Terminology

- **World Coordinate System** (Object Space): The space in which the application model is defined. The representation of an object is measured in some physical or abstract units.
- **Screen Coordinate System** (Image Space): The space in which the image is displayed. Usually measured in pixels, but could use any units.

Terminology (cont.)

- **(World) Window**: The rectangle defining the part of the world we wish to display
- **(Screen) Window**: The visual representation of the screen coordinate system for "windowed" displays (coordinate system moves with screen window)
- **Viewport**: The rectangle within the screen window defining where the image will appear (Default is usually entire interface window)
Terminology (cont.)
- **Window-Viewport Transformation**: The process of mapping from a window in world coordinates to a viewport in screen coordinates.

Windows and Viewports
- **World Window**: The window in the world coordinate system.
- **Screen Window**: The window in the screen coordinate system.
- **Viewport**: A window that is clipped to show only the information within.

### Window-Viewport Transformation
1. Choose Window in World Coordinates
2. Clip to size of world window
3. Translate to origin
4. Scale to size of Viewport
5. Translate to proper position in screen window
Window-Viewport Transformation Example

For a point (x, y):
3. Translate window to origin: \( x' = x - 10 \); \( y' = y - 5 \)
4. Scale to size of viewport: \( x'' = \frac{5}{4}x' \); \( y'' = 2y' \)
5. Translate to viewport pos.: \( x''' = x'' + 25 \); \( y''' = y'' + 25 \)

Notes on W-V Transformation
- **Panning**: Moving the window about the world
- **Zooming**: Reducing or increasing the window size
- As the window increases in size, the image in the viewport decreases in size and vice versa
- Beware of aspect ratio.
W-V transformation in openGL

- `gluOrtho2d(left, right, bottom, top)` takes care of it for you!
- ex: if my window is 5 meters by 4 meters and has a lower-left corner at (2, 0) in world coordinates: `gluOrtho2D(2, 7, 0, 4)` will automatically do the transformation for me
- For generality, also need `glViewport(x, y, width, height)`, but the default viewport uses the whole interface window.

Sequence of openGL transformations

- Vertex
- Modelview Matrix
- Projection Matrix
- Viewport Xform

GL_MODELVIEW mode
- `glTranslate()`
- `glRotate()`
- `glScale()`
- `glLoadMatrix()`
- `glMultMatrix()`
- `gluLookAt()`

GL_PROJECTION mode
- `glOrtho()`
- `gluOrtho2D()`
- `glFrustum()`
- `gluPerspective()`
- `glViewport()`

Reshaping the window

- We can reshape and resize the OpenGL display window by pulling the corner of the window
- What happens to the display?
  - Must redraw from application
  - Two possibilities
    - Display part of world
    - Display whole world but force to fit in new window
      - Can alter aspect ratio
Reshape possibilities

original  reshaped

The Reshape callback

glutReshapeFunc(myreshape)
void myreshape(int w, int h)

- Gets width and height of new window (in pixels) as parameters
- A redisplay is posted automatically at end of execution of the callback
- GLUT has a default reshape callback but you probably want to define your own
- The reshape callback is good place to put viewing functions because it is invoked when the window is first opened

Example Reshape

- This reshape preserves shapes by making the viewport and world window have the same aspect ratio

  void myReshape(int w, int h)
  |
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION); /* switch matrix mode */
  glLoadIdentity();
  if (w <= h)
    gluOrtho2D(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w);
  else  gluOrtho2D(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0);
  glMatrixMode(GL_MODELVIEW); /* return to modelview mode */
openGL example for windows, viewports, and reshaping

Clipping Lines

A point is visible if
\[ x_l < x < x_r \]
and
\[ y_b < y < y_t \]

- A line is completely visible if both of its end points are in the window.
- **Brute Force Method**: Solve simultaneous equations for intersections of lines with window edges.

Cohen-Sutherland Algorithm

- **Region Checks**: Trivially reject or accept lines and points
- Fast for large windows (everything is inside) and for small windows (everything is outside)
- Each vertex is assigned a four-bit **outcode**
### Cohen-Sutherland Clipping (cont.)

- **Bit 1**: Above
- **Bit 2**: Below
- **Bit 3**: Right
- **Bit 4**: Left

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>Accepted</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>Below</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>Above</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>Right</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>Left</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>Left</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>Right</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>Right</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>Above</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>Both</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>Both</td>
</tr>
</tbody>
</table>

- Bit 1: 1 if \( y > y_t \), else 0
- Bit 2: 1 if \( y < y_b \), else 0
- Bit 3: 1 if \( x > x_r \), else 0
- Bit 4: 1 if \( x < x_l \), else 0

### Clipping Lines Not Accepted or Rejected

- In the case where a line can be neither trivially accepted nor rejected, the algorithm uses a "divide and conquer" method.

**Line AB:**
1. Test outcodes of A and D → can’t accept or reject.
2. Calculate intersection point B, which is on the dividing line between the window and the "above" region. Form new line segment AB and discard BD because above the window.
3. Test outcodes of A and B. Reject.

**Line EH:**
??
Polygon Clipping

- Polygons can be clipped against each edge of the window one edge at a time. Window/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges. Note that the number of vertices usually changes and will often increase.

Polygon Clipping Algorithm

- The window boundary determines a visible and invisible region.
- The edge from vertex i to vertex i+1 can be one of four types:
  - Exit visible region - save the intersection
  - Wholly outside visible region - save nothing
  - Enter visible region - save intersection and endpoint
  - Wholly inside visible region - save endpoint

Polygon clipping issues

- The final output, if any, is always considered a single polygon.
- The spurious edge may not be a problem since it always occurs on a window boundary, but it can be eliminated if necessary.
Because polygon clipping does not depend on any other polygons, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. This way four polygons can be at different stages of the clipping process simultaneously. This is often implemented in hardware.