2D Viewing and Clipping

Kim Steenstrup Pedersen

kimstp@itu.dk

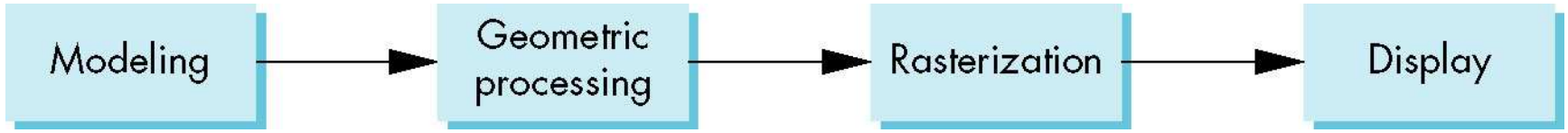
www.itu.dk/courses/IG/F2004/

The IT University of Copenhagen

The plan for today

- Clipping window and viewport.
- Normalized coordinates.
- 2D viewing in OpenGL.
- Clipping of points, lines, and polygons.
- Aside: Glut event handling.

Data flow in the graphics pipeline



Modeling: Specifying scene objects in terms of geometric primitives defined in world coordinates.

Geometric processing: Transformation, projection, and clipping of geometric primitives as well as hidden-surface removal.

Rasterization: Geometric primitives are rasterized or scan converted. Raster graphics is the approximation of mathematical primitives such as points, lines, polygons, etc.

Display: The contents of the framebuffer is displayed on the screen.

Clipping window and viewport

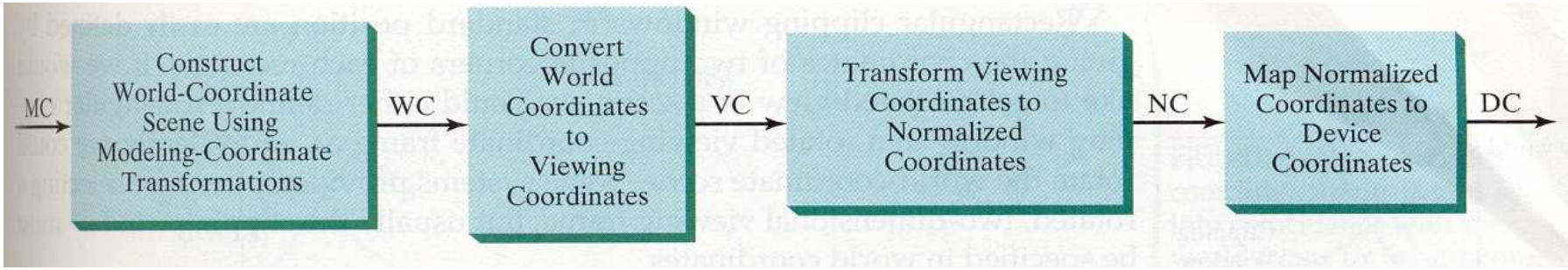
Display window: The window on the screen.

Clipping window: Part of the scene we want to draw on the screen.

Viewport: Part of the display window where the contents of the clipping window is mapped to/drawn.

From world coordinates to device coordinates

The two dimensional viewing transformation:



Viewing coordinate frame: Clipping window frame.

Device coordinate frame: Coordinates of the display window measured in pixels.

