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

Viewing Transformation Matrix

\[
\begin{bmatrix}
  x_v \\
  y_v \\
  1
\end{bmatrix} =
\begin{bmatrix}
  x_w \\
  y_w \\
  1
\end{bmatrix} P_w
\]

\[
P_w = \begin{bmatrix}
  x_v & y_v & 1
\end{bmatrix}
\]

\[
Tv = \begin{bmatrix}
  C1 & 0 & C2 \\
  0 & C3 & C4 \\
  0 & 0 & 1
\end{bmatrix}
\]

- \(C1 = \frac{W_v}{W_w}\)
- \(C2 = x_{v1} - \left(\frac{W_v}{W_w}\right) x_{w1}\)
- \(C3 = \frac{H_v}{H_w}\)
- \(C4 = y_{v1} - \left(\frac{H_v}{H_w}\right) y_{w1}\)

Be Careful with y-axis down

- y equation will be different!
Viewing Transformation in Windows: Mapping Modes

- Windows Implementation of the Viewing Transformation
- See: CS-360, CS-460/560 Notes and Sample Programs:
  - http://www.cs.binghamton.edu/~reckert/360/mapmode1_cpp.htm

Windows Mapping Modes

- Mapping Mode: a GDI Attribute
- Default DC coordinates: "Device Units"
  - For video screen: pixels
    - Origin at upper left corner of screen
  - For a printer: "dots"
    - Origin at upper left corner of printed page
- Problem—not all pixels/dots same size
  - So screen image can become a "postage stamp" on printer

Windows Mapping Modes

- Create logical system of units
  - A logical coordinate system
- 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

<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&quot;</td>
<td>Right</td>
<td>Up</td>
</tr>
<tr>
<td>MM_ISOTROPIC</td>
<td>Arbitrary (x==y) Selectable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM_ANISOTROPIC</td>
<td>Arbitrary (x!=y) Selectable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changing the Mapping Mode

- pDC->SetMapMode(MAP_MODE);
- Maps logical to device coordinates
  - DC (physical) units: pixels, +x: right, +y: down
  - Converts logical ("window") to device ("viewport") coordinates as follows
    \[ x_{V} = (x_{W} - x_{WOrg}) \times \frac{x_{WExt}}{x_{WExt}} + x_{VOrg} \]
    \[ y_{V} = (y_{W} - y_{WOrg}) \times \frac{y_{WExt}}{y_{WExt}} + y_{VOrg} \]
- (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); // log. units
  - For x,y positive, think of this as moving the upper left-hand corner of the screen/paper 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 coordinate system where client area is always 1000 units high & wide, y-axis up:
  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 orthogonal projection
  - gluOrtho2D(xmin, xmax, ymin, ymax)
    - Equivalent to taking z=0 & setting a "window" with clipping boundaries: xmin<=x<=xmax, ymin<=y<=ymax
    - Will be mapped to entire client area of physical window
- Since projection transformations are done with matrices, must first set the matrix mode and initialize the matrix:
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity();
  - 2D: gluOrtho2D(xmin, xmax, ymin, ymax);
  - 3D: glOrtho(xmin, xmax, ymin, ymax, zmin, zmax);

Clipping

- Elimination of parts of scene outside a window or viewport
- Clipping with respect to a window
  (Given: xmin, ymin, xmax, ymax)
  - Clip at this level ==> fewer points go through viewing transformation
- Clipping with respect to a viewport
  (Given: xmin, ymin, xmax, ymax)
**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) \((x_{min}, y_{min}, x_{max}, y_{max})\)

- Point test:
  
  \[
  \begin{align*}
  \text{if } (x \leq x_{max}) \& \& (x \geq x_{min}) \\
  \& \& (y \leq y_{max}) \& \& (y \geq y_{min})
  \end{align*}
  \]

  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 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 or 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**

