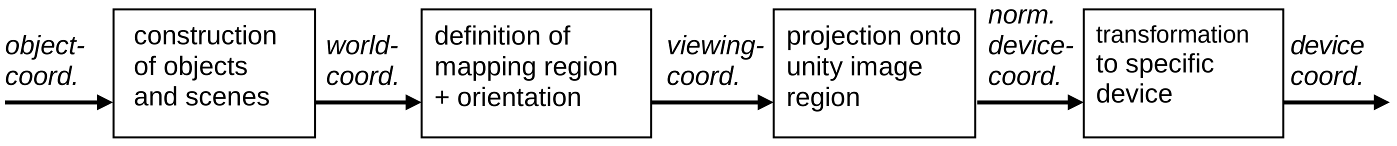
# 2D-Viewing

## 2D Viewing Pipeline

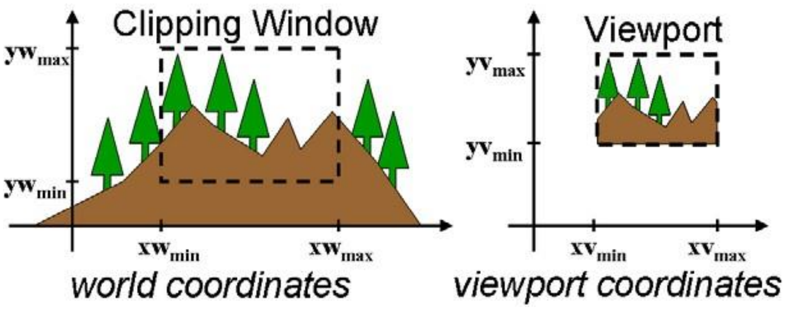
The term Viewing Pipeline describes a series of transformations, which are passed by geometry data to end up as image data being displayed on a device. The 2D viewing pipeline describes this process for 2D data:


object-coord.
construction of objects and scenes
world-coord.
definition of mapping region + orientation
viewing-coord.
projection onto unity image region
norm. device-coord.
transformation to specific device
device coord.

The coordinates in which individual objects (models) are created are called ***model (or object) coordinates***.
When several objects are assembled into a scene, they are described by ***world coordinates***.
After transformation into the coordinate system of the camera (viewer) they become ***viewing coordinates***.
Their projection onto a common plane (window) yields device-independent ***normalized coordinates***.
Finally, after mapping those normalized coordinates to a specific device, we get ***device coordinates***.

## Window-Viewport Transformation

A window-viewport transformation describes the mapping of a (rectangular) window in one coordinate system into another (rectangular) window in another coordinate system. This transformation is defined by the section of the original image that is transformed (clipping window), the location of the resulting window (viewport), and how the window is translated, scaled or rotated.



The following derivation shows how easy this transformation generally is (i.e. without rotation):

The transformation is linear in x and y, and $(xw_{min}/yw_{min}) \rightarrow (xv_{min}/yv_{min})$, $(xw_{max}/yw_{max}) \rightarrow (xv_{max}/yv_{max})$ .
For a given point (xw/yw) which is transformed to (xv,yv), we get:

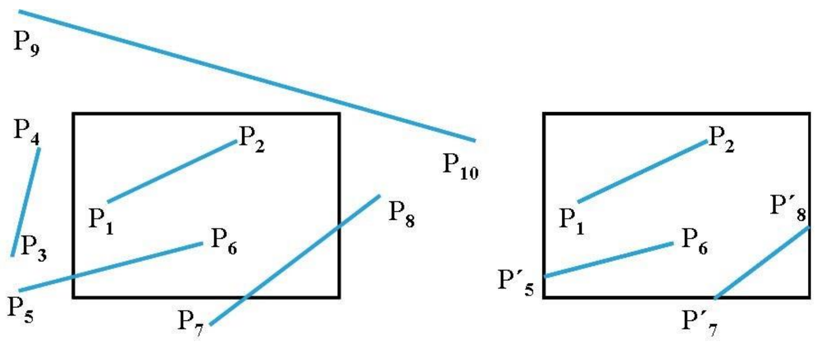
$$xw = xw_{min} + \lambda(xw_{max}-xw_{min}) \quad \text{where } 0<\lambda<1 \quad \Rightarrow \quad xv = xv_{min} + \lambda(xv_{max} - xv_{min})$$

