

Transformations in 2D

Transformations allow to put objects modeled with polygons at specific locations, move them relative to the camera or to each other or to change their dimensions (scaling). This makes them particularly important in graphics. A compact and convenient way of representing transformations is by matrices.

1 2D transformations as 2x2 matrices

A lot of transformations of the plane can be represented by two by two matrices. Here are a few examples

- Rotation around the origin by the angle of α , $\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}$.
- Scaling by a factor of c_x along the x -axis and by a factor of c_y along the y -axis $\begin{bmatrix} c_x & 0 \\ 0 & c_y \end{bmatrix}$.
- Uniform scaling by a factor c (if $c = 2$ it makes everything two times bigger) $\begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix}$.
- Symmetry about the origin is equivalent to uniform scaling by a factor of -1 $\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$.
- Symmetry about the x -axis scales y by a factor of -1 and leaves the x coordinate unchanged $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$.
- Similarly, symmetry about the y -axis has the matrix $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.
- Shear along the x -axis has the matrix $\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$, and shear along y - $\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$
(Figure 1 shows how it works).

A problem with with 2x2 matrices as a representation of 2D transformations is that translations do not fit into this framework: there is no way to represent a nontrivial translation as a 2x2 matrix. Homogenous coordinates are a way around this problem: by adding one extra third coordinate they allow to represent translations as 3x3 matrices. They also allow to represent 2D perspective projections as a 3x3 matrices, which makes them even more useful for graphics.

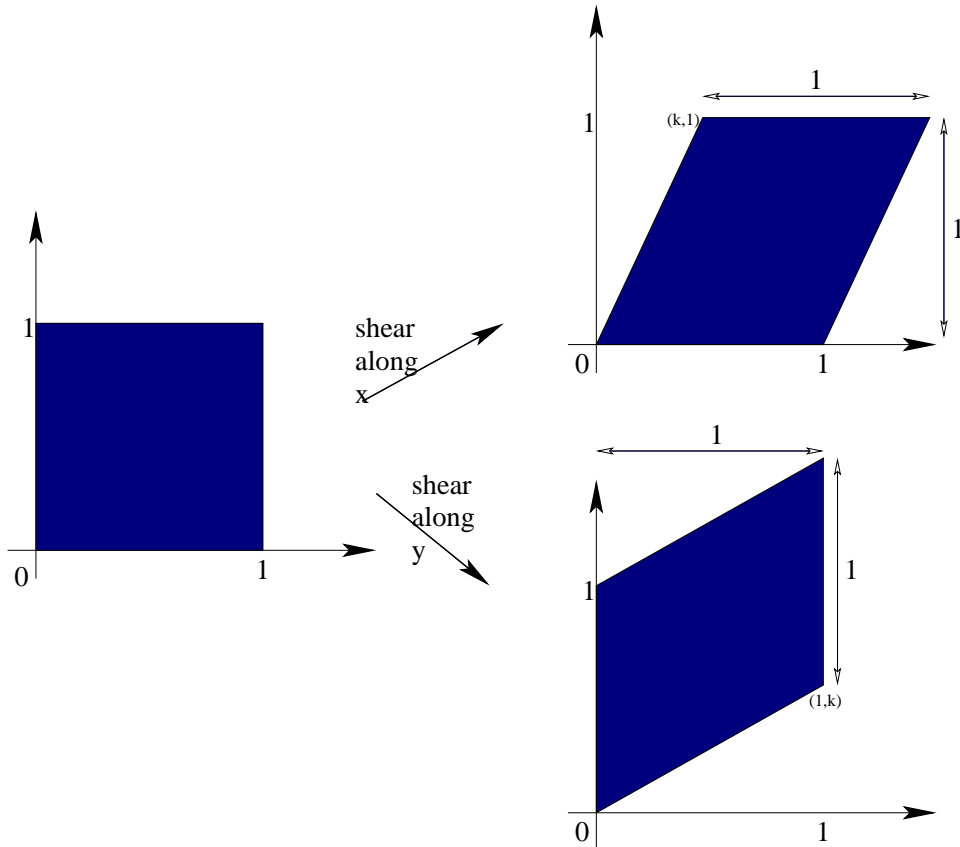


Figure 1: Shear transformations; notice they do not really change the dimensions along the axes.

2 Homogenous Coordinates

The idea is very simple: we'll think of a 2D point (x, y) as a point in 3D, namely $(x, y, 1)$. This immediately allows to represent a translation by a vector $[T_x, T_y]$ as the 3x3 matrix

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}.$$

It maps the point $(x, y, 1)$ (the 3D point corresponding to (x, y)) to $(x + T_x, y + T_y, 1)$, the 3D point corresponding to the translated point $(x + T_x, y + T_y)$. All transformations discussed in the previous section can be represented in homogenous coordinates as 3x3 matrices: just complete its 2x2 matrix to a 3x3 one by adding one row and one column like this (replace the stars with the entries of

the 2x2 matrix):

$$\begin{bmatrix} * & * & 0 \\ * & * & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3 Perspective projection in 2D

Perhaps the most interesting property of the homogenous coordinates is that they allow to represent perspective projections as matrices. Figure 2 shows a perspective projection we'll be trying to work out.

