

Teoria i praktyka programowania gier komputerowych

Viewing transformations

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

2 grudnia 2015

- 1 **Aim of this lecture**
- 2 M_{view} — **viewport transformation**
- 3 M_{orth} — **ortographic projection transformation**
- 4 M_{cam} — **camera transformation**
- 5 P — **projection transformation**
- 6 **Finall step**

Aim of this lecture

Viewing transformations 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 map 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.

Viewing transformations

The idea

We can break up this complicated process into 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**.

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 orthographic 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 specified point of the world and look at specified direction with specified orientation.

M_{view} — viewport transformation

Viewing transformations

Viewport transformation: unified view volume

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 camera looking in the $-z$ direction. So 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.

Viewing transformations

Viewport transformation: convention

We use integer numbers as pixel coordinates. Physical pixel has some dimensions and its 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 $[-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$.

Viewing transformations

Viewport transformation: windowing transformations – general case

Imagine that we need to create a transform matrix that takes points in the rectangle $[x_{s_{min}}, x_{s_{max}}] \times [y_{s_{min}}, y_{s_{max}}]$ to the rectangle $[x_{t_{min}}, x_{t_{max}}] \times [y_{t_{min}}, y_{t_{max}}]$. It's not difficult to note that this can be accomplished with two transformation in sequence: a single 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 $(x_{s_{min}}, y_{s_{min}})$ 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 $(x_{t_{min}}, y_{t_{min}})$.

Viewing transformations

Viewport transformation: windowing transformations – general case: step 1

Step 1: move source rectangle

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

$$T_{\text{source} \rightarrow \text{origin}} = T_{s_0} = \begin{bmatrix} 1 & 0 & -x_{s_{min}} \\ 0 & 1 & -y_{s_{min}} \\ 0 & 0 & 1 \end{bmatrix}$$

To verify this, let's take a point $p = (x, y)$, correct T_{s_0} matrix for it and check if the result of calculation (x_r, y_r) returns a point $(0, 0)$

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

Viewing transformations

Viewport transformation: windowing transformations – general case: step 2

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

$$T_{scale} = T_s = \begin{bmatrix} \frac{x^t_{max} - x^t_{min}}{x^s_{max} - x^s_{min}} & 0 & 0 \\ 0 & \frac{y^t_{max} - y^t_{min}}{y^s_{max} - y^s_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Viewing transformations

Viewport transformation: windowing transformations – general case: step 3

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

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

Viewing transformations

Viewport transformation: windowing transformations – general case: final window transformation for 2D case

$$T_w = T_{ot} T_s T_{s0} = \begin{bmatrix} \frac{x_{t_{max}} - x_{t_{min}}}{x_{s_{max}} - x_{s_{min}}} & 0 & \frac{x_{t_{min}} x_{s_{max}} - x_{t_{max}} x_{s_{min}}}{x_{s_{max}} - x_{s_{min}}} \\ 0 & \frac{y_{t_{max}} - y_{t_{min}}}{y_{s_{max}} - y_{s_{min}}} & \frac{y_{t_{min}} y_{s_{max}} - y_{t_{max}} y_{s_{min}}}{y_{s_{max}} - y_{s_{min}}} \\ 0 & 0 & 1 \end{bmatrix}$$

Viewing transformations

Viewport transformation: windowing transformations – general case: finall widow transformation for 3D case

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

$$T_w = \begin{bmatrix} \frac{x_{t_{max}} - x_{t_{min}}}{x_{s_{max}} - x_{s_{min}}} & 0 & 0 & \frac{x_{t_{min}} x_{s_{max}} - x_{t_{max}} x_{s_{min}}}{x_{s_{max}} - x_{s_{min}}} \\ 0 & \frac{y_{t_{max}} - y_{t_{min}}}{y_{s_{max}} - y_{s_{min}}} & 0 & \frac{y_{t_{min}} y_{s_{max}} - y_{t_{max}} y_{s_{min}}}{y_{s_{max}} - y_{s_{min}}} \\ 0 & 0 & \frac{z_{t_{max}} - z_{t_{min}}}{z_{s_{max}} - z_{s_{min}}} & \frac{z_{t_{min}} z_{s_{max}} - z_{t_{max}} z_{s_{min}}}{z_{s_{max}} - z_{s_{min}}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing transformations

Viewport transformation: solution

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

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

Viewing transformations

Viewport transformation: solution

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.

