Basics of 2D and 3D graphics

Viewing transformations
Frustum

Piotr Fulmański

piotr@fulmanski.pl

December 10, 2016

Spis treści

We assume that we are drawing a model consisting only of 3D line segments that are specified by the (x, y, z) coordinates of their two end points. The viewing transformation we are going to show now has mapp 3D locations (3D lines), represented as (x, y, z) coordinates in some arbitrary coordinate system, to coordinates in the image, expressed in units of pixels. This process depends on

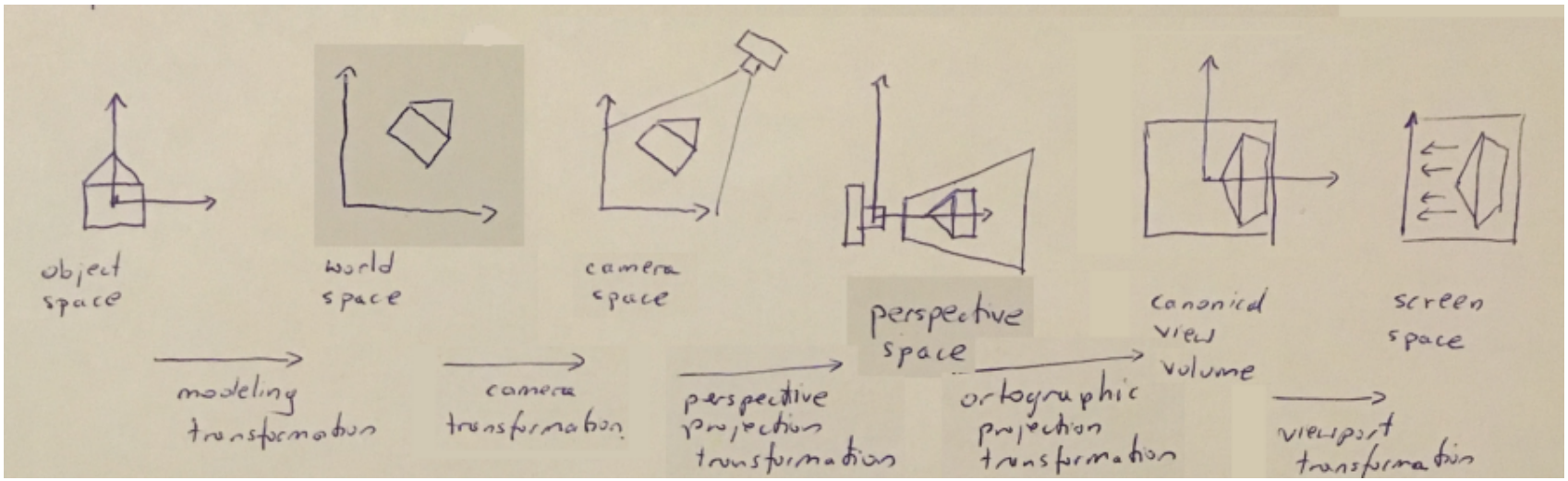
- the camera position and orientation,
- the type of projection,
- the field of view,
- and the resolution of the image.

We can break up this complicated process in to a product of several simpler steps (transformations). Most graphics systems do this by using a sequence of three transformations

- A **camera transformation** (or eye transformation), which is a rigid body transformation that places the camera at the origin in a convenient orientation. It **depends only on the position and orientation of the camera**.
- A **projection transformation**, which projects points from camera space so that all visible points fall in the range from -1 to 1 for both x and y. It **depends only on the type of projection desired**.
- A **viewport transformation** (or windowing transformation), which maps this unit image rectangle to the desired rectangle in pixel coordinates. It **depends only on the size and position of the output image**.

Viewing transformations

The idea



The sequence of transformations that gets object from its original object space into screen space.

So we are looking for transformation matrix M

$$M = M_{\text{view}} M_{\text{orth}} P M_{\text{cam}}$$

where

- M_{view} is a viewport transformation,
- M_{orth} is an ortographic projection transformation which projects points from any cube (view volume) to unified view volume where all visible points fall in the range from -1 to 1,
- P is a projection transformation, which projects points from camera space to some cube,
- M_{cam} is a camera transformation, which places the camera at the cpecified point of the world and look at specified direction with specified orientation.

M_{view} — viewport transformation

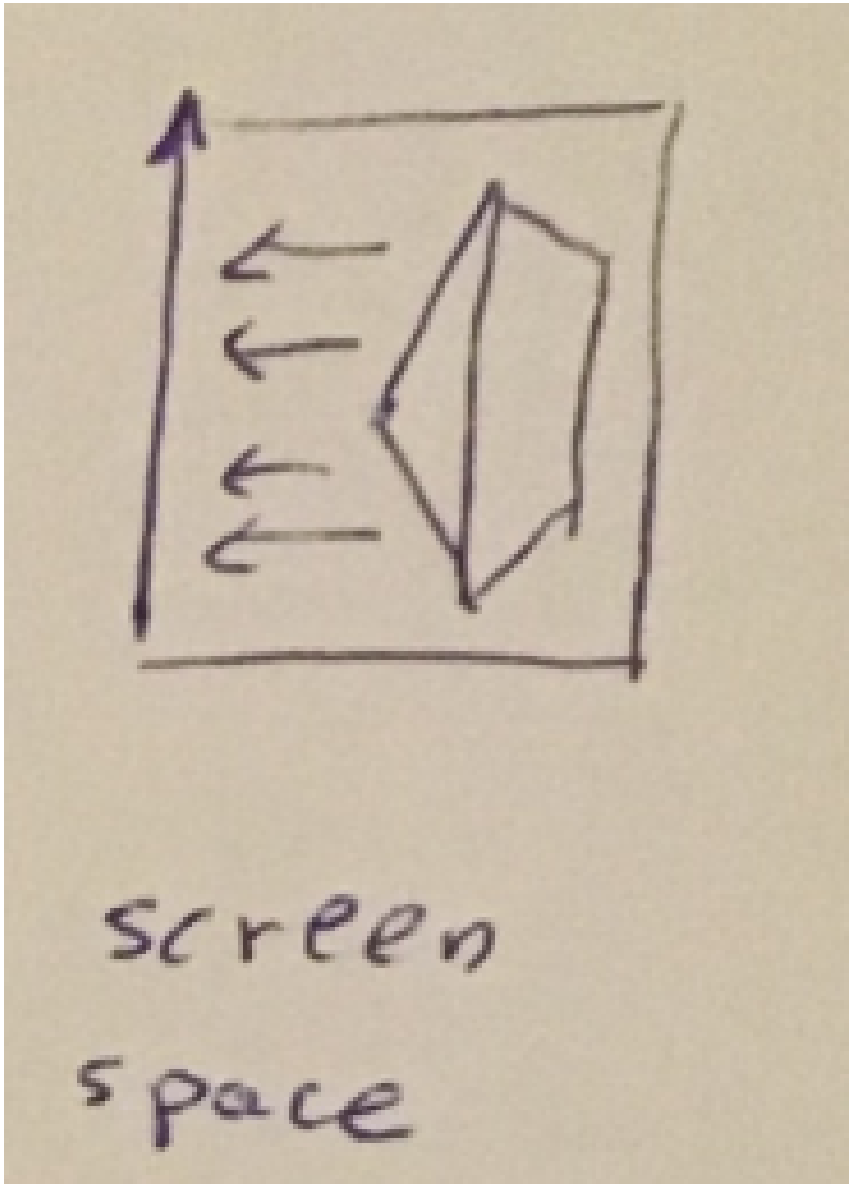
Let's introduce a concept of the unified (sometimes we say: *canonical*) view volume: this could be any (but fixed) arbitrarily chosen volume. In our case the unified view volume is the cube containing all 3D points whose Cartesian coordinates x, y and z are between -1 and $+1$.
Now we assume that the geometry we want to view is described in this unified volume, and we wish to view it with an orthographic[1] camera looking in the $-z$ direction. Se we project

- $x = -1$ to the left side of the screen,
- $x = +1$ to the right side of the screen,
- $y = -1$ to the bottom of the screen,
- $y = +1$ to the top of the screen.

