Viewing Transformation

Clipping

2-D Viewing Transformation
2-D Viewing Transformation

- Convert from Window Coordinates to Viewport Coordinates
- \((x_w, y_w) \rightarrow (x_v, y_v)\)
- Maps a world coordinate window to a screen coordinate viewport
- Window defined by: \((x_{w1}, y_{w1}), (x_{w2}, y_{w2})\)
- Viewport defined by: \((x_{v1}, y_{v1}), (x_{v2}, y_{v2})\)
- Basic idea is to maintain proportionality

Window to Viewport Transformation

\[
\frac{x_v-x_v1}{x_{v2}-x_{v1}} = \frac{x_w-x_w1}{x_{w2}-x_{w1}}
\]

\[
\frac{y_v-y_v1}{y_{v2}-y_{v1}} = \frac{y_w-y_w1}{y_{w2}-y_{w1}}
\]

where:
- \(W_w = x_{w2} - x_{w1}\) (window width)
- \(W_v = x_{v2} - x_{v1}\) (viewport width)
- \(H_w = y_{w2} - y_{w1}\) (window height)
- \(H_v = y_{v2} - y_{v1}\) (viewport height)
Viewing Transformation in Windows: Mapping Modes

Windows Viewing Transformation: Mapping Modes

- Create logical coordinate system
  - Define direction of axes
  - Define units
  - Can also move the origin
- Windows maps output to real device
  - e.g., plot at 100,100 "logical millimeters"
  - Windows figures out where on screen
  - Not exact, but close
- It’s Windows way of implementing the viewing transformation
### Windows Mapping Modes

<table>
<thead>
<tr>
<th>MAPPING MODE</th>
<th>LOGICAL UNIT</th>
<th>X-AXIS</th>
<th>Y_AXIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM_TEXT</td>
<td>Pixel</td>
<td>Right</td>
<td>Down</td>
</tr>
<tr>
<td>MM_HIENGLISH</td>
<td>.001 inch</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_LOENGLISH</td>
<td>.01 inch</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_HIMETRIC</td>
<td>.01 mm</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_LOMETRIC</td>
<td>.1 mm</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_TWIPS</td>
<td>1/20 point=1/1440”</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_ISOTROPIC</td>
<td>Arbitrary (x=y)</td>
<td>Selectable</td>
<td></td>
</tr>
<tr>
<td>MM_ANISOTROPIC</td>
<td>Arbitrary (x!=y)</td>
<td>Selectable</td>
<td></td>
</tr>
</tbody>
</table>

### Changing the Mapping Mode

- `pDC->SetMapMode(MAP_MODE);`
- Maps logical coordinates to device coordinates
  - Device Coordinate (physical)
    - units: pixels
    - +x: right, +y: down
  - Converts logical ("window") to device ("viewport") coordinates as follows
    - \( xV = \frac{xWExt}{xWExt} \times (xW - xWOrg) + xVOrg \)
    - \( yV = \frac{yVExt}{yVExt} \times (yW - yWOrg) + yVOrg \)
- \((xWOrg, yWOrg)\) and \((xVOrg, yVOrg)\) are the origins of the window and viewport
- Both are (0,0) in the default device context
Moving Origins

- `pDC->SetWindowOrg(x,y);` // logical units
  - For x,y positive, think of this as moving the upper left-hand corner of the physical device viewport (screen) up and right by (x,y) logical units
- `pDC->SetViewportOrg(x,y);` // device units--pixels
  - For x,y positive, think of this as moving the lower left-hand corner of the logical window down and right by (x,y) device units
- Both move the coordinate system origin to (x,y), but units of x,y are different

Variable Unit Mapping Modes

- Coordinate axes can have any size/orientation
- `MM_ISOTROPIC`-- x & y units must be same size
- `MM_ANISOTROPIC`-- different x and y units
- Set the X and Y scaling factors with:
  - `pDC->SetWindowExt (xWExt, yWExt);`
  - `pDC->SetViewportExt (xVExt, yVExt);`
- X scaling factor in going from Logical Coordinates to Device Coordinates = `xVExt/xWExt`
- Y scaling factor = `yVExt/yWExt`
Example 1

Create coordinate system where each logical unit is two pixels:
– twice the default device unit coordinates
  pDC->SetMapMode(MM_ISOTROPIC);
  pDC->SetWindowExt(1, 1);
  pDC->SetViewportExt(2, 2);

Example 2

Create coordinate system with y-axis up, each y-unit = 1/4 pixel; x-axis unchanged:
  pDC->SetMapMode(MM_ANISOTROPIC);
  pDC->SetWindowExt(1, -4);
  pDC->SetViewportExt(1, 1);
Example 3