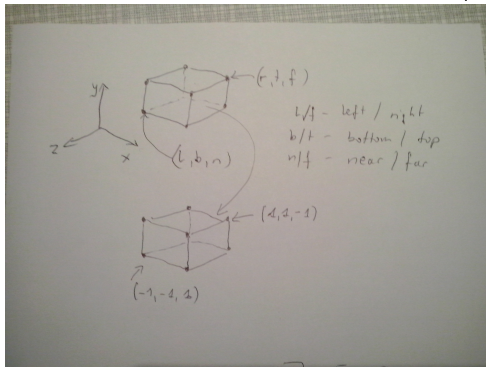
$$M_{\text{view}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

M_{orth} — orthographic 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

Viewing transformations

The orthographic projection transformation

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 $[-1, 1]^3$.

Viewing transformations

The orthographic projection transformation

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

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

Viewing transformations

The orthographic projection transformation

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) should be transformed to $(1, 1, -1)$

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r \\ t \\ f \\ 1 \end{bmatrix} =$$
$$= \begin{bmatrix} \frac{2}{r-l} \cdot r - \frac{r+l}{r-l} \\ \frac{2}{t-b} \cdot t - \frac{t+b}{t-b} \\ \frac{2}{n-f} \cdot f - \frac{n+f}{n-f} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{r+l}{r-l} \\ \frac{t+b}{t-b} \\ \frac{-n+f}{n-f} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{r-l}{r-l} \\ \frac{t+b}{t-b} \\ \frac{-\cancel{2}(n-f)}{n-f} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

M_{cam} — camera transformation

Viewing transformations

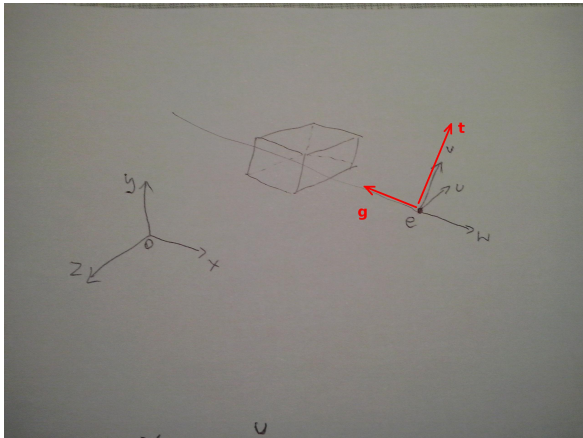
The camera transformation

We'd like to be able 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 .

Viewing transformations

The camera transformation



Viewing transformations

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.

Viewing transformations

The camera transformation: coordinate system transformation

From geometric point of view a coordinate system (or coordinate frame) consists of an origin and a basis which is a set of vectors. A basis in most cases is orthonormal (which means that vectors are orthonormal, that is, they are all unit vectors and orthogonal to each other).

In 2D case with origin e and basis $\{u, v\}$, the coordinates (u_p, v_p) describe the point

$$p = e + u_p u + v_p v.$$

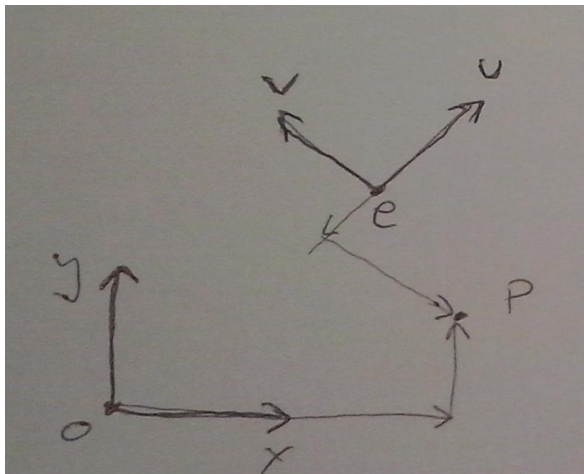
Similarly, we can express point p in terms of another coordinate system

$$p = o + x_p x + y_p y$$

(see next slide).

Viewing transformations

The camera transformation: coordinate system transformation



Viewing transformations

The camera transformation: coordinate system transformation

We can express this relationship using matrix transformation

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

Note that this assumes we have the point e and vectors u and v stored in some canonical coordinates which are in this case from the (x, y) -coordinate system.

Viewing transformations

The camera transformation: coordinate system transformation

In most cases we write this matrix like this

$$p_{xy} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} p_{uv}$$

It takes points expressed in the (u, v) coordinate system and converts them to the same points expressed in the (x, y) coordinate system (but (u, v) coordinate system has to be described in the (x, y) coordinate system terms).