Calculating $\lambda$ from the first equation and substituting it into the second yields:
$xv = xv_{min} + (xv_{max} - xv_{min})(xw - xw_{min})/(xw_{max} - xw_{min}) = xw(xv_{max} - xv_{min})/(xw_{max} - xw_{min}) + t_x$
where $t_x$ is constant for all points, as is the factor $s_x = (xv_{max} - xv_{min})/(xw_{max} - xw_{min})$.
Processing y analogically, we get an ordinary linear transformation: $\mathbf{xv = s_x xw + t_x,\ yv = s_y yw + t_y}$,

The chapter “Transformations” describes, how such transformations can be notated even simpler.

## Line Clipping

Clipping is the method of cutting away parts of a picture that lie outside the displaying window (see also picture above). The earlier clipping is done within the viewing pipeline, the more unnecessary transformations of parts which are invisible anyway can be avoided:



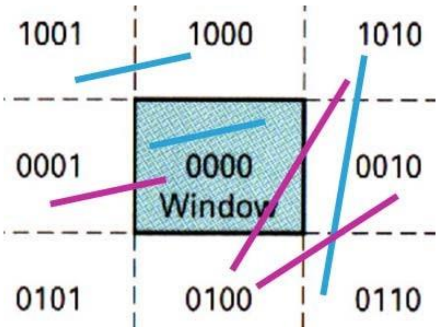
- **in world coordinates**, which means analytical calculation as early as possible,
- **during raster conversion**, i.e. during the algorithm, which transforms a graphic primitive to points,
- **per pixel**, i.e. after each calculation, just before drawing a pixel.

Since clipping is a very common operation it has to be performed simply and quickly.

***Clipping lines: Cohen-Sutherland-Method***
Generally, line clipping algorithms benefit from the fact, that in a rectangular window each line can have at most one visible part. Furthermore, they should exploit the basic principles of efficiency, like early elimination of simple and common cases and avoiding needless expensive operations (e.g. intersection calculations). Simple line clipping could look like this:

```
for endpoints (x0,y0), (xend,yend)
intersect parametric representation
     x = x0 + u*(xend - x0)
     y = y0 + u*(yend - y0)
with window borders:
     intersection ⇔ 0 < u < 1
```

The Cohen-Sutherland algorithm first classifies the endpoints of a line with respect to their relative location to the clipping window: above, below, left, right, and codes this information in 4 bits. Now the following verification can be performed quickly with the codes of the two endpoints:



1. OR of the codes = 0000 $\Rightarrow$ line completely visible
2. AND of the codes $\neq$ 0000 $\Rightarrow$ line completely invisible
3. otherwise, intersect the line with the relevant edge of the window and replace the cut away point by the intersection point. GOTO 1.

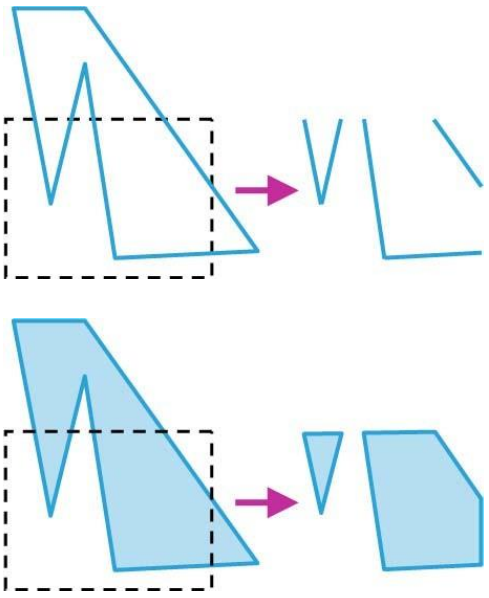
Intersection calculations:

| | | |
|---|---|---|
| with vertical window edges: | $y = y_0 + m(xw_{min} - x_0)$, | $y = y_0 + m(xw_{max} - x_0)$ |
| with horizontal window edges: | $x = x_0 + (yw_{min} - y_0)/m$, | $x = x_0 + (yw_{max} - y_0)/m$ |

Endpoints which lie exactly on an edge of the window have to be interpreted as inside, of course. Then the algorithm needs 4 iterations at most. As we can see, intersection calculations are performed only if it is really necessary.