Create coord system where client area is always 1000 units high & wide, y-axis up:

```cpp
CSize size;
size = pDC->GetWindowExt(); // get client area size
// returns size in default device units--here pixels
pDC->SetMapMode(MM_ANISOTROPIC);
pDC->SetWindowExt(1000, -1000);
pDC->SetViewportExt(size.cx, size.cy);
```

Now (1000,1000) will always be at upper right edge of client area

OpenGL Viewing Transformation

OpenGL designed for 3D graphics

Must project onto 2D window

Also do window to viewport transformation

– with clipping

For 2D graphics, use an orthographic projection

– gluOrtho2D(xmin,xmax,ymin,ymax)

• Equivalent to taking z=0 & setting a “window” with clipping boundaries: xmin<=x<=xmax,
  ymin<=y<=ymax  -- logical units used

  – Will be mapped to entire client area of physical window

  – Client area determined by:
    – glutInitWindowSize(width,height)
    – Device units used
OpenGL Viewport

- gluOrtho2d(left, right, bottom, top) and glutInitWindowSize(w, h) map the “window” to the entire w X h client area
- glViewport(x, y, w, h) maps the “window” to the specified viewport within the client area
  - Device units used

Clipping
Clipping

- Elimination of parts of scene outside a window or viewport
- Clipping with respect to a window (Given: xwmin, ywmin, xwmax, ywmax)
  - Clip at this level $\Rightarrow$ fewer points go through viewing transformation
- Clipping with respect to a viewport (Given: xvmin, yvmin, xvmax, yvmax)

Clipping

- Points
- Lines
  - Cohen-Sutherland Line Clipper
- Polygons
  - Sutherland-Hodgeman Polygon Clipper
  - Weiler-Atherton Polygon Clipper
- Other Curves
- Text
Point Clipping

Given:
- point \((x,y)\)
- clipping rectangle (window or viewport) \((x_{\text{min}},y_{\text{min}},x_{\text{max}},y_{\text{max}})\)

Point test:
if \(((x \leq x_{\text{max}}) \&\& (x \geq x_{\text{min}}) \&\& (y \leq y_{\text{max}}) \&\& (y \geq y_{\text{min}}))\)
the point \(x,y\) lies inside the clip area
- so keep it!

Line Clipping

Could apply point test to all points on the line
- Too much work
Need a simple test involving the line's endpoint coordinates
Cohen-Sutherland Line Clipper

Observation-- All lines fall into one of three categories
1. Both endpoints inside clip rectangle
   • (Trivially accept entire line)
2. Both endpoints outside clip rectangle on the same side of one of its borders
   • (Trivially reject entire line)
3. Neither 1 nor 2
   • (Chop off part of line outside one of borders and repeat)

Region Code

A tool in assigning lines to Category 1 or 2
4-bit region code number assigned to an endpoint (x,y)
Any set bit means endpoint is outside of one of the 4 borders of the clip rectangle
Each bit position corresponds to a different border
Region Code RC = LRBT

- L=left (if x<xmin, L=1, else L=0)
- R=Right (if x>xmax, R=1, else R=0)
- B=Bottom (if y<ymin, B=1, else B=0)
- T=Top (if y>ymax, T=1, else B=0)

The Region Code Divides the entire x-y plane 9 regions

Region Codes (LRBT)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>0001</td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0010</td>
<td>0110</td>
</tr>
</tbody>
</table>
Category 1 Lines

Assume region codes for the line’s endpoints are RC1 and RC2

Take Boolean OR of two region codes

if (RC1 | RC2 == 0)
    both RCs are 0000
    both endpoints are inside
    so it’s Category 1 (trivial accept)

Category 2 Lines

Both endpoints are outside same border
   – (Category 2 line)

Then both region codes will have the same bit set in one of the four bit positions
   – Boolean AND will give a non-zero result:
      if (RC1 & RC2 != 0)
         • both endpoints are outside same border
         • so it’s Category 2 (trivial reject)
Category 3 Lines

- Want to chop off outside part of line
- May have both endpoints (P1 & P2) outside different borders of clip region
  - So it’s not important which end is chopped off first
- But if one endpoint’s in and other’s out:
  - Want to chop off the outside end
  - So Arrange things so P1 is the outside point
    • (swap P1 & P2 if necessary)