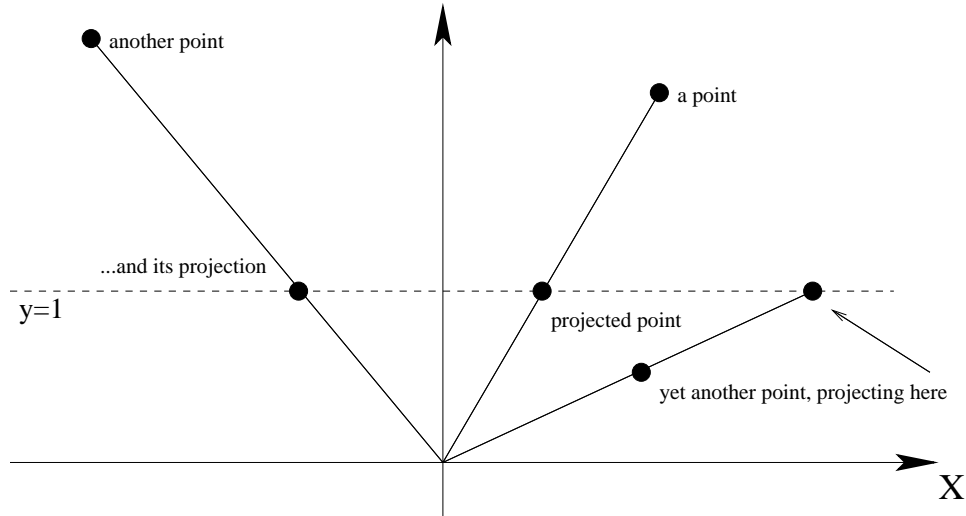


Figure 2: A perspective projection; the center of the projection (think of it as the location of the eye) is the origin; the 'screen' onto which points are projected is the dashed line described by $y = 1$.

Notice that it sends the point (x, y) to $(x/y, 1)$. Thus, it requires division, an operation not provided by the matrix formalism. To help us write the projection as a 3x3 matrix we'll use the homogenous coordinate for scaling the first two: we'll think of a point (x, y, t) as $(x/t, y/t)$. With this convention, the perspective projection can be written as the matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

This is a matrix which maps $(x, y, 1)$ ((x, y) in homogenous coordinates) to (x, y, y) which, by our scaling convention, corresponds to $(x/y, 1)$.

To get an even more interesting representation of the perspective projection we can use the matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

which represents the transformation mapping (x, y) to $(x/y, (y + 1)/y)$. It can be shown to map lines into lines. The x-coordinate of the result is exactly the x-coordinate of the perspective projected input point. The y-coordinate can be interpreted as the opposite of the depth (closeness?): as y increases from 0 to infinity, $(y + 1)/y$ decreases monotonously from infinity to 1. Thus, the upper half-plane is mapped onto the half-plane above the screen (the dashed line in Figure 2). The order of the y -coordinates is reversed by that mapping: if a point p_1 is above another point p_2 then the image of p_1 is below the image of p_2 . Therefore, something like $-(y + 1)/y$, the negative of the second coordinate of the result, is a great candidate for the depth: it reflects the order of points with respect to the eye and can be computed from vertices by linear interpolation. An interesting property of our mapping is that it maps families of parallel lines onto families of lines intersecting at one point of the screen as shown in Figure 3. This should not be a surprise, since any parallel lines in the 'real world' appear to come together at the horizon – think about lanes on a straight highway, or look at Figure 4.

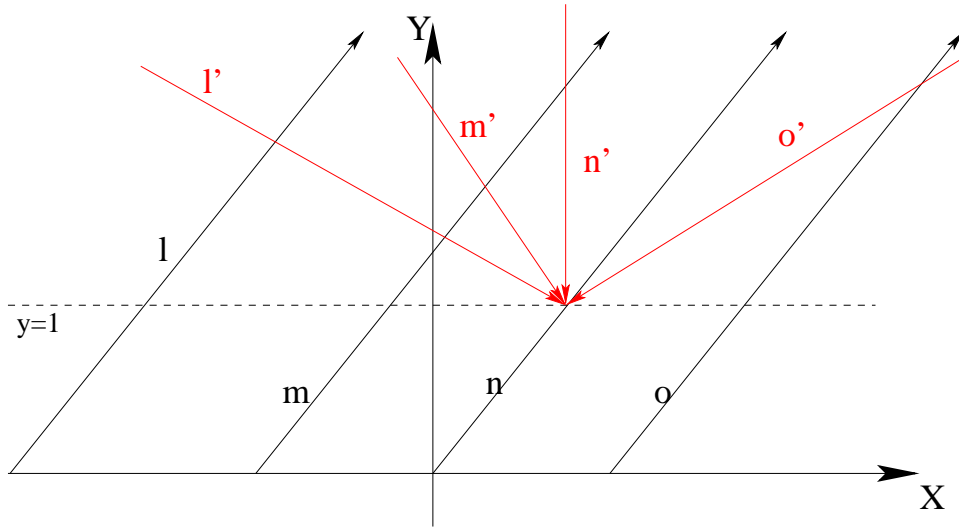


Figure 3: The black lines (in a family of parallel lines) are mapped into the red lines, all of which meet at one point on the screen. This is the point where the black line n passing through the eye intersects the screen. Notice that the entire line n projects to a single point – thus the x-coordinate of the projection is constant and the corresponding line n' is vertical.



