p_k < 0$
     - Line starts outside this boundary
       - $r_k = q_k / p_k$
       - $u_1 = \max(0, r_k, u_1)$

Liang-Barsky Line Clipping

- Cases: (cont'd)
  3. $p_k > 0$
     - Line starts outside this boundary
       - $r_k = q_k / p_k$
       - $u_2 = \min(1, r_k, u_2)$
  4. If $u_1 > u_2$, the line is completely outside

Liang-Barsky Line Clipping

- Also extends to 3D
  - Just add equations for $z = z_1 + u\Delta z$
  - ➡ 2 more p's and q's

Liang-Barsky Line Clipping

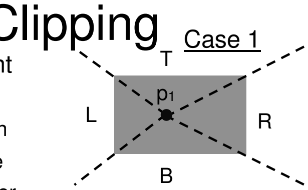
- In most cases, Liang-Barsky is slightly more efficient
  - According to the Hearn-Baker textbook
  - Avoids multiple shortenings of line segments
- However, Cohen-Sutherland is much easier to understand (I think)
  - An important issue if you're actually implementing

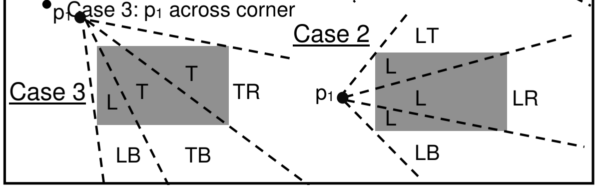
Nicholl-Lee-Nicholl Line Clipping

- This is a theoretically optimal clipping algorithm (at least in 2D)
  - However, it only works well in 2D
- More complicated than the others
- Just do an overview here

Nicholl-Lee-Nicholl Line Clipping

- Partition the region based on the first point (p_1):
  - Case 1: p_1 inside region
  - Case 2: p_1 across edge
  - Case 3: p_1 across corner





Nicholl-Lee-Nicholl Line Clipping

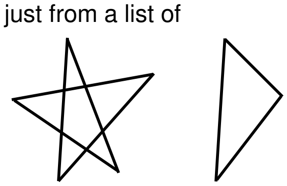
- Can use symmetry to handle all other cases
- "Algorithm" (really just a sketch):
  - Find slopes of the line and the 4 region bounding lines
  - Determine what region p_2 is in
    - If not in a labeled region, discard
    - If in a labeled region, clip against the indicated sides

A Note on Redundancy

- Why am I presenting multiple forms of clipping?
  - Why do you learn multiple sorts?
    - Fastest can be harder to understand / implement
    - Best for the general case may not be for the specific case
      - Bubble sort is really great on mostly sorted lists
    - "History repeats itself"
      - You may need to use a similar algorithm for something else; grab the closest match

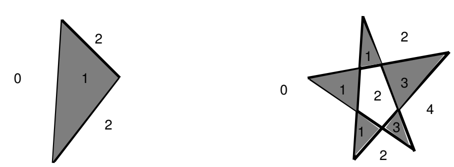
Polygon Inside/Outside

- Polygons have a distinct inside and outside
- How do you tell, just from a list of vertices/edges?
  - Even/odd
  - Winding number



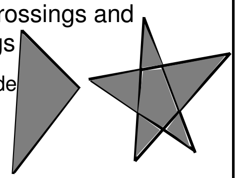
Polygon Inside/Outside: Even / Odd

- Count edge crossings
  - If the number is even, that area is outside
  - If odd, it is inside



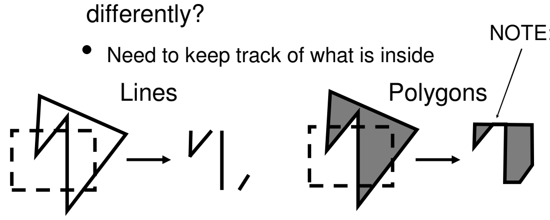
Polygon Inside/Outside: Winding Number

- Each line segment is assigned a direction by walking around the edges in some pre-defined order
  - OpenGL walks counter-clockwise
- Count right->left edge crossings and left->right edge crossings
  - If equal, the point is outside



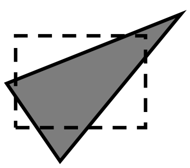
Polygon Clipping

- Polygons are just composed of lines. Why do we need to treat them differently?
  - Need to keep track of what is inside