[1] Orthographic projection (or orthogonal projection) is a means of representing a three-dimensional object in two dimensions. It is a form of parallel projection, where all the projection lines are orthogonal to the projection plane.

Viewport transformation

Unified view volume



We use integer numbers as pixel coordinates. Physical pixel has some dimensions and it's shape is square (or rectangular), so we can ask which pixel's point has these integer coordinates?

Let's assume, that pixel's center point corresponds to integer coordinates. Other words, for every pixel there is a corresponding unit square centered at integer coordinates.

In consequence

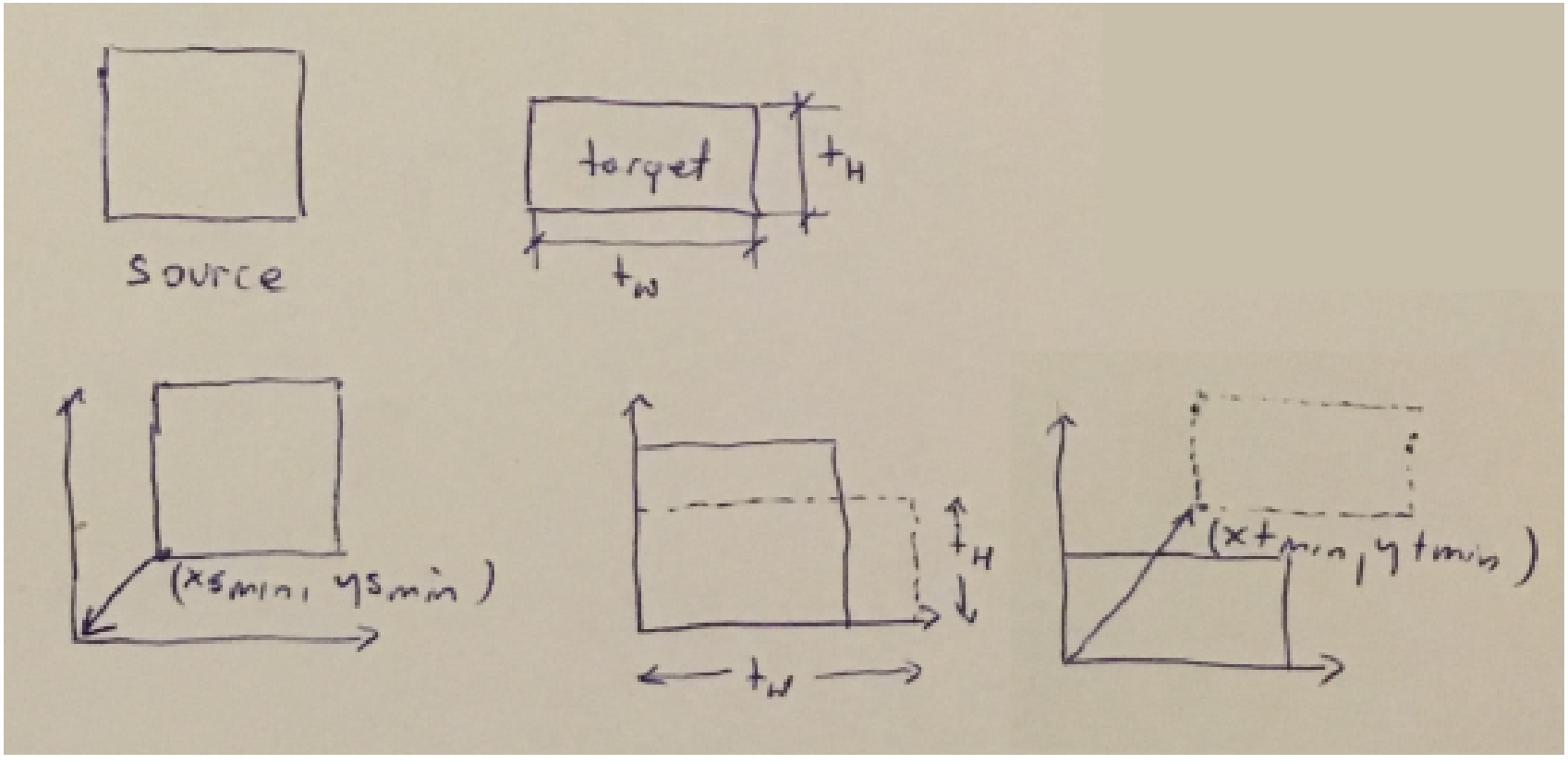
- the image boundaries have a half-unit overshoot from the pixel centers;
- the smallest pixel center coordinates are $(0, 0)$;
- we are drawing into an image (or window on the screen) that has n_x by n_y pixels, we need to map the square $[-1, 1] \times [-1, 1]$ to the rectangle $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

Imagine that we need to create a transform matrix that takes points in the rectangle $[xs_{min}, xs_{max}] \times [ys_{min}, ys_{max}]$ to the rectangle $[xt_{min}, xt_{max}] \times [yt_{min}, yt_{max}]$. It's not difficult to note that this can be accomplished with two transformation in sequence: a scale and translate. However, to find correct transformation it would be more convenient to think about it as a sequence of three operations.

1. Move source rectangle so the point (xs_{min}, ys_{min}) is located in the origin.
2. Scale the rectangle to be the same size as the target rectangle.
3. Move the origin to the point (xt_{min}, yt_{min}).