Clipping window

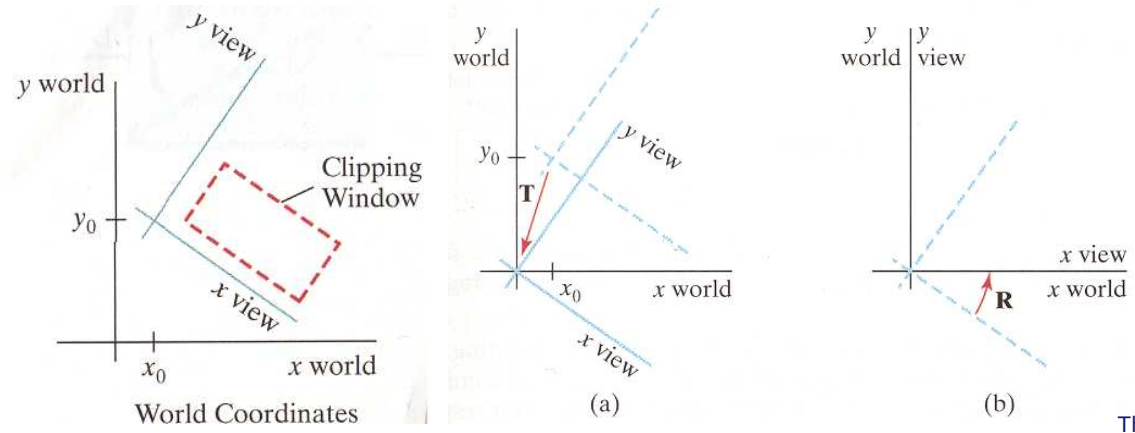
The clipping window can be any shape but usually axis aligned rectangles are used for efficiency.

We can specify other rectangular clipping windows by specifying the transformation needed between the world coordinate frame and the clipping window frame. This requires at least a translation $\mathbf{T}$ followed by a rotation $\mathbf{R}$:

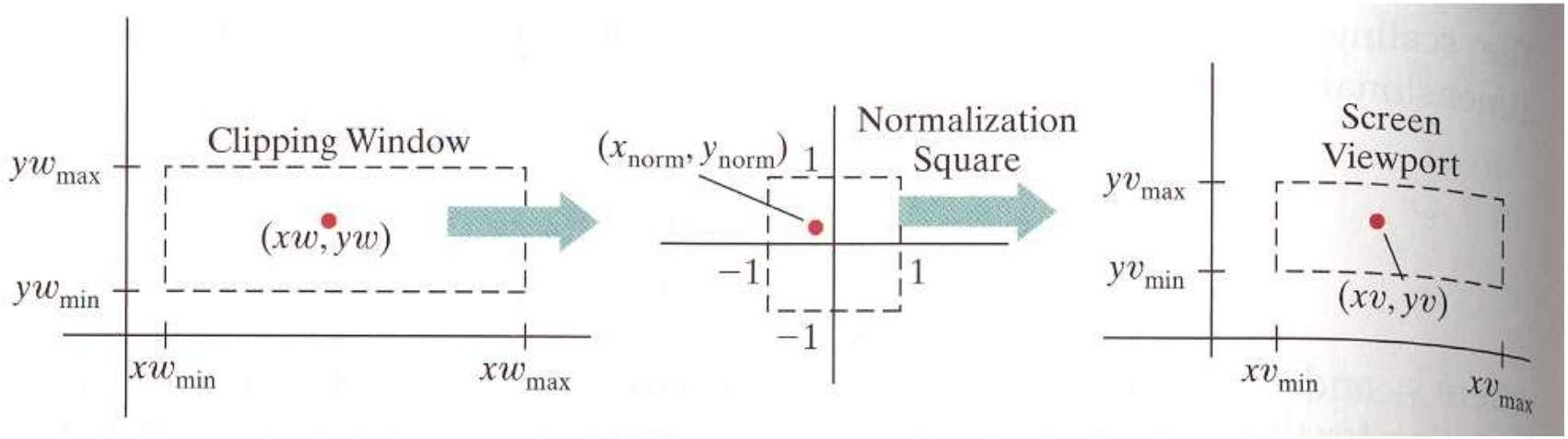
$$\mathbf{M}_{WC,VC} = \mathbf{RT} \ .$$

Apply this transformation either to the viewing coordinate frame or to the scene objects specified in the world coordinate frame.