Polygon Clipping

- Many tricky bits
  - Maintaining inside/outside
  - Introduces variable number of vertices
  - Need to handle screen corners correctly

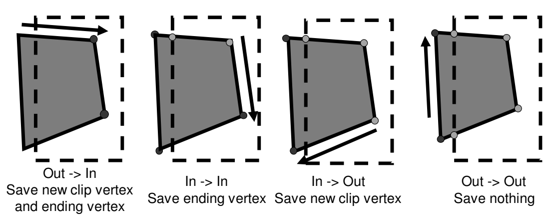


Sutherland-Hodgeman Polygon Clipping

- Simplify via separation
- Clip the entire polygon with one edge
  - Clip the output polygon against the next edge
  - Repeat for all edges
- Extends easily to 3D (6 edges instead of 4)
- Can create intermediate vertices that get thrown out later

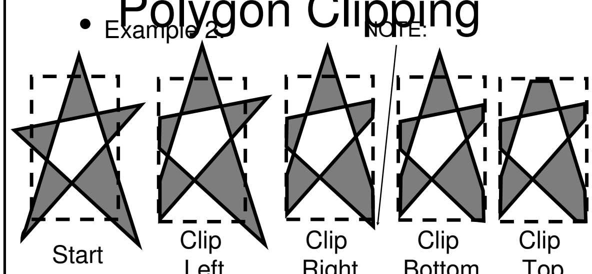
Sutherland-Hodgeman Polygon Clipping

- Example 1:



Sutherland-Hodgeman Polygon Clipping

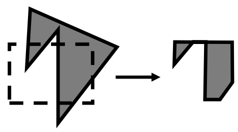
- Example 2:



Weiler-Atherton Polygon Clipping

- When using Sutherland-Hodgeman, concavities can end up linked

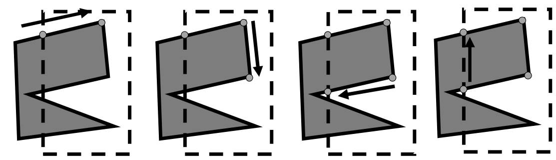
Remember this?



- A different clipping algorithm, the Weiler-Atherton algorithm, creates separate polygons

Weiler-Atherton Polygon Clipping

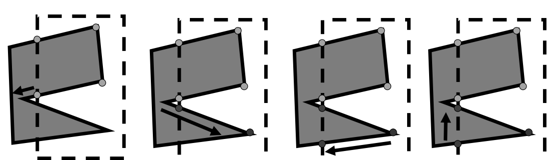
- Example:



Out -> In Add clip vertex Add end vertex	In -> In Add end vertex	In -> Out Add clip vertex Cache old direction	Follow clip edge until (a) new crossing found (b) reach vertex already added

Weiler-Atherton Polygon Clipping

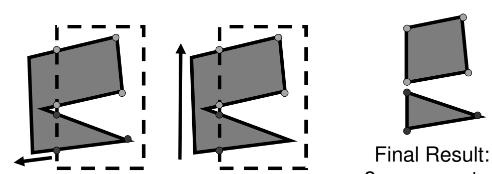
- Example (cont'd):



Continue from cached vertex and direction	Out -> In Add clip vertex Add end vertex	In -> Out Add clip vertex Cache old direction	Follow clip edge until (a) new crossing found (b) reach vertex already added

Weiler-Atherton Polygon Clipping

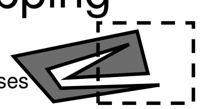
- Example (cont'd):

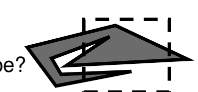


Continue from cached vertex and direction	Nothing added Finished	Final Result: 2 unconnected polygons

Weiler-Atherton Polygon Clipping

- Difficulties:
  - What if the polygon recrosses an edge?





  - How big should your cache be?
  - Geometry step must be able to create new polygons
    - Not 1 in, 1 out

Done with Clipping

- Point Clipping (really just culling)
  - Easy, just do inequalities
- Line Clipping
  - Cohen-Sutherland
  - Liang-Barsky
  - Nicholl-Lee-Nicholl
- Polygon Clipping
  - Sutherland-Hodgeman
  - Weiler-Atherton

Any Questions?

Next Time

- Moving on down the pipeline
- Rasterization
  - Line drawing