Figure 4: Projections of parallel (here: vertical) lines intersect at one point.

4 Viewing transformation

A cool thing about the transformation discussed in the previous section is that it allows to turn any perspective projection into a parallel projection onto the x -axis: an extremely simple transformation. It also allows to replace a *view volume* – in general, a prism-shaped volume containing all objects to be drawn, into a rectangularly shaped one. This makes some processing operations, like clipping (we'll talk about it later) much simpler.

The 2D counterpart of the view volume is bounded by four lines (see 5):

- The *far clipping plane*: no object (or part of an object) behind that plane will not be drawn,
- The *near clipping plane*: no object in front of it will be drawn,
- Two lines defining the field of view through the screen.

Of course, in 3D we have the near and far planes and the field of view is bounded by four planes, each passing through the eye and one of the edges of the screen.

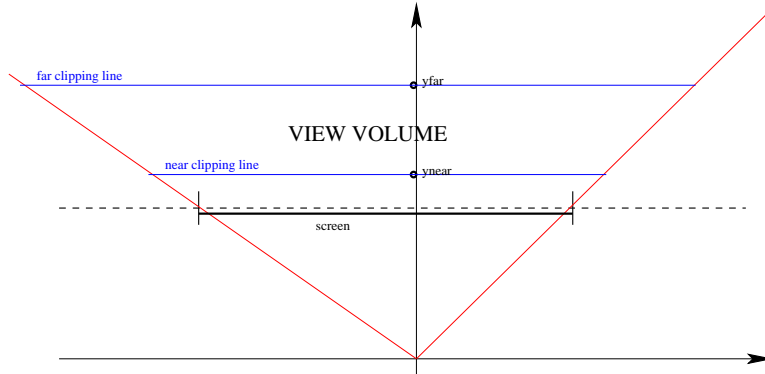


Figure 5: View volume: the red lines bound the field of view through the screen.

The view volume after the perspective transformation $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, is applied becomes an axis-aligned rectangle shown in Figure 6. Certain operations like polygon or line clipping are easier to perform against an axis-aligned rectangle than against a general view volume. Perspective projection becomes a parallel projection (in our case, orthogonal onto the x-axis). Since lines are mapped to lines, linear interpolation of depth in the polygon scan-conversion stage leads to exactly correct results. This is illustrated in Figure 7.

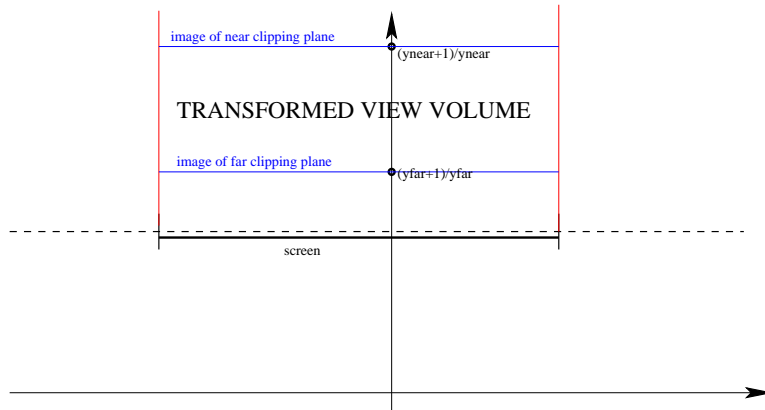


Figure 6: View volume: the red lines bound the field of view through the screen.

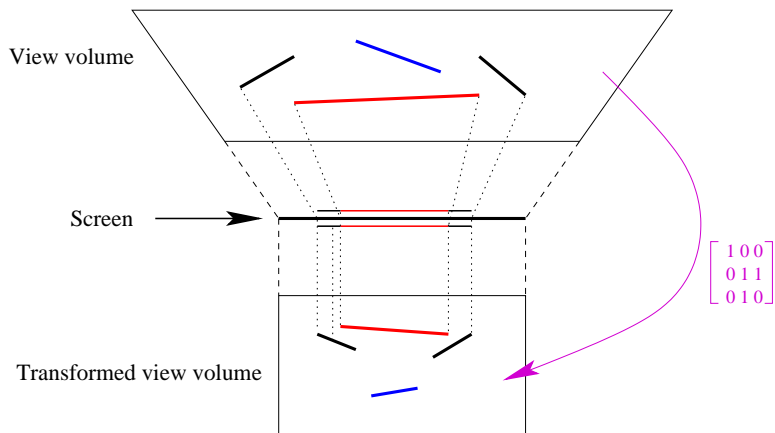


Figure 7: Perspective transformation turns a perspective projection into a parallel projection. View volume becomes a rectangle. The parallel projection of transformed primitives (here: color lines inside the transformed view volume) is the same as the image of of the untransformed ones under a perspective projection.