In OpenGL you have to perform transformations of the scene objects and specify the view window in world coordinates.



Mapping the clipping window into a normalized square



Transforming objects inside the clipping window into normalized coordinates preserves relative positions, and objects outside the clipping window is also outside the normalized square after transformation.

But the aspect ratio might change causing stretched/compressed objects. To avoid this keep the aspect ratio (ratio between x and y scaling) fixed between the clipping window and viewport.

Mapping the clipping window into a normalized square (cont.)

We need to

1. Scale the clipping window relative to $(xw_{\min}, yw_{\min})$.
2. Translate $(xw_{\min}, yw_{\min})$ to $(xv_{\min}, yv_{\min})$

From clipping window to normalized square in homogeneous coordinates:

$$\mathbf{M}_{\text{window,norm}} = \begin{bmatrix} \frac{2}{xw_{\max} - xw_{\min}} & 0 & -\frac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \frac{2}{yw_{\max} - yw_{\min}} & -\frac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

Apply clipping algorithms on normalized coordinates followed by transformation to viewport:

$$\mathbf{M}_{\text{norm,viewport}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, translate the viewport coordinates into place in the display window.

2D viewing in OpenGL

In OpenGL, the clipping window is setup using the projection transformation matrix. OpenGL is inherently 3D, but the GLU library provides a function for setting up a 2D clipping window in world coordinates:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrhto2D(xwmin, xwmax, ywmin, ywmax);
```

Effectively `gluOrhto2D` defines an orthogonal projection in 3D. OpenGL uses normalized coordinates in the range -1 to 1.

The viewport may be specified by

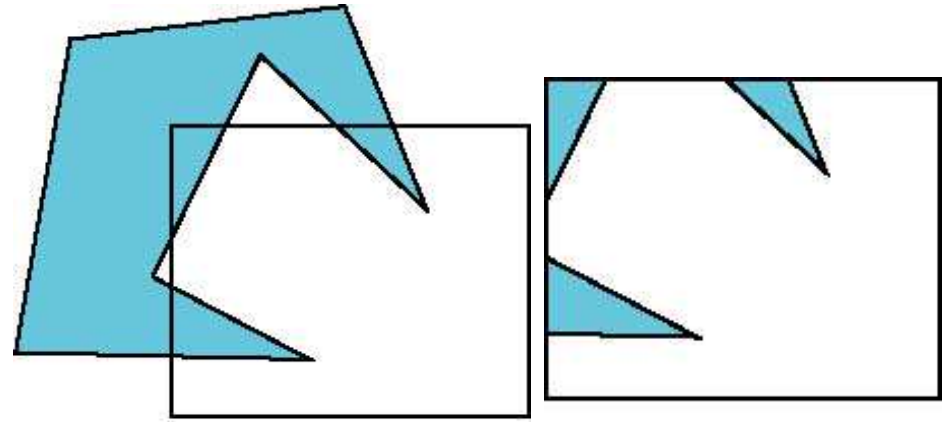
```
glViewport(xvmin, yvmin, vpWidth, vpHeight);
```

where $(xv_{\min}, yv_{\min})$ is specified in number of pixels relative to the lower left corner of the display window, and `vpWidth` and `vpHeight` is the width and height in pixels.