Viewing transformations

The camera transformation: coordinate system transformation

Consider a following example:

$$e = (2, 2)$$

$$u = (1, 0)$$

$$v = (0, -1)$$

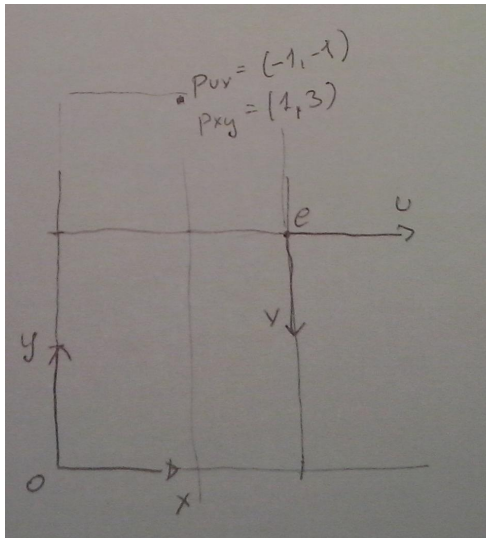
$$p_{uv} = (-1, -1)$$

so

$$p_{xy} = \begin{array}{c|ccc|} & 0 & 1 & 2 & \\ \hline & -1 & 0 & 2 & \\ \hline & 0 & 0 & 1 & \end{array} p_{uv} \Rightarrow p_{xy} = (1, 3, 1)$$

Viewing transformations

The camera transformation: coordinate system transformation



Viewing transformations

The camera transformation: coordinate system transformation

In 3D case we have

$$P_{xyz} = \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix} P_{uvw}$$

Viewing transformations

The camera transformation: coordinate system transformation

Notice that if we need transformation from xyz -coordinate system to uvw -coordinate system

$$P_{xyz} = \begin{bmatrix} u & v & w & e \\ 0 & 0 & 0 & 1 \end{bmatrix} P_{uvw}$$

Viewing transformations

The camera transformation: general case for constructing coordinate system

The cross product $a \times b$ is defined as a vector c that is perpendicular to both a and b , with a direction given by the right-hand rule and a magnitude equal to the area of the parallelogram that the vectors span. The cross product is defined by the formula

$$a \times b = (\|a\| \|b\| \sin \phi) n,$$

where ϕ is the angle between a and b in the plane containing them, $\|a\|$ and $\|b\|$ are the magnitudes of vectors a and b , and n is a unit vector perpendicular to the plane containing a and b in the direction given by the right-hand rule if we use right-handed coordinate system. If a left-handed coordinate system is used, the direction of the vector n is given by the left-hand rule and points in the opposite direction.

In a right-handed system, we take our right hand and create a 90 angle between our thumb and index finger. Next we likewise make a 90 angle between our index finger and middle finger. We then line up our index finger with a and middle finger with b . The direction our thumb points in is the direction the cross product will face.

Viewing transformations

The camera transformation: general case for constructing coordinate system

We can calculate vector c as

$$c = [a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x].$$

Viewing transformations

The camera transformation: general case for constructing coordinate system

We can calculate orthonormal basis that is aligned with a given vector. That is, given a vector a , we want an orthonormal u , v , and w such that w points in the same direction as a .

This can be done using cross products as follows.

First make w a unit vector in the direction of a :

$$w = \frac{a}{\|a\|}$$

Then choose any vector t not collinear with w , and use the cross product to build a unit vector u perpendicular to w :

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

Once w and u are in hand, completing the basis is simple:

$$v = w \times u.$$

Viewing transformations

The camera transformation: construct coordinate system from vectors g and t

Using the construction we have just described we have

$$w = - \frac{g}{\|g\|}$$
$$u = \frac{t \times w}{\|t \times w\|}$$
$$v = w \times u$$

Viewing transformations

The camera transformation

If we combine

- general case for coordinate system transformation
- with new coordinate system from 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 — projection transformation

Perspective projection

Projective transformations: homogeneous coordinates

Homogeneous coordinates (pl. *współrzędne jednorodne*) (or **projective coordinates** are a system of coordinates used in projective geometry (pl. *geometrii rzutowej*, as Cartesian coordinates are used in Euclidean geometry. They have the advantage that **the coordinates of points, including points at infinity, can be represented using finite coordinates**. Formulas involving homogeneous coordinates are often simpler and more symmetric than their Cartesian counterparts. Homogeneous coordinates have a range of applications, including computer graphics and 3D computer vision, where they allow affine transformations and, in general, projective transformations to be easily represented by a matrix.

Perspective projection

Projective transformations: homogeneous coordinates

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**.

Perspective projection

Projective transformations: homogeneous coordinates

Another definition of the real projective plane can be given in terms of equivalence classes.

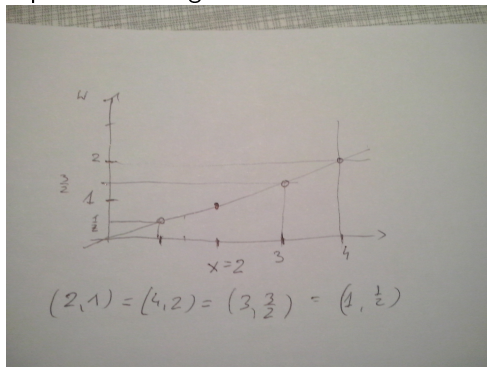
Definition

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

Projective transformations: homogeneous coordinates

Of course both previous definitions are equivalent, but focus on different aspects of homogeneous coordinates. Let's consider a following example.