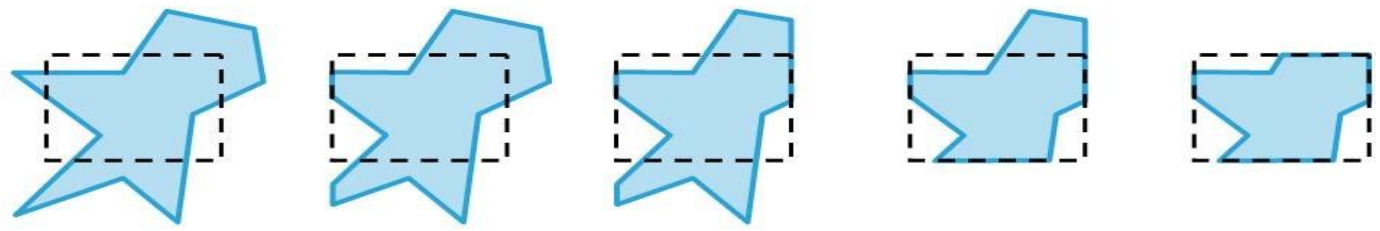
There are similar methods for clipping circles, however they have to consider that circles can be divided into more than one part when being clipped.

## Polygon Clipping

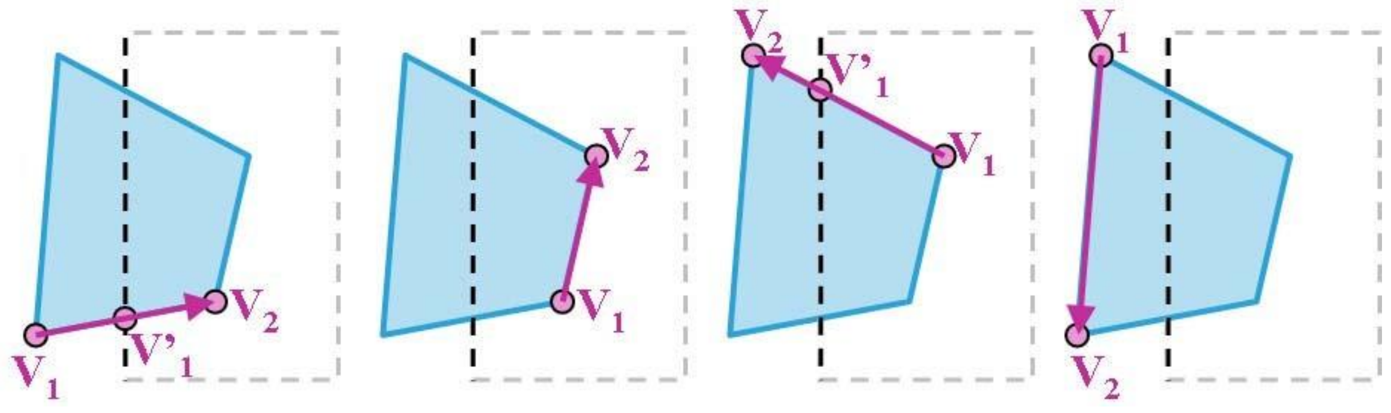


When clipping a polygon we have to make sure, that the result is *one* single polygon again, even if the clipping procedure cuts the original polygon into more than one piece. The upper figure shows a polygon clipped by a line clipping algorithm. Afterwards it is not decidable what is inside and what is outside the polygon. The lower figure shows the result of a correct polygon clipping procedure. The polygon is divided into several pieces, each of which can be filled correctly.

***Clipping polygons: Sutherland-Hodgman-Method***
The basic idea of this method yields from the fact that clipping a polygon at only one edge doesn't create many complications. Therefore, the polygon is sequentially clipped against each of the 4 window edges, and the result of each step is taken as input for the next one:

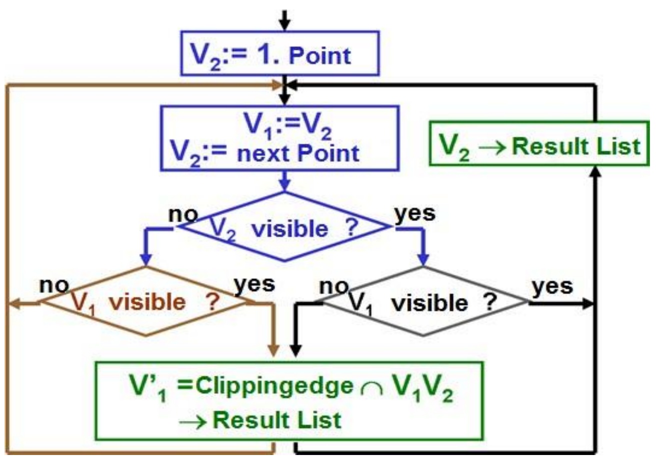


There are 4 different cases how an edge ($V_1$, $V_2$) can be located relative to an edge of the window. Thus, each step of the sequential procedure yields one of the following results:



1. out→in (output $V'_1$, $V_2$) 2. in→in (output $V_2$) 3. in→out (output $V'_1$) 4. out→out (no output)

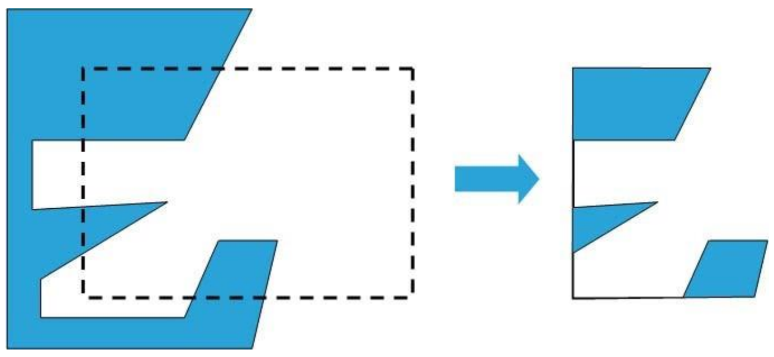
Thus the algorithm ***for one edge*** works like this:



The polygon's vertices are processed sequentially. For each polygon edge we verify which one of the 4 mentioned cases it fits, and then create an according entry in the result list. After all points are processed, the result list contains the vertices of the polygon, which is already clipped on the current window's edge. Thus it is a valid polygon again and can be used as input for the next clipping operation against the next window edge.

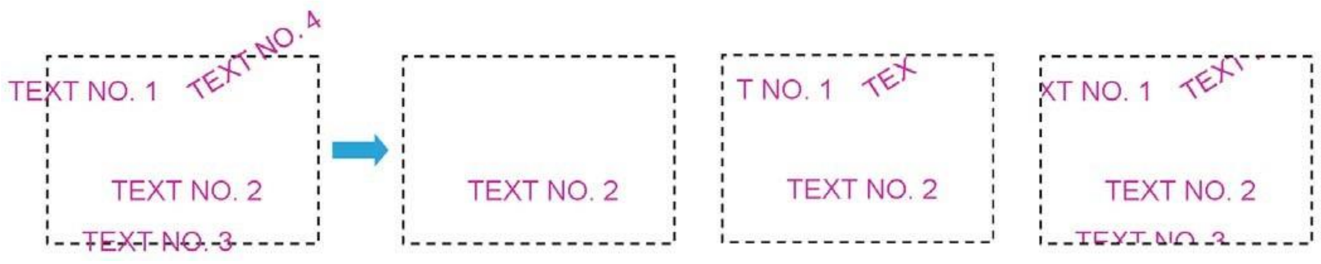
These three intermediate results can be avoided by calling the procedure for the 4 window edges *recursively*, and so using each result point instantly as input point for the next clipping operation. Alternatively, we can construct a pipeline through these 4 operations, which has a polygon as final output – the polygon which is correctly clipped against the 4 window edges.

When cutting a polygon to pieces, this procedure creates connecting edges along the clipping window's borders. In such cases, a final verification respectively post-processing step may be necessary.



## Text Clipping

At first glance clipping of text seems to be trivial, however, one little point has to be kept in mind: Depending on the way the letters are created it can happen, that either only text is displayed that is fully readable (i.e. all letters lie completely inside the window), or that text is clipped letter by letter (i.e. all letters disappear that do not lie completely inside the window), or that text is clipped correctly (i.e. also half letters can be created).



TEXT NO. 1
TEXT NO. 4
TEXT NO. 2
TEXT NO. 3
TEXT NO. 2
T NO. 1
TEX
TEXT NO. 2
XT NO. 1
TEXT NO. 2