Default viewport size and position is the whole display window.

Clipping of geometric primitives

We want to draw only those primitives that are within the clipping window in order to save computations.



Aside:
In this case we actually created several new polygons, which might be an undesired side-effect. Can be avoided by assuming only convex polygons.

Clipping points

Clipping a point (x, y) is easy assuming that the clipping window is an axis aligned rectangle defined by $(xw_{\min}, yw_{\min})$ and $(xw_{\max}, yw_{\max})$:

Keep point (x, y) if

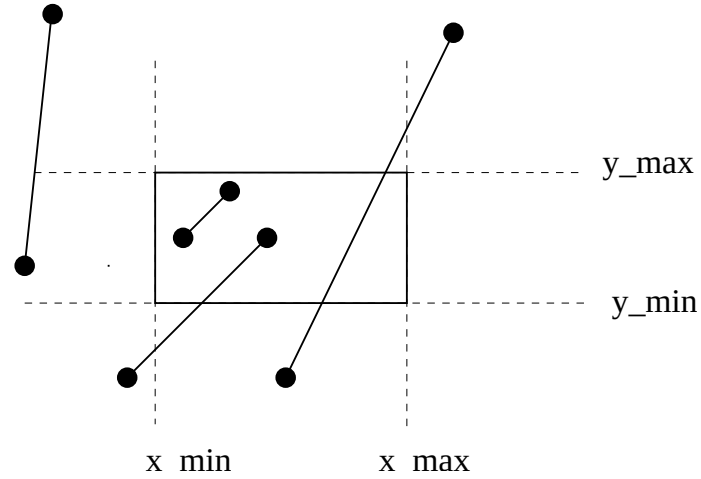
$$xw_{\min} \leq x \leq xw_{\max}$$
$$yw_{\min} \leq y \leq yw_{\max}$$

otherwise clip the point.

Using normalized coordinates, we have $(xw_{\min} = -1, yw_{\min} = -1)$ and $(xw_{\max} = 1, yw_{\max} = 1)$.

Clipping lines

We use clipping to restrict a line to a rectangular region. Line intersection calculations are expensive so we want to reduce the number of these. Several cases must be considered:



Facts:

1. Line is always drawn if both end points lie inside rectangle.