Viewport transformation

Windowing transformation – general case



Step 1: move source rectangle

Move source rectangle so the point (xs_{min}, ys_{min}) is located in the origin. We do this with move by a vector $[-xs_{min}, -ys_{min}]$. In matrix form, this transformation (which is translation) takes form

$$T_{\text{source}\to\text{origin}} = T_{so} = \begin{bmatrix} 1 & 0 & -xs_{min} \\ 0 & 1 & -ys_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

To verify this, let's take a point $p = (x, y)$, correct T_{so} matrix for it and check if the result of calculation (x_r, y_r) returns a point $(0, 0)$

$$\begin{bmatrix} x_r \\ y_r \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Step 2: scale the rectangle

Scale the rectangle to be the same size as the target rectangle.
Verify, that we do this with transformation (which is scaling) taking a form

$$T_{scale} = T_s = \begin{bmatrix} \frac{xt_{max} - xt_{min}}{xs_{max} - xs_{min}} & 0 & 0 \\ 0 & \frac{yt_{max} - yt_{min}}{ys_{max} - ys_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 3: move the origin

Move the origin to the point (xt_{min}, yt_{min}).
We do this with move by a vector $[xt_{min}, yt_{min}]$. In matrix form, this transformation (which is translation) takes form

$$T_{\text{origin}\to\text{target}} = T_{ot} = \begin{bmatrix} 1 & 0 & xt_{min} \\ 0 & 1 & yt_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

Viewport transformation

Windowing transformations – general case: finall window transformation for 2D case

$$T_w = T_{ot}\, T_s\, T_{so} = \begin{bmatrix} \frac{xt_{max} - xt_{min}}{xs_{max} - xs_{min}} & 0 & \frac{xt_{min}\, xs_{max} - xt_{max}\, xs_{min}}{xs_{max} - xs_{min}} \\ 0 & \frac{yt_{max} - yt_{min}}{ys_{max} - ys_{min}} & \frac{yt_{min}\, ys_{max} - yt_{max}\, ys_{min}}{ys_{max} - ys_{min}} \\ 0 & 0 & 1 \end{bmatrix}$$

An exactly analogous construction can be used to define a 3D windowing transformation

$$T_w = \begin{bmatrix} \frac{xt_{max} - xt_{min}}{xs_{max} - xs_{min}} & 0 & 0 & \frac{xt_{min}xs_{max} - xt_{max}xs_{min}}{xs_{max} - xs_{min}} \\ 0 & \frac{yt_{max} - yt_{min}}{ys_{max} - ys_{min}} & 0 & \frac{yt_{min}ys_{max} - yt_{max}ys_{min}}{ys_{max} - ys_{min}} \\ 0 & 0 & \frac{zt_{max} - zt_{min}}{zs_{max} - zs_{min}} & \frac{zt_{min}zs_{max} - zt_{max}zs_{min}}{zs_{max} - zs_{min}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Going back to our problem: we need to map the unified square $[-1,1] \times [-1,1]$ to the screen rectangle $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ what can be accomplished with windowing transformation

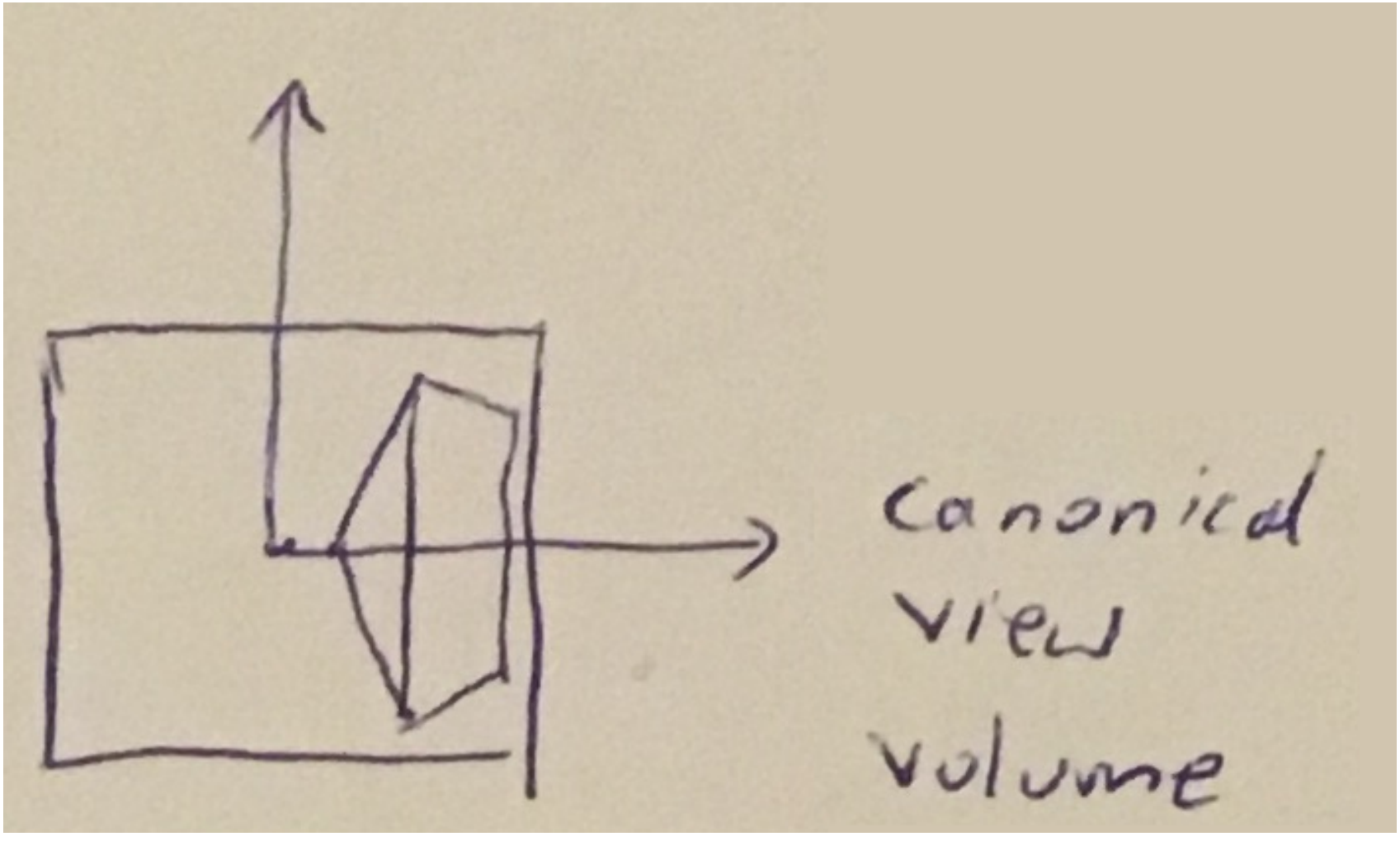
$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{unified}} \\ y_{\text{unified}} \\ 1 \end{bmatrix}$$

Note that M_{view} matrix ignores the z-coordinate of the points in the unified view volume, because a point's distance along the projection direction doesn't affect where that point projects in the image.
In spite of this, it's a good idea to keep information about z-coordinate without changing it. We can use the $z$values to make closer surfaces hide more distant surfaces.

$$M_{\text{view}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

M_{orth} — ortographic projection transformation to map points from any cube (view volume) to unified view volume where all visible points fall in the range from -1 to 1

