Clipping

- Don’t want to draw stuff that is not in screen area
  - Waste of time
  - Bogus memory access
  - Looks ugly
- Clipping removes unwanted stuff
- Can also clip for other reasons
  - Cut away views

Basic Line Clipping

Cohen-Sutherland Line Clip

- Clip line against each edge of clip region in turn
  - If both endpoints outside, discard line and stop
  - If both endpoints in, continue to next edge (or finish)
  - If one in, one out, chop line at crossing pt and continue
- Works in both 2D and 3D for convex clipping regions

Cohen-Sutherland Line Clip

- Trivial reject and accept (speed things up…)

Trivial Keep  Trivial Reject  NOT Trivial Reject
Cohen-Sutherland Line Clip

- Out codes

Liang-Barsky Clipping

- Parametric clipping - view line in parametric form and reason about the parameter values
- More efficient, as not computing the coordinate values at irrelevant vertices
- Works for rectilinear clip regions in 2D or 3D
- Clipping conditions on parameter: Line is inside clip region for values of $t$ such that (for 2D):

\[ x_{\text{min}} \leq x + \Delta x \leq x_{\text{max}} \]
\[ y_{\text{min}} \leq y + \Delta y \leq y_{\text{max}} \]

Liang-Barsky Clipping

- Infinite line intersects clip region edges at:

\[ t_k = \frac{q_k}{p_k} \quad \text{where} \]
\[ p_1 = -\Delta x \quad q_1 = x_i - x_{\text{max}} \]
\[ p_2 = \Delta x \quad q_2 = x_{\text{max}} - x_i \]
\[ p_3 = -\Delta y \quad q_3 = y_i - y_{\text{max}} \]
\[ p_4 = \Delta y \quad q_4 = y_{\text{max}} - y_i \]

Note: Left edge is 1, right edge is 2, top edge is 3, bottom is 4

Liang-Barsky Clipping

- When $p_k < 0$, as $t$ increases line goes from outside to inside - entering
- When $p_k > 0$, line goes from inside to outside - leaving
- When $p_k = 0$, line is parallel to an edge (clipping is easy)
- If there is a segment of the line inside the clip region, sequence of infinite line intersections must go: enter, enter, leave, leave

Liang-Barsky Clipping

- Compute entering $t$ values, which are $q_k/p_k$ for each $p_k < 0$
- Compute leaving $t$ values, which are $q_k/p_k$ for each $p_k > 0$
- Parameter value for small $t$ end of line is $t_{\text{small}} = \max(0, \text{entering } t)$
- Parameter value for large $t$ end of line is: $t_{\text{large}} = \min(1, \text{leaving } t)$
- If $t_{\text{small}} \leq t_{\text{large}}$, there is a line segment - compute endpoints by substituting $t$ values
- Improvement (and actual Liang-Barsky):
  - Compute $t$’s for each edge in turn (some rejects occur earlier like this)
Sutherland-Hodgman Polygon Clip

- Clip the polygon against each edge of the clip region in turn
  - Clip polygon each time to line containing edge
  - Only works for convex clip regions (Why?)

General Clipping (Weiler Algorithm)

- To clip a polygon to a line/plane:
  - Consider the polygon as a list of vertices
  - One side of the line/plane is considered inside the clip region, the other side is outside
  - We are going to rewrite the polygon one vertex at a time – the rewritten polygon will be the polygon clipped to the line/plane
  - Check start vertex: if “inside”, emit it, otherwise ignore it
  - Continue processing vertices as follows...

Inside-Outside Testing

Rearranging pointers makes it possible to enumerate all components of the intersection
Inside/Outside in Screen Space

Finding Intersection Points

Backface Culling

Z-Buffer (a.k.a. Depth Buffer)
- Use additional buffer to store a Z value at each pixel
- When filling pixel write Z value to buffer also
- Don’t fill if new Z value is larger than old
- Quantization and aliasing problems sometimes occur
- Very commonly used
- Standard feature in hardware
- Interpolating Z values over polygons

Z-Buffer and Transparency
- Must render in order back to front

A-Buffer
- Store list of polygons at each pixel (fragments)
- Draw opaque stuff first, only keep closest frag
- Second pass for transparent stuff
- Allows antialiasing...
- Not good for hardware implementation

Talk about compositing later…
Scan Line Algorithm

*Assume polygons don’t intersect each other*
- As scanning keep list of active edges
- Decide who’s on top when crossing edges

Advantages:
- Simple
- Potentially fewer quantization errors (more bits available for depth)
- Don’t over-render (each pixel only drawn once)
- Filter anti-aliasing can be made to work (have information about all polygons at each pixel)

Disadvantages:
- Invisible polygons clog AEL, ET
- Non-intersection criteria may be hard to meet

Painters Algorithm

- Sort based on depth and draw back to front

- How do we sort quickly?
- What about intersecting polygons?

BSP-Trees (Object Precision)

- Construct a binary space partition tree
  - Depth-first traversal gives a rendering order (variant)
  - Tree splits 3D world with planes
    - The world is broken into convex cells
    - Each cell is the intersection of all the half-spaces of splitting planes on tree path to the cell
  - Also used to model the shape of objects, and in other visibility algorithms
    - BSP visibility in games does not necessarily refer to this algorithm

BSP-Tree Example

Building BSP-Trees

- Choose polygon (arbitrary)
- Split its cell using plane on which polygon lies
  - May have to chop polygons in two (Clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - Optimal means minimum number of polygon fragments in a balanced tree
Building Example

- We will build a BSP tree, in 2D, for a 3 room building
  - Ignoring doors
- Splitting edge order is shown
  - “Back” side of edge is side with the number

Building Example

Building Example (Done)

BSP-Tree Rendering

- Observation: Things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint
- Rendering algorithm is recursive descent of the BSP Tree
- At each node (for back to front rendering):
  - Recurse down the side of the sub-tree that does not contain the viewpoint
  - Test viewpoint against the split plane to decide which tree
  - Draw the polygon in the splitting plane
  - Paint over whatever has already been drawn
  - Recurse down the side of the tree containing the viewpoint
BSP-Tree Rendering Example

BSP-Tree Rendering

- Advantages:
  - One tree works for any viewing point
  - Filter anti-aliasing and transparency work
    - Have back to front ordering for compositing
    - Can also render front to back, and avoid drawing back polygons that cannot contribute to the view
  - User two trees – an extra one that subdivides the window
- Disadvantages:
  - Can be many small pieces of polygon

Cells and Portals

- Assume the world can be broken into cells
  - Simple shapes
    - Rooms in a building, for instance
- Define portals to be the transparent boundaries between cells
  - Doorways between rooms, windows, etc
- In a world like this, can determine exactly which parts of which rooms are visible
  - Then render visible rooms plus contents
- An Exact Visibility Algorithm — sometimes

Cell and Portal Visibility

- Start in the cell containing the viewer, with the full viewing frustum
- Render the walls of that room and its contents
- Recursively clip the viewing frustum to each portal out of the cell, and call the algorithm on the cell beyond the portal

Cell-Portal Example

Cell-Portal Example
Acknowledgments

- Some of these slide contents were based on original materials from
  - Prof. David Forsyth of U.C. Berkeley
  - Prof. Stephen Chenney of U.W. Madison