2. Line is not drawn If both end points lie left, right, above, or below the clipping rectangle.
3. If non of the above, the line has to be clipped.

Clipping lines: The simple but slow approach

Find intersections with the rectangle boundaries using the parametric line:

$$x = x_0 + u(x_1 - x_0)$$
$$y = y_0 + u(y_1 - y_0)$$

Example:
Intersection with the $xw_{\min}$ boundary:

$$xw_{\min} = x_0 + u(x_1 - x_0) \Rightarrow u = \frac{xw_{\min} - x_0}{x_1 - x_0}$$

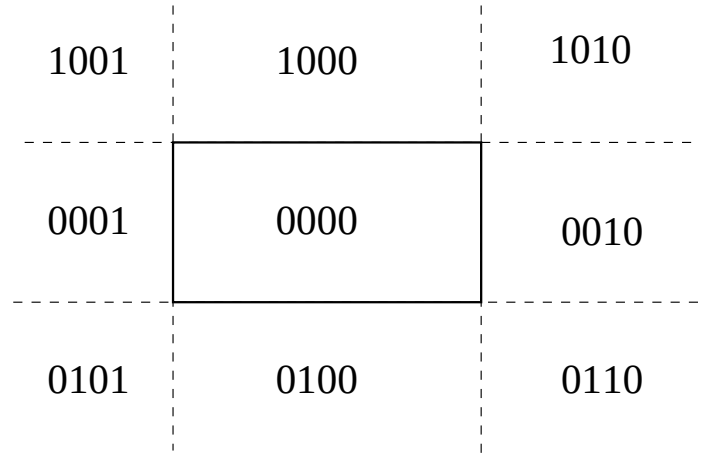
If $0 \leq u \leq 1$, intersection at $xw_{\min}$ and

$$y' = y_0 + \frac{xw_{\min} - x_0}{x_1 - x_0}(y_1 - y_0)$$

Check for intersections with the other boundaries based on the new line given by $(xw_{\min}, y')$ and (x_1, y_1) (assuming (x_0, y_0) was outside).

Clipping lines: Cohen-Sutherland

Half-plane book-keeping:



1st bit $x < xw_{\min}$

2nd bit $x > xw_{\max}$

3rd bit $y < yw_{\min}$

4th bit $y > yw_{\max}$

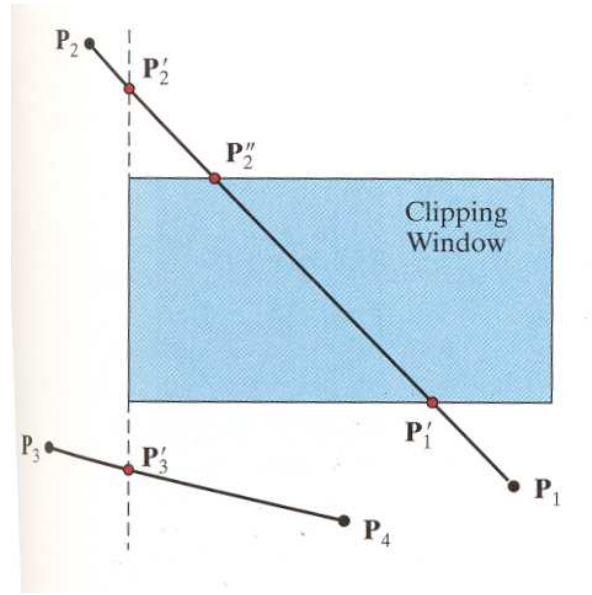
1. Draw/intersect if $c_{\text{begin}} \vee c_{\text{end}} = 0000$ ($\vee$ denotes logical or)