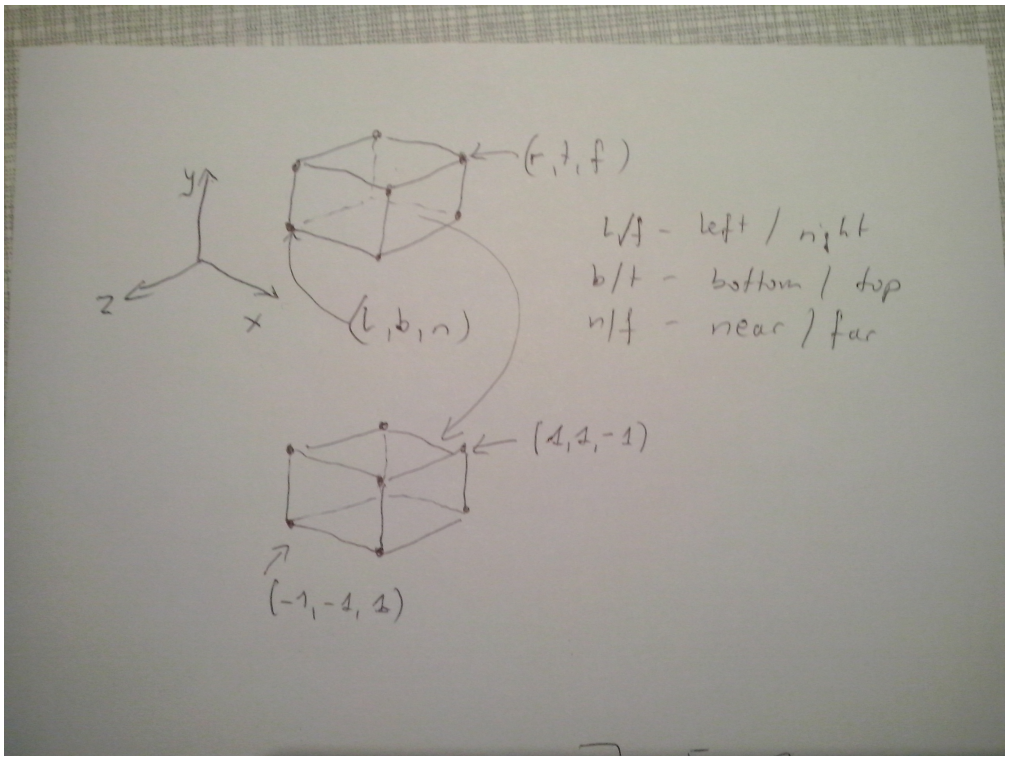
The orthographic projection transformation



The orthographic projection transformation

Idea

Of course, we usually want to render geometry in some region of space other than the unified (canonical) view volume. In other words, we have to map points from some arbitrary cube (volume) to unified volume $[-1, 1]^3$.



The orthographic projection transformation

Matrix form

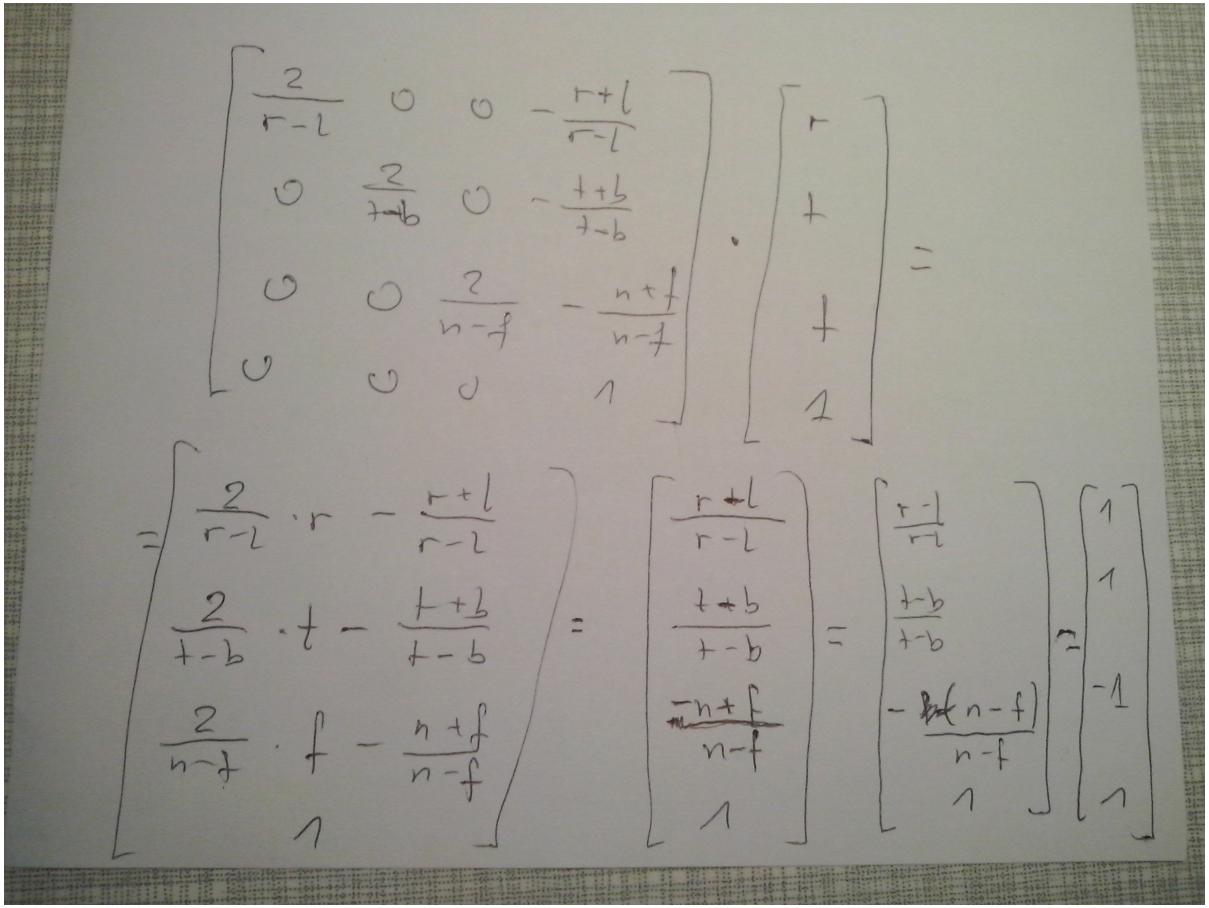
It's not difficult to check that the following matrix does this transformation

$$M_{\text{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The orthographic projection transformation