The point $x = 2$ is represented by any point on the line $x = 2w$. However, before we interpret x as a conventional Cartesian coordinate, we first divide by w to get $(x, w) = (2, 1)$.

Viewing transformations

Perspective projection

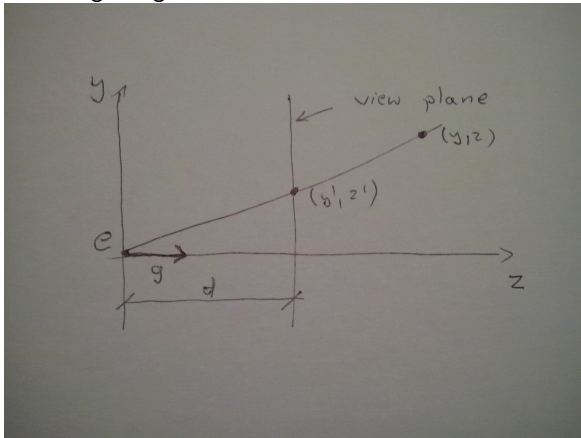
Let's see now why the homogeneous coordinates could be a right tool to solve our perspective projection problem. Summarize the environment assumption and what the perspective projection transformation needs to do with points in camera space.

- 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.

Viewing transformations

Perspective projection

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



Viewing transformations

Perspective projection

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.

Viewing transformations

Perspective projection

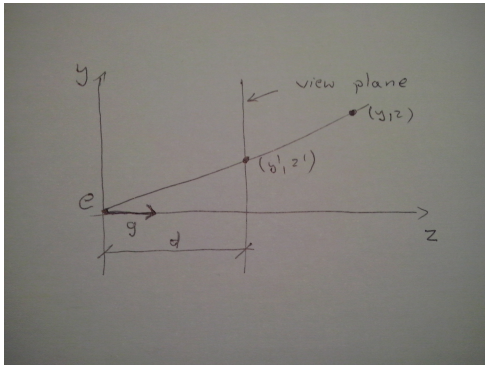
Now it should be clear why the mechanism of projective transformations and homogeneous coordinates makes it simple to implement the division by z required to implement perspective. This type of transformation, in which one of the coordinates of the input vector appears in the denominator, can't be achieved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

Viewing transformations

Perspective projection

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

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



Viewing transformations

Perspective projection

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*)

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

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

Viewing transformations

Perspective projection

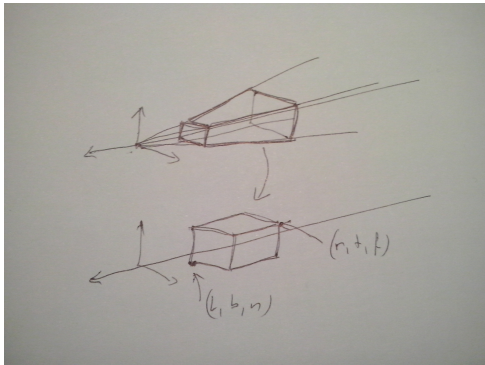
The first, second, and fourth rows simply implement the perspective equation. A little bit odd is the third row. This row is designed to save somehow the z -coordinate so that we can use it later for hidden surface removal. In the perspective projection, though, the addition of a non-constant denominator (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.

Viewing transformations

Perspective projection: properties

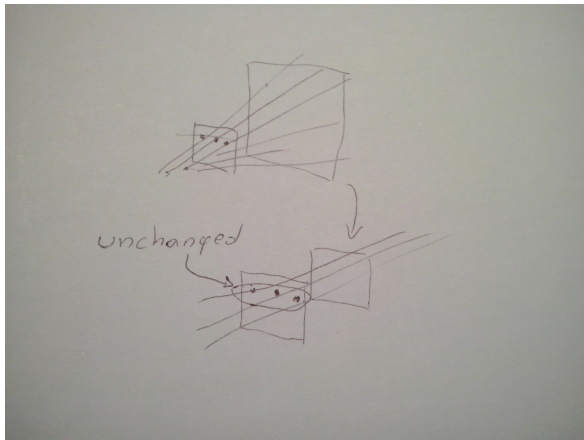
- The perspective projection leaves points on the near plane unchanged and maps the large far rectangle at the back of the perspective volume to the small far rectangle at the back of the orthographic volume.



Viewing transformations

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.



Viewing transformations

Final step

Now we know all the components of the final perspective 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 through 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 final 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)
```