2. Don't draw if $(c_{\text{begin}} \wedge c_{\text{end}}) \neq 0000$ ($\wedge$ denotes logical and).

Clipping lines: Cohen-Sutherland (cont.)

Find intersections by checking the bits for each boundary:

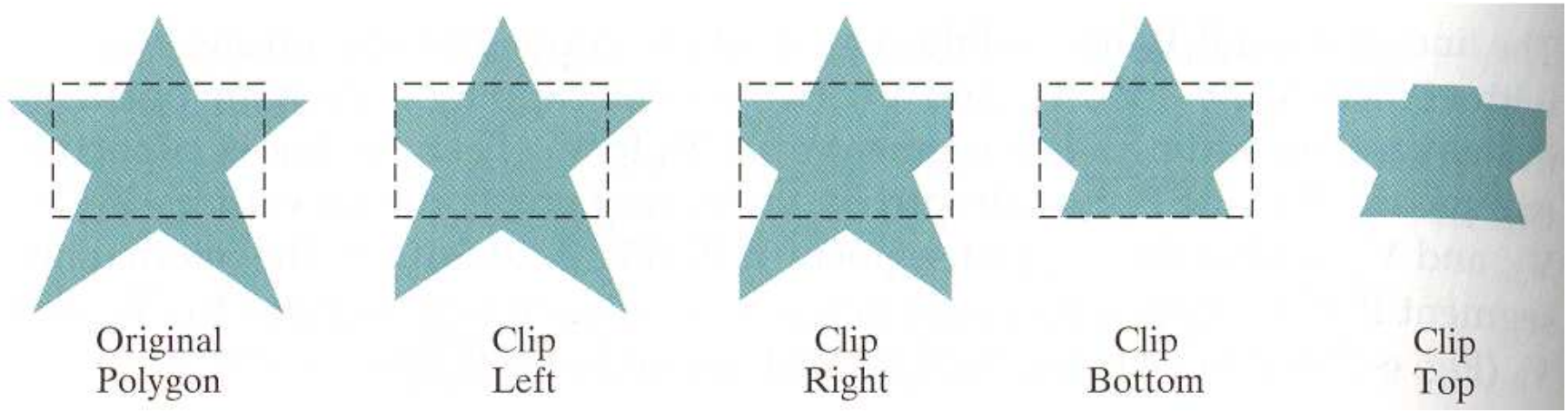
Example:

1. If both end points have different bit values in the first bit, the line intersects the left boundary (given by $xw_{\min}$).
2. If the two bits have the same value there is no intersection, continue to next boundary and bit.
3. If intersection, calculate the intersection point and check region codes. If region code check fails throw away line, otherwise keep new line segment.



Clipping polygons

Clip polygon against clipping window boundaries.

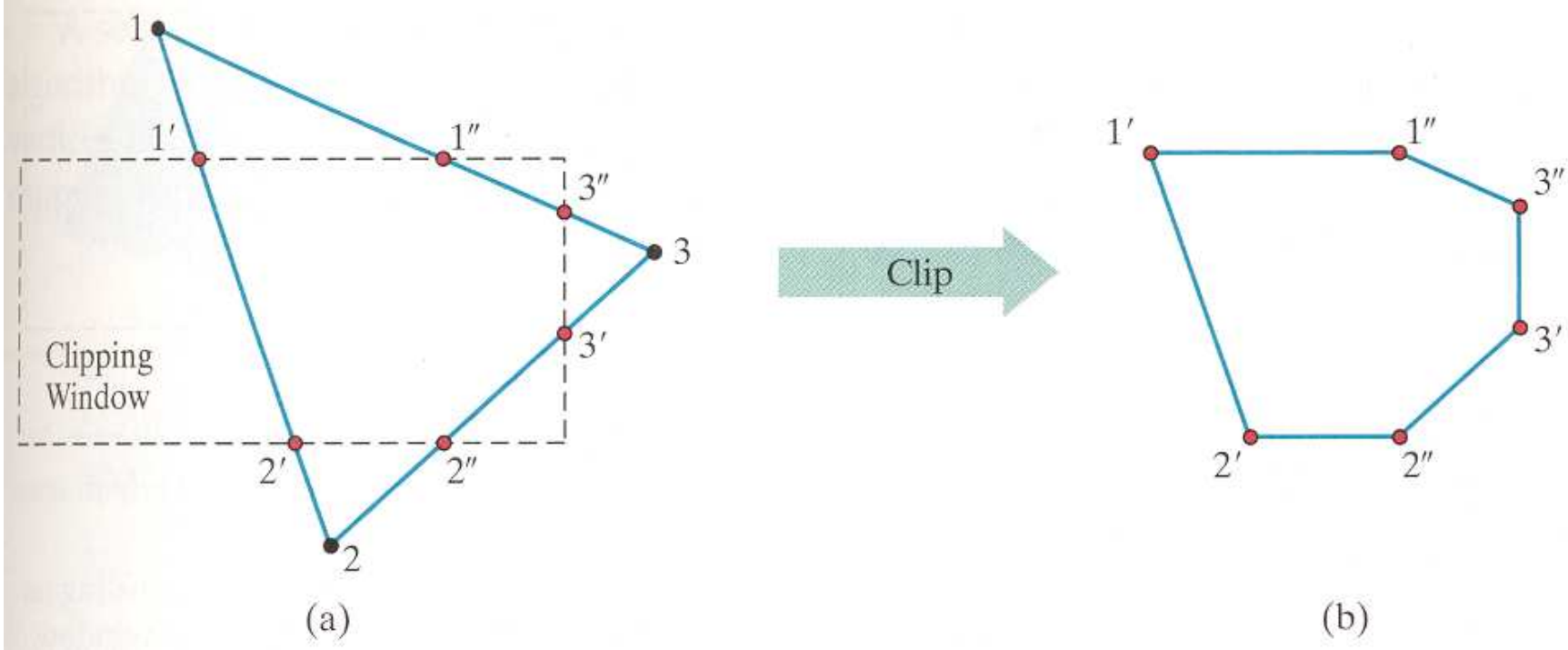


Check coordinate extents of polygon (sides of bounding box):

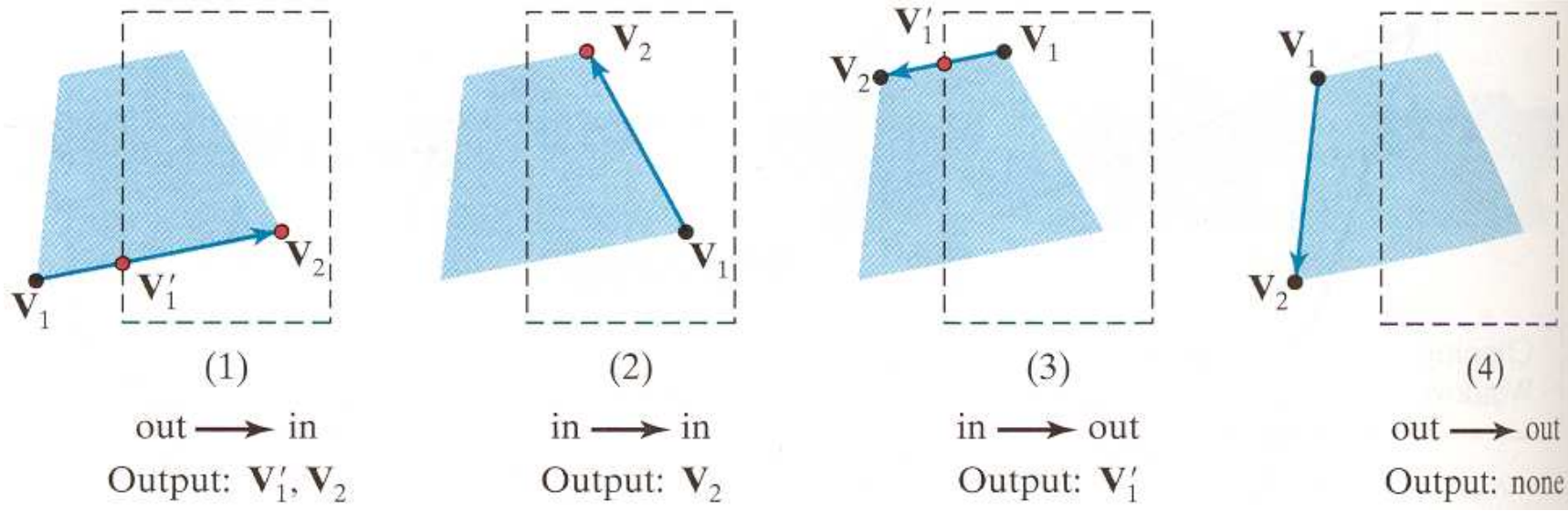
1. if min and max coordinates are inside then the polygon is inside.
2. If the coordinate extents are outside one of the clipping window boundaries then the polygon is outside.