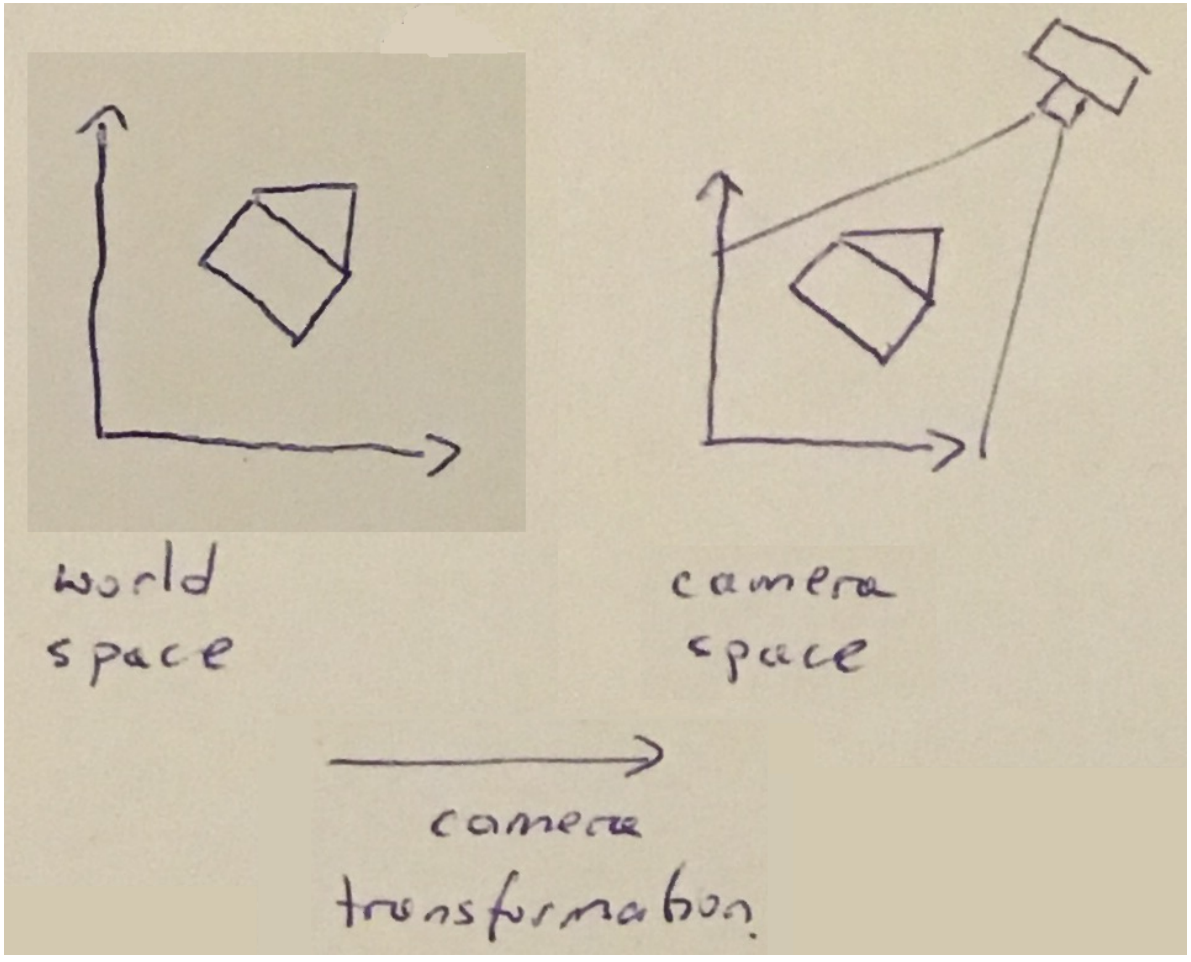
Example

Verify that the M_{orth} matrix transforms point from $[l, r] \times [b, t] \times [f, n]$ to $[-1, 1]^3$: for example point (r, t, f) shoud be transformed to $(1, 1, -1)$



M_{cam} — camera transformation

The camera transformation

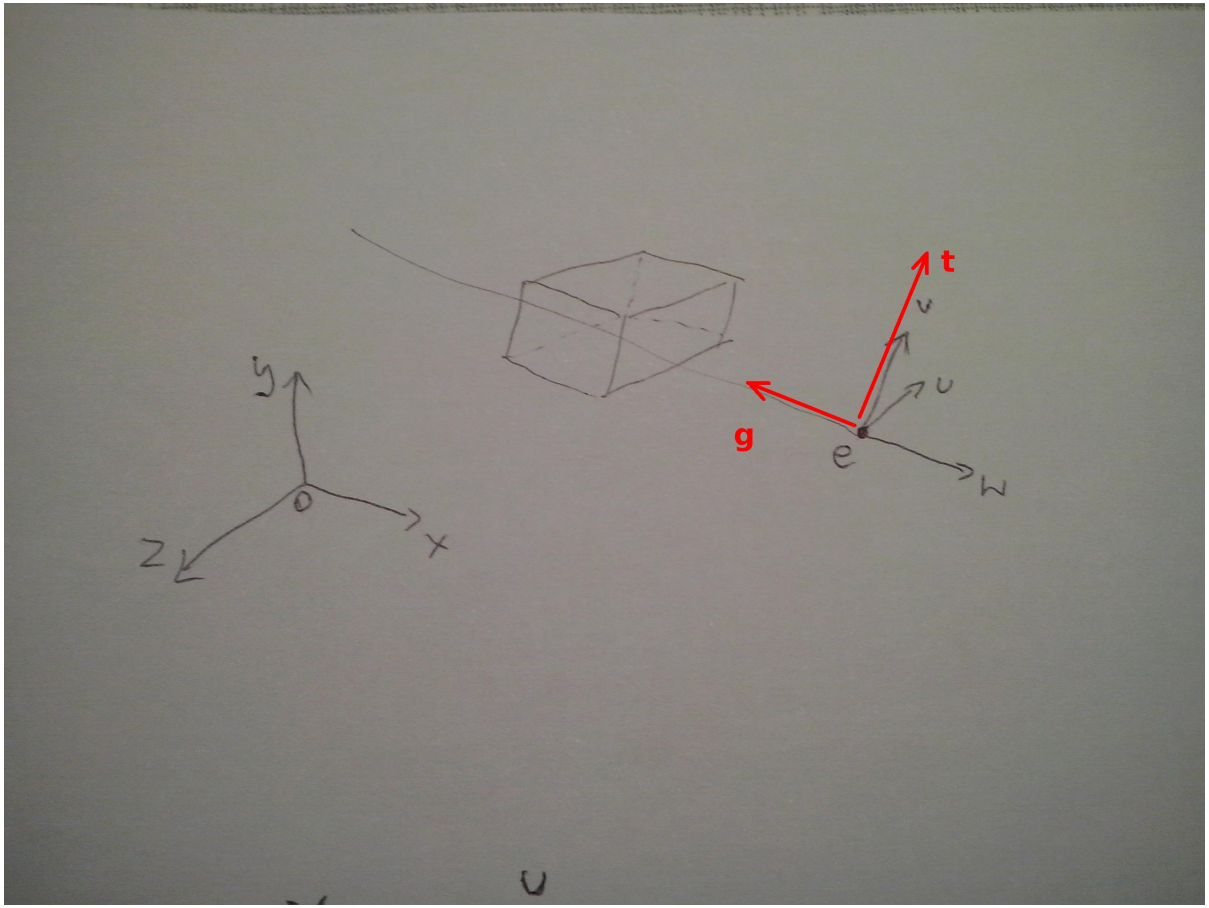


The camera transformation

We'd like to to change the viewpoint in 3D and look in any direction. There are a multitude of conventions for specifying viewer position and orientation. We will use the following one

- the eye position e,
- the gaze direction g,
- the view-up vector t.

The camera transformation



The camera transformation

Our job would be done if all points we wished to transform were stored in coordinates with origin e and some new basis vectors u, v, and w. As we can see, the coordinates of the model are stored in terms of the canonical (or world) origin o and the x-, y-, and z-axes. Therefore we need to convert the coordinates of the line segment endpoints we wish to draw from xyz-coordinates into uvw-coordinates.

The problem of coordinate system transformation and constructing coordinate system was discussed in *Basics of 2D and 3D graphics. Transformations* lecture.