- \(L=\text{left} \quad (\text{if } x<x_{min}, L=1, \text{ else } L=0)\)
- \(R=\text{Right} \quad (\text{if } x>x_{max}, R=1, \text{ else } R=0)\)
- \(B=\text{Bottom} \quad (\text{if } y<y_{min}, B=1, \text{ else } B=0)\)
- \(T=\text{Top} \quad (\text{if } y>y_{max}, T=1, \text{ else } B=0)\)
- Divides 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>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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
- If both endpoints are outside same border
  – (Category 2 line)
- Then both region codes will have 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:
  x1 <- xmin
  y1 <- m*(xmin-x2) + y2
  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:
  - \( y_1 \leftarrow \text{ymin} \)
  - \( x_1 \leftarrow (\text{ymin} - y_2)/m + x_2 \)
  - Reset RC’s B bit

If RC == xxx1 (P1 above ymax)

- New endpoint should be on the top boundary:
  - \( y_1 \leftarrow \text{ymax} \)
  - \( x_1 \leftarrow (\text{ymax} - y_2)/m + y_2 \)
  - Reset RC’s T bit

Horizontal and vertical lines are special cases

The C-S Line Clipping Algorithm

- Input:
  - Original endpoints \((x_1,y_1,x_2,y_2)\)
  - Clip region boundaries \((\text{xmin},\text{ymin},\text{xmax},\text{ymax})\)

- Output:
  - Accept Code (AC)
    - \( \text{AC} = \text{TRUE} \) --> some part of line was inside
    - \( \text{AC} = \text{FALSE} \) --> no part of line was inside
  - Clipped Line endpoints \((x_1,y_1,x_2,y_2)\)
    - only if \( \text{AC} = \text{TRUE} \)

C-S Algorithm Pseudo-code:

```c
CS_LineClip(xmin, ymin, xmax, ymax, x1, y1, x2, y2, AC)
done = FALSE
While (!done)
    Calculate endpoint codes rcl, rc2
    If ((rcl | rc2) == 0) // Category 1
        done = TRUE
        AC = TRUE
    Else
        If ((rcl & rc2) != 0) // Category 2
            done = TRUE
            AC = FALSE
        Else
            If (P1 is inside)
                Swap (x1, y1), (x2, y2); and rcl, rc2
```

```
```
If (L-bit of rc1 is set) // lxxx
  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

Polygon Clipping
- Clip a polygon to a rectangular clip area
- Input
  - Ordered list of polygon vertices (vin[])
  - Clip rectangle boundary coordinates (xmin, ymin, xmax, ymax).
- Output:
  - An ordered list of clipped polygon vertices (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 polygon edges
      - vin[] -> Clip Top -> Clip Bottom -> Clip Right -> vtemp0[]
      - vtemp1[] -> Clip Top -> vtemp2[]
      - vtemp3[]
      - 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)

    VERTEX i VERTEX i+1 ACTION
    in    in    Add Vertex i+1 to output list
    out   out   Add no vertex to output list
    in    out   Add intersection point with edge to output list
    out   in    Add intersection point with edge and vertex i+1 to output list

Original Polygon

Clipped Polygon
Sample Traversal

Using sh_clip() to clip a polygon

Make four calls to sh_clip():

- sh_clip(ni, vi[], 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);

Example of S-H Clipping

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

Implementation

Function sh_clip()

- Will clip an input polygon (ni, vi[])
- With respect to a given boundary (bndry)
- Generating an output polygon (no, vo[])

Enumerate the boundaries as:

- LEFT, RIGHT, BOTTOM, and TOP

sh_clip(ni, vi[], no, vo[], xmin, ymin, xmax, ymax, bndry);

vi[] and vo[] could be arrays of POINTs
ni, no: number of points in each array
xmin, ymin, xmax, ymax: clip region boundaries

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)
      Else
        output(Second_V, no, vo)   // no "out-out" case
  First_V = Second_V             // prepare for next edge
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

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
- $V = A \times B$
- $|V| = |A| |B| \sin(\theta)$
- Direction: RH Rule
- In terms of components
  - $A \times B = \begin{vmatrix} i & j & k \\ Ax & Ay & Az \\ Bx & By & Bz \end{vmatrix}$
  - $i, j, k$: 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

- E0 X E1 -> k
- E1 X E2 -> k
- E2 X E3 -> k

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

abcd -> aed & ebc

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