How to do the Chopping

- Want to determine the new endpoint
- Endpoint coordinates (x1,y1), (x2,y2) are known
- Slope m can be computed from them
- So \( y = m(x-x2) + y2 \) (point slope form)
- Or \( x = (y-y2)/m + x2 \)
- Look at P1’s region code (RC1)
- Four possible cases:
If RC1 == 1xxx (P1 to left of xmin)

- New endpoint should be on the left boundary:
  - $x_1 \leftarrow \text{xmin}$
  - $y_1 \leftarrow m(x_{\text{min}}-x_2) + y_2$
  - Reset RC’s L bit

If RC1 == x1xx (P1 right of xmax)

- New endpoint should be on the right boundary:
  - $x_1 \leftarrow \text{xmax}$
  - $y_1 \leftarrow m(x_{\text{max}}-x_2) + y_2$
  - Reset RC’s R bit
If RC1 == xx1x (P1 below ymin)

- New endpoint should be on the bottom boundary:
  - \( y_1 \leftarrow y_{\text{min}} \)
  - \( x_1 \leftarrow \frac{(y_{\text{min}}-y_2)}{m} + x_2 \)
  - Reset RC’s B bit

If RC == xxx1 (P1 above ymax)

- New endpoint should be on the top boundary:
  - \( y_1 \leftarrow y_{\text{max}} \)
  - \( x_1 \leftarrow \frac{(y_{\text{max}}-y_2)}{m} + y_2 \)
  - Reset RC’s T bit
Horizontal and vertical lines are special cases

- Horizontal:
  - y doesn't change and \( x = \text{xboundary} \)
- Vertical:
  - x doesn't change and \( y = \text{yboundary} \)

The C-S Line Clipping Algorithm

**Input:**
- Original endpoints \( (x_1,y_1,x_2,y_2) \)
- Clip region boundaries \( (x_{\text{min}},y_{\text{min}},x_{\text{max}},y_{\text{max}}) \)

**Output:**
- Accept Code (AC)
  - AC==TRUE ==> some part of line was inside
  - AC==FALSE ==> no part of line was inside
- Clipped Line endpoints \( (x_1,y_1,x_2,y_2) \)
  - only if AC==TRUE
C-S Algorithm Pseudo-code:

CS_LineClip(xmin, ymin, xmax, ymax, x1, y1, x2, y2, AC)

done = FALSE

While (!done)
    Calculate endpoint codes rc1, rc2
    If ((rc1 | rc2) == 0) // Category 1
        done = TRUE
        AC = TRUE
    Else
        If ((rc1 & rc2) != 0) // Category 2
            done = TRUE
            AC = FALSE
        Else
            If (P1 is inside)
                Swap (x1, y1), (x2, y2); and rc1, rc2
    If (L-bit of rc1 is set) // 1xxx
        x1 = xmin
        y1 = m*(xmin-x2) + y2
    Else
        If (R-bit of rc1 is set) // x1xx
            x1 = xmax
            y1 = m*(xmax-x2) + y2
        Else
            If (B-bit of rc1 is set) // xx1x
                y1 = ymin
                x1 = (ymin-y2)/m + x2
            Else // xxx1
                y1 = ymax
                x1 = (ymax-y2)/m + x2
Cohen-Sutherland Clipping Example

Step 1
RC1=1010, RC2=0101

Step 2
RC1=0010, RC2=0101

Step 3
RC1=0101, RC2=0000

Step 4
RC1=0001, RC2=0000

Step 5: Accept DONE!!

Polygon Clipping
Polygon Clipping

- Clip a polygon to a rectangular clip area

Input
- Ordered list of polygon vertices (nin, vin[ ])
- Clip rectangle boundary coordinates (xmin, ymin, xmax, ymax).

Output:
- An ordered list of clipped polygon vertices (nout, vout[ ]).
- vin[ ] and vout[ ] could be arrays of POINTs

Approaches to Polygon Clipping

- Use a line clipper on each polygon edge???
- But we usually won’t get back a polygon
  - Parts of the clip rectangle will be edges of the clipped polygon that line clipper won’t get
- Really need new list of edges (or vertices)
Sutherland-Hodgeman Polygon Clipper

Approach:
- Clip all polygon edges with respect to each clipping boundary
- Do four passes; on each pass:
  - Traverse current polygon and clip with respect to one of the four boundaries
  - Assemble output polygon edges as you go

Vin[ ] --> Clip Left --> vtemp1[ ] --> Clip Right --> vtemp2[ ] --> Clip Bottom --> vtemp3[ ] --> Clip Top --> Vout[ ]

On any polygon traversal the clip boundary divides plane into "in" side and "out" side

For any given edge (vertices i and i+1), during traversal, there are four possibilities:
- (Assume vertex i has already been processed)