The camera transformation

Construct coordinate system from vectors g and t

Using the construction we have described in *Basics of 2D and 3D graphics. Transformations* lecture, we have

$$w = -\frac{g}{||g||}$$

$$u = \frac{t \times w}{||t \times w||}$$

$$v = w \times u$$

The camera transformation

If we combine

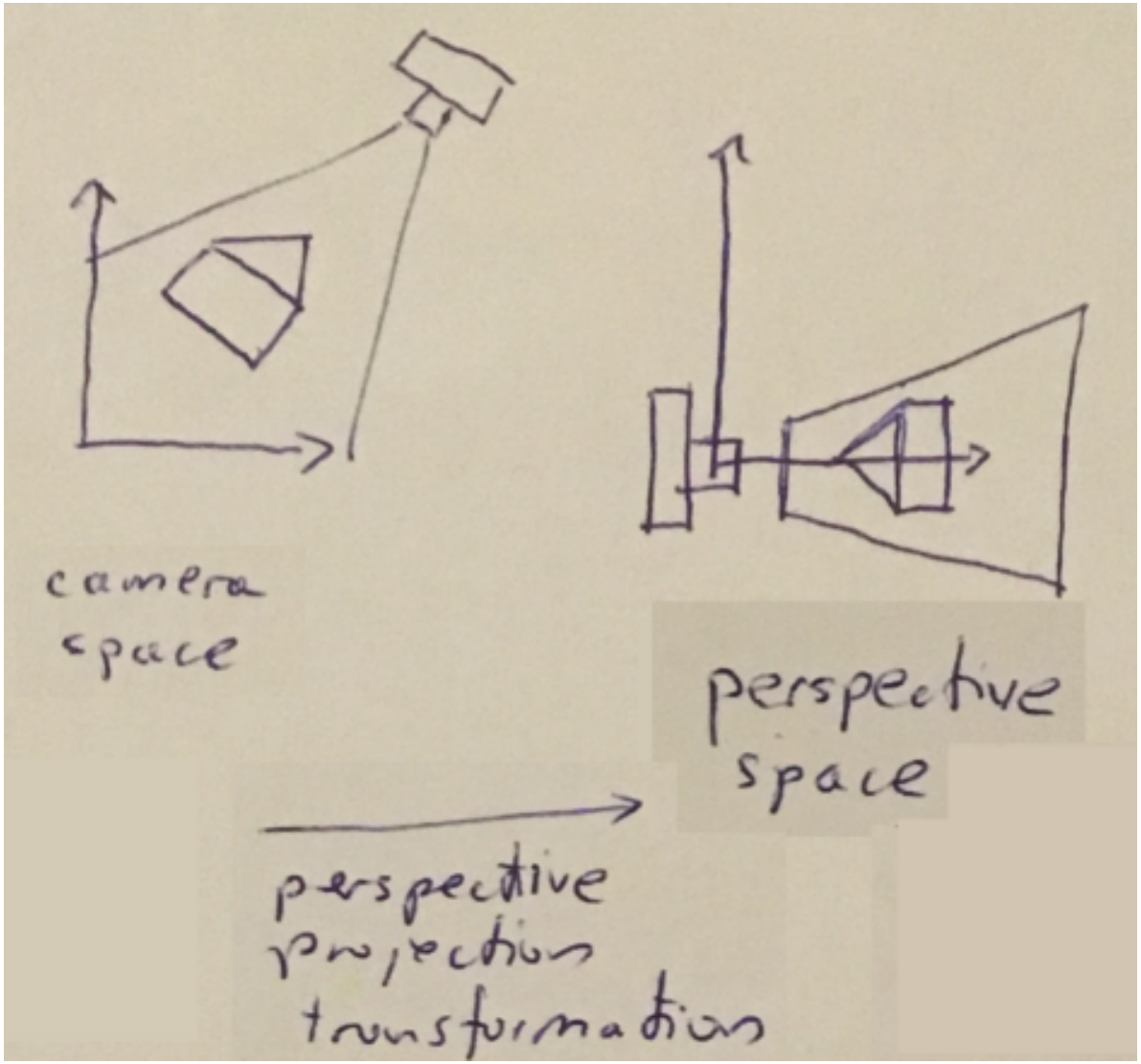
- general case for coordinate system transformation
- with new coordinate system *uvw* based on vectors g and t construction

we obtain

$$M_{\text{cam}} = \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

P — projective transformation

Projective transformations



To accomplish this transformation we have to use a concept of homogeneous coordinates we have discussed in *Basics of 2D and 3D graphics. Linear algebra* lecture.

Recall one of the homogeneous coordinates definition given in *Basics of 2D and 3D graphics. Linear algebra* lecture.

Definition

Given a point $p = (x, y)$ on the Euclidean plane, for any non-zero real number w, the triple (xw, yw, w) is called a set of homogeneous coordinates for the point p. By this definition, **multiplying the three homogeneous coordinates by a common, non-zero factor gives a new set of homogeneous coordinates for the same point**. In particular, $(x, y, 1)$ is such a system of homogeneous coordinates for the point (x, y).

For example, the Cartesian point $(1, 2)$ can be represented in homogeneous coordinates as $(1, 2, 1)$ or $(2, 4, 2)$. The original Cartesian coordinates are recovered by dividing the first two positions by the third. Thus unlike Cartesian coordinates, **a single point can be represented by infinitely many homogeneous coordinates**.

Second definition we will use soon was given in terms of equivalence classes.

Definition

For non-zero element of R^3, define $(x_1, y_1, z_1) \sim (x_2, y_2, z_2)$ to mean there is a non-zero λ so that $(x_1, y_1, z_1) = (\lambda x_2, \lambda y_2, \lambda z_2)$. Then $\sim$ is an equivalence relation and the projective plane can be defined as the equivalence classes of $R^3 \setminus \{0\}$. If (x, y, z) is one of the elements of the equivalence class p then these are taken to be homogeneous coordinates of p.