Clipping polygons (cont.)

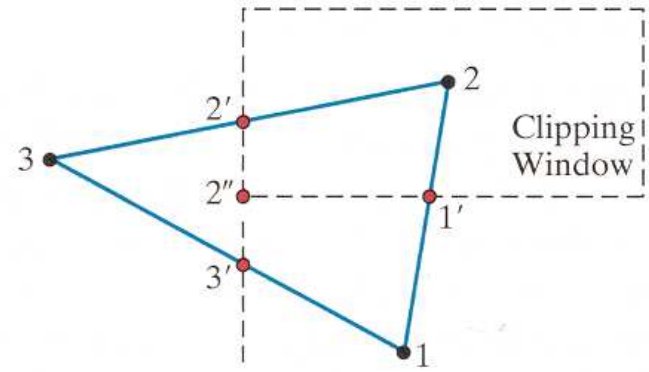
Clip convex polygons by updating a vertex list while clipping polygon edges against each boundary: $\{1, 2, 3\} \Rightarrow \{1', 2', 2'', 3', 3'', 1''\}$

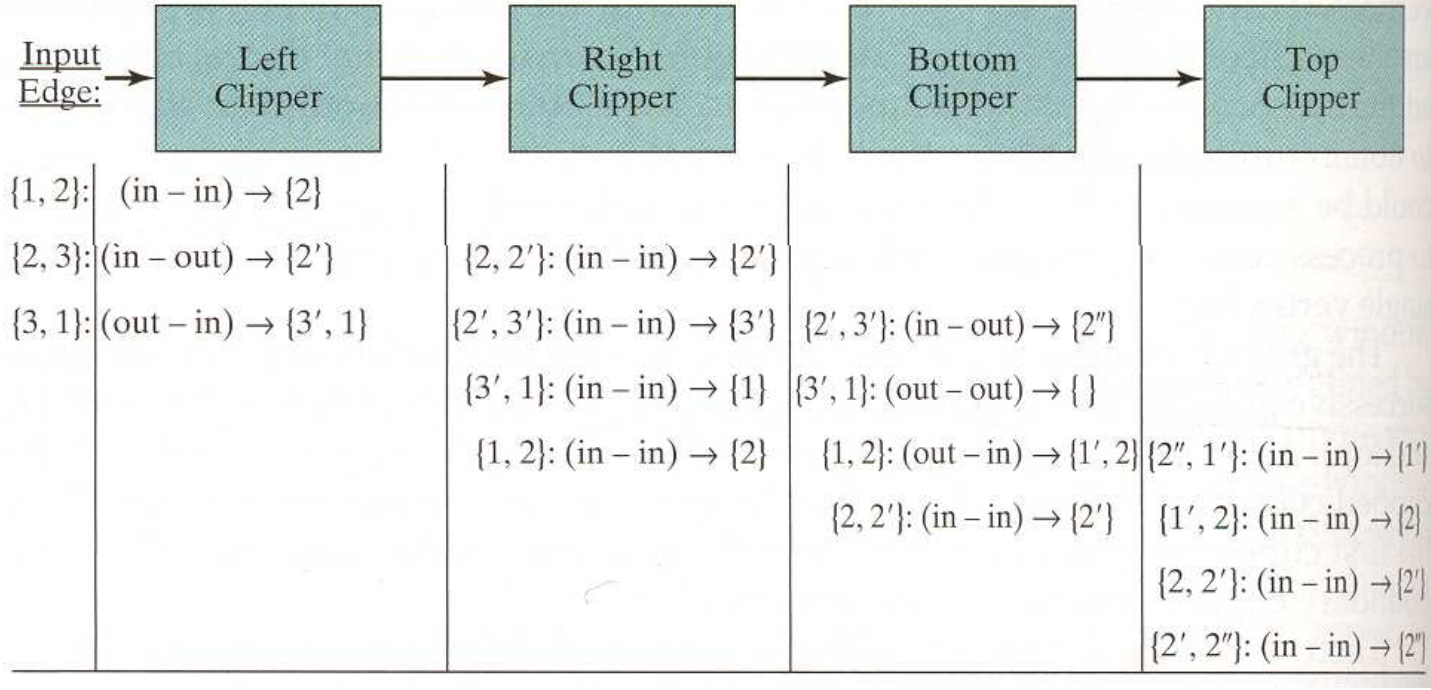


Clipping polygons: Sutherland-Hodgman



Clipping polygons: Sutherland-Hodgman (cont.)





Aside: Glut event handling

Glut is event-based and events are handled through callback functions.
Possible events:

- Display
- Reshape
- Mouse
- Keyboard

Display event:

```
glutDisplayFunc(pictureDescrip);
```

The display callback should draw the scene.

You can force a display event by:

```
glutPostRedisplay();
```

Aside: Glut event handling (cont.)

Reshape event:

```
glutReshapeFunc(winReshapeFcn);
```

The reshape callback is called at reshape events with new window width and height as parameters.

Idle callback function:

```
glutIdleFunc(backgroundProcess);
```

Called whenever no events occur.

Event loop:

```
glutMainLoop();
```

Handles events and call the appropriate callback functions.