<table>
<thead>
<tr>
<th>VERTEX i</th>
<th>VERTEX i+1</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>in</td>
<td>Add Vertex i+1 to output list</td>
</tr>
<tr>
<td>out</td>
<td>out</td>
<td>Add no vertex to output list</td>
</tr>
<tr>
<td>in</td>
<td>out</td>
<td>Add intersection point with edge to output list</td>
</tr>
<tr>
<td>out</td>
<td>in</td>
<td>Add intersection point with edge and vertex i+1 to output list</td>
</tr>
</tbody>
</table>
Sample Traversal

<table>
<thead>
<tr>
<th>Traversal</th>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>in-in</td>
<td>Add point b</td>
</tr>
<tr>
<td>2</td>
<td>in-out</td>
<td>Add intersection point x</td>
</tr>
<tr>
<td>3</td>
<td>out-out</td>
<td>Add nothing</td>
</tr>
<tr>
<td>4</td>
<td>out-in</td>
<td>Add intersection point y and point a</td>
</tr>
</tbody>
</table>

Function `sh_clip()`
- Will clip an input polygon \( (n_i, v_i[]) \)
- With respect to a given boundary \( (bndry) \)
- Generating an output polygon \( (n_o, v_o[]) \)

Enumerate the boundaries as:
- LEFT, RIGHT, BOTTOM, and TOP

```
sh_clip(n_i, v_i[], n_o, v_o[], xmin, ymin, xmax, ymax, bndry):
```
- `v_i[]` and `v_o[]`: could be arrays of POINTs
- `n_i`, `n_o`: number of points in each array
- `xmin`, `ymin`, `xmax`, `ymax`: clip region boundaries
Using `sh_clip()` to clip a polygon

Make four calls to `sh_clip()`:

- `sh_clip(nin, vin[], ntemp1, vtemp1[], xmin, ymin, xmax, ymax, LEFT);`
- `sh_clip(ntemp1, vtemp1[], ntemp2, vtemp2[], xmin, ymin, xmax, ymax, RIGHT);`
- `sh_clip(ntemp2, vtemp2[], ntemp3, vtemp3[], xmin, ymin, xmax, ymax, BOTTOM);`
- `sh_clip(ntemp3, vtemp3[], nout, vout[], xmin, ymin, xmax, ymax, TOP);`

Three Helper Functions

- `BOOL inside(V, xmin, ymin, xmax, ymax, Bndry)`
  - Returns `TRUE` if vertex point `V` is on the "in" side of boundary `Bndry`
- `intersect(V1, V2, xmin, ymin, xmax, ymax, Bndry, Vnew)`
  - Computes intersection point of edge whose endpoints are `V1` and `V2` with boundary `Bndry`
  - Returns the resulting point in `Vnew`
- `output(V, n, vout[])`
  - Adds vertex point `V` to the polygon `(n, v[])`
    - `n` will be incremented by 1
    - vertex `V` added to end of polygon's vertex list `v[]`
sh_clip (ni, vi[], no, vo[], bndry)
no = 0                         // output list begins empty
First_V = vi[0]                // first vertex (i)
For (j=0 to ni-1)              // traverse polygon
    Second_V = v[(j+1) % ni]   // second vertex (i+1)
    If (inside(First_V, bndry)
        If (inside(Second_V, bndry) // "in-in" case
            output(Second_V, no, vo)
        Else                         // "in-out" case
            intersect(First_V, Second_V, bndry, Vtemp)
            output (Vtemp, no, vo)
        Else
            If (inside(Second_V, bndry) // "out-in" case
                intersect(First_V, Second_V, bndry, Vtemp)
                output (Vtemp, no, vo)
                output(Second_V, no, vo)   // no "out-out" case
            First_V = Second_V             // prepare for next edge
Example of S-H Clipping
Sutherland-Hodgeman Problems

- Works fine with convex polygons
- But some concave polygons problematic
  - Extraneous edges along a clip boundary may be generated as part of the output polygon
  - Could cause problems with polygon filling

Solutions to S-H Problems

- Add a postprocessing step
  - Check output vertex list for multiple (>2) vertex points along any clip boundary
  - Correctly join pairs of vertices

Output Polygon: 0, 1, 2, 3, 4, 5
Extraneous Edge: 2-3

Polygon: 0, 1, 2, 3, 4, 5
0, 2, 3, 5 are vertices along the left boundary
So break into two polygons: 0, 1, 2 and 3, 4, 5
Other Solutions