Perspective projection

Idea

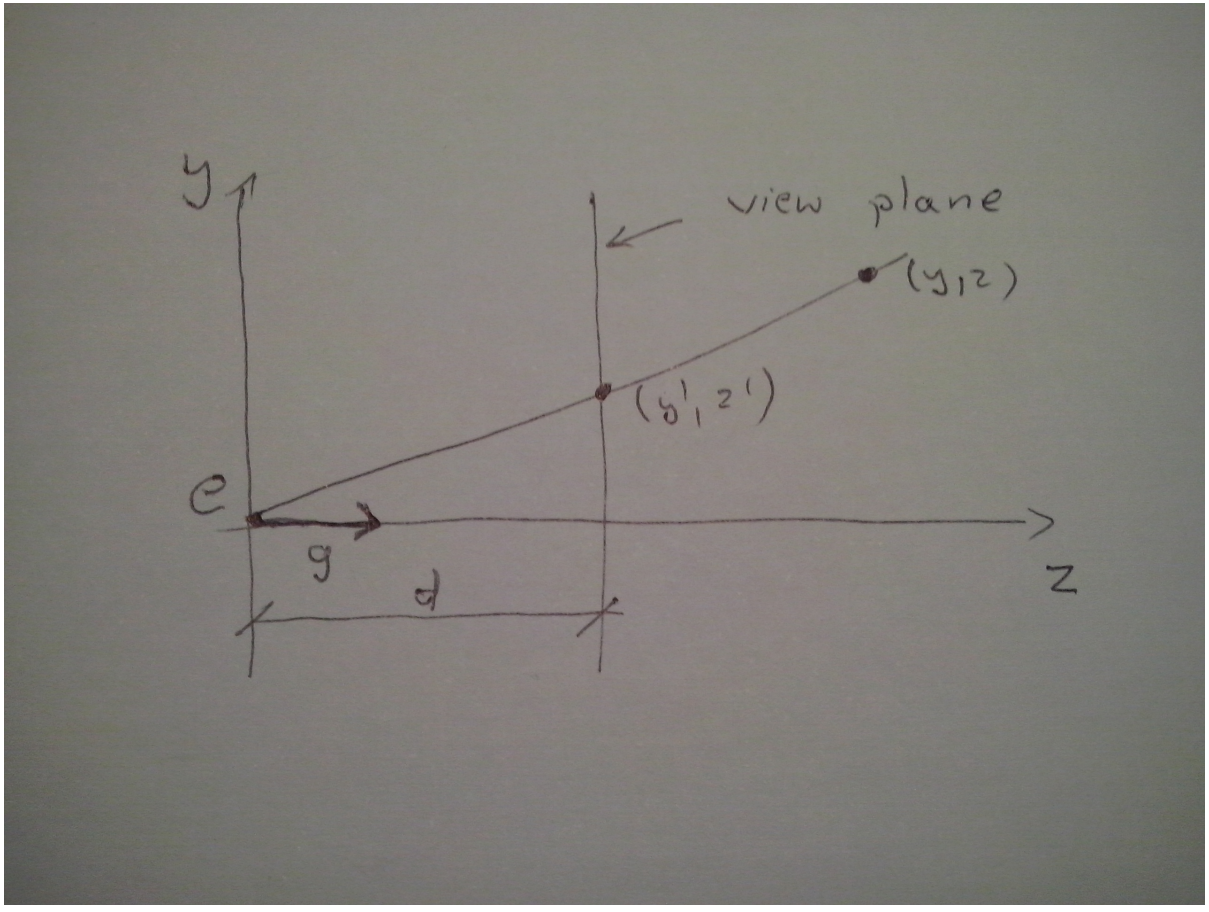
Let's see now why the homogeneous coordinates could be a right tool to solve our perspective projection problem. Summarize the environment assumption and what the perspective projection transformation needs to do with points in camera space.

- The viewpoint (the viewer's eye) e is positioned at the origin.
- The camera is looking along the z-axis. The gaze direction g direct into negative part of z-axis.
- The view plane is a distance d from viewpoint (the eye).
- A point p is projected toward e and where it intersects the view plane is where it is drawn. This is how we get p' point.

Perspective projection

Idea

Recall homogeneous coordinates example image and compare it with the following image



Note that with the above assumptions, the size of an object on the view plane (the screen) is proportional to $1/z$ for an eye at the origin looking up the negative z-axis. This can be expressed more precisely in an equation for the geometry

$$y' = \frac{d}{-z} y$$

So, the division by z is required to implement perspective.

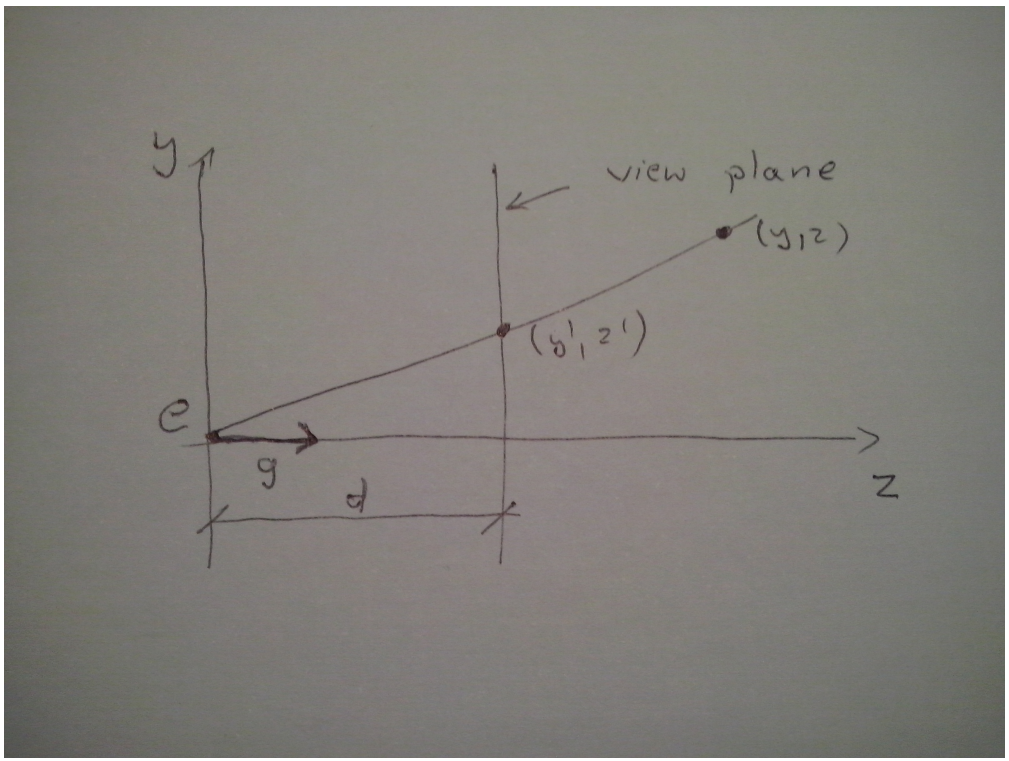
Now it should be clear why the mechanism of projective transformations and homogeneous coordinates makes it simple to implement the division by z required to implement perspective. This type of transformation, in which one of the coordinates of the input vector appears in the denominator, can't be achieved using affine transformations like translations, scaling or rotation.

Perspective projection

Idea

In the 2D example, we can implement the perspective projection with a matrix transformation as follows

$$\begin{bmatrix} \frac{dy}{z} \\ 1 \end{bmatrix} = \begin{bmatrix} y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} dy \\ z \end{bmatrix} = \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}$$



Following the above idea, the general perspective projection matrix in 3D can be as follow (we use n which means *near* to denote d; f means *far*)

$$P = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$P \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ z(n+f)-fn \\ z \end{bmatrix} \sim \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{bmatrix}$$

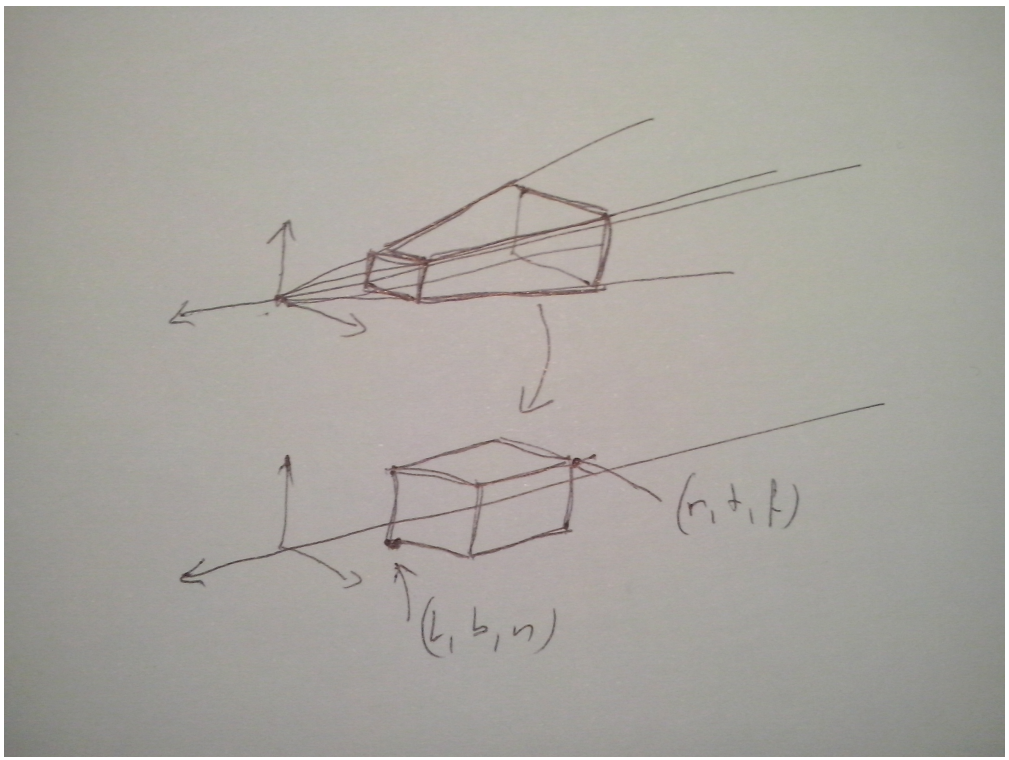
Perspective projection

The first, second, and fourth rows simply implement the perspective equation. A little bit odd is the third row. This row is designed to save somehow the z-coordinate so that we can use it later for hidden surface removal. In the perspective projection, though, the addition of a non-constant denominator (z) prevents us from actually preserving the value of z — it's actually impossible to keep z from changing while getting x and y to do what we need them to do. Instead we've opted to keep z unchanged for points on the near or far planes.
There are many matrices that could function as perspective matrices, and all of them non-linearly distort the z-coordinate. The matrix P has the nice properties: it leaves points on the near plane entirely alone, and it leaves points on the far plane while ,,squishing" them in x and y by the appropriate amount (see next slide). The transformation also preserves the relative order of z values between near and far plane, allowing us to do depth ordering after this matrix is applied. This will be important when we do hidden surface elimination.

Perspective projection

Properties

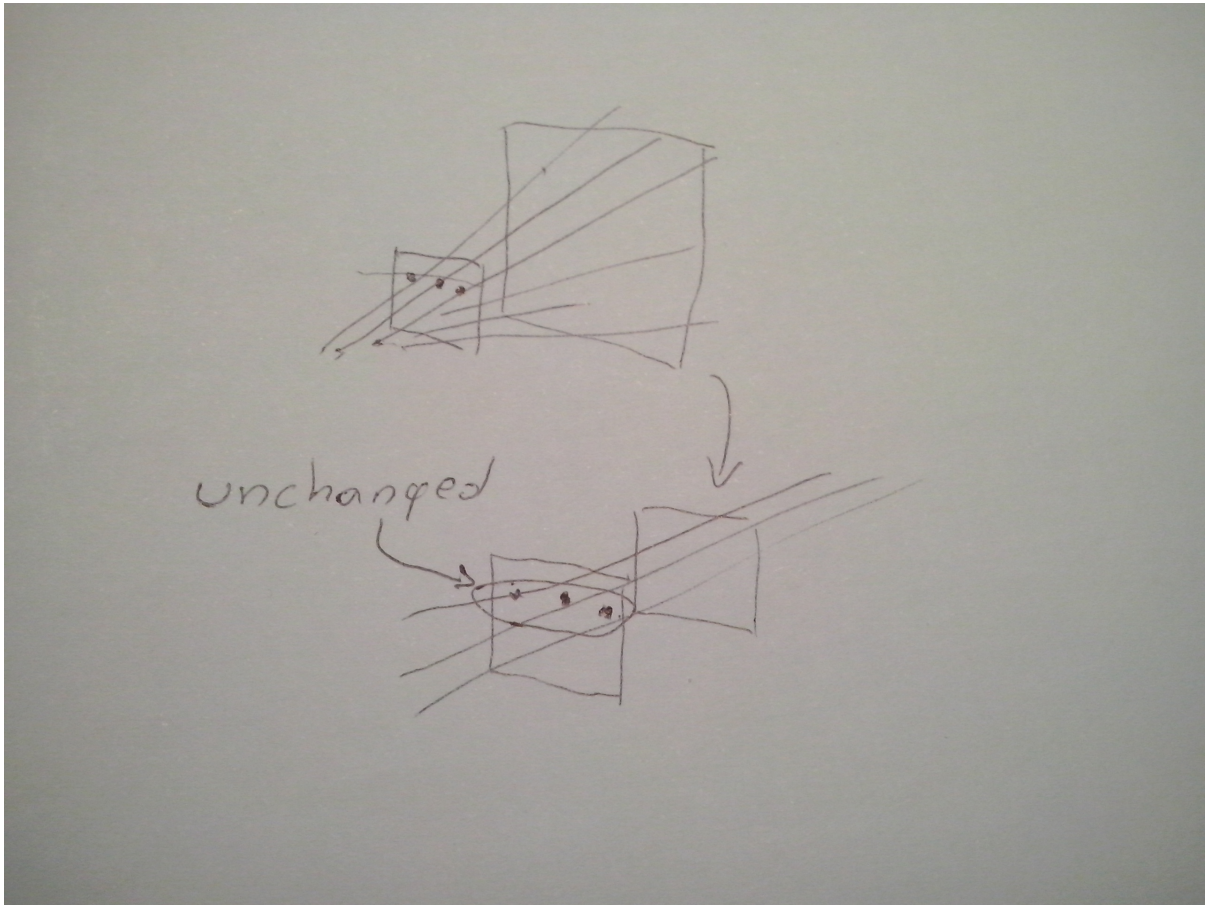
- The perspective projection leaves points on the near plane unchanged and maps the large far rectangle at the back of the perspective volume to the small far rectangle at the back of the orthographic volume.



Perspective projection

Properties

- The perspective projection maps any line through the origin (eye) to a line parallel to the z-axis and without moving the point on the line at near plane.



Now we know all the components of the finall prespective viewing matrix from the beginning of this lecture

$$M = M_v M_{\text{orth}} P M_{\text{cam}}$$

The values l, r, b and t are determined by the window throught which we look. Notice that sometimes matrices M_{orth} and P are combined into one matrix M_{per},

$$M_{\text{per}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

so the final matrix M takes the form

$$M = M_v M_{\text{per}} M_{\text{cam}}$$

In consequence we can express the finall algorithm as follow

```
compute matrix M
for each line segment (a_i, b_i) do
    p = Ma_i
    q = Mb_i
    draw line from (x_p/w_p, y_p/w_p) to (x_q/w_q, y_q/w_q)
```