- Add a preprocessing step
  - Split concave polygon into convex polygons
- Or use a more general clipping algorithm
  - For example, the Weiler-Atherton polygon clipper

Splitting Concave Polygons

- Split into convex polygons
- Use edge vector cross products
Vector Product of Two Vectors

\[ V = A \times B \]
\[ |V| = |A| \cdot |B| \cdot \sin(\theta) \]

Direction: RH Rule

In terms of components

\[ \begin{vmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} \]

i, j, k: unit vectors in x, y, z directions

Splitting Concave Polygons

Process edges in clockwise order
Form successive edge vectors
Compute vector cross product between successive edge vectors
If all cross products are not negative
   Polygon is concave
   Split it along line of first edge vector in the cross-product pair:
      Compute intersections of this line with other edges
      This splits polygon into two pieces
Repeat this until no other edge cross products are positive
Splitting Concave Polygons

1. Define a sequence of three consecutive vertices to be a new polygon (triangle)
2. Delete middle vertex from original vertex list
3. Continue to form triangles until original polygon has only three vertices

Splitting Convex Polygon into Triangles

Often convenient since triangles are the simplest polygon
1. Define a sequence of three consecutive vertices to be a new polygon (triangle)
2. Delete middle vertex from original vertex list
3. Continue to form triangles until original polygon has only three vertices
Weiler-Atherton Polygon Clipper

- Clips a "Subject Polygon" to a "Clip Polygon"
- Both polygons can be of any shape
- Result: one or more output polygons that lie entirely inside the clip polygon
- Basic idea:
  - Follow a path that may be a subject polygon edge or a clip polygon boundary

The Weiler-Atherton Algorithm

1. Set up vertex lists for subject and clip polygons
   Ordering: as you move down each list, inside of polygon is always on the right side (clockwise)
2. Compute all intersection points between subject polygon and clip polygon edges
   Insert them into each polygon's list
   Mark as intersection points
   Mark “out-in” intersection points
   (subject polygon edge moving from outside to inside of clip polygon edge)
Intersection Points & out-in Marking (General)

- If clip polygon is a rectangle:
  - Use point in/out test
  - e.g., for intersection with left boundary:
    \[ x < x_{\text{min}} \text{ means outside, } x \geq x_{\text{min}} \text{ means inside} \]

- Intersections also easy
  - Use Cohen-Sutherland ideas
    - e.g., for intersection with left boundary:
      \[
      x = x_{\text{min}} \\
      y = m^{*}(x_{\text{min}} - x_{1}) + y_{1}
      \]

Intersection Points and Out-In Marking (Simple)

- If clip polygon is a rectangle:
  - Use point in/out test
    - e.g., for intersection with left boundary:
      \[ x < x_{\text{min}} \text{ means outside, } x \geq x_{\text{min}} \text{ means inside} \]

- Intersections also easy
  - Use Cohen-Sutherland ideas
    - e.g., for intersection with left boundary:
      \[
      x = x_{\text{min}} \\
      y = m^{*}(x_{\text{min}} - x_{1}) + y_{1}
      \]
Weiler-Atherton Algorithm, continued

3. Do until all intersection points have been visited:
   - Traverse subject polygon list until a non-visited out-in intersection point is found;
   - Output it to new output polygon list
   - Make subject polygon list be the active list
   - Do until a vertex is revisited:
     • Get next vertex from active list & output
     • If vertex is an intersection point,
       – make the other list active
   - End current output polygon list

1st Iteration:
Subject: a→1'→2 b 3→4 c 5' d e 6 f 1 revisited so stop out poly
Clip: A B 2→3' C D 6 5'→4→1'
Output: 1 2 3 4

2nd Iteration:
Subject: a 1'→2 b 3→4 c 5'→d e 6 f 5 revisited so stop out poly
Clip: A B 2→3' C D 6→5' 4→1' 5 d e 6
Output: All intersection points visited ⇒ Done!
Clipping Other Curves

- Must compute intersection points between curve and clip boundaries
- In general solve nonlinear equations
- Many times approximation methods must be used
- Time consuming

Clipping Text

- Use successively more expensive tests
- 1. Clip string
   - Embed string in rectangle
   - Clip rectangle (4 point tests)
     - entirely in ==> keep string
     - entirely out ==> reject string
     - neither ==> next test
2. Clip each Character
   Embed character in rectangle
   Clip rectangle (4 point tests)
   • entirely in ==> keep character
   • entirely out==>reject character
   • neither==>next test
3. Two possibilities for Character Clipping
   – Bitmapped: look at each pixel
   – Stroked: Apply line clipper to each stroke