Perspective Projection

Slides Adapted from Wolfgang Hurst
Assistant Professor Game and Media
Utrecht University

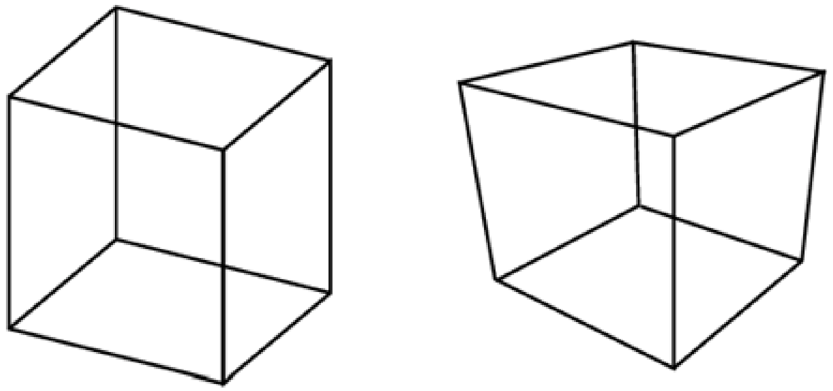
Perspective

Goal: create 2D images of 3D scenes

Standard approach: linear perspective,
i.e. straight lines in the scene become straight lines in the image
(in contrast to, e.g., fisheye views)

Two important distinctions:

- parallel projection
- perspective projection



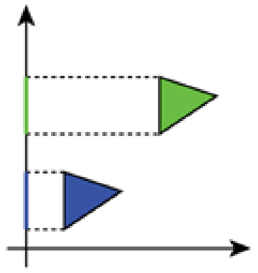
Parallel projection

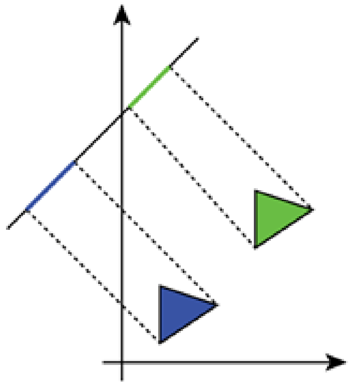
Maps 3D points to 2D by moving them along a projection direction until they hit an image plane

- image plane perpendicular to viewing direction: orthographic
- otherwise: oblique
- (note: other definitions exist)

Characteristics:

- keep parallel lines parallel
- preserve size and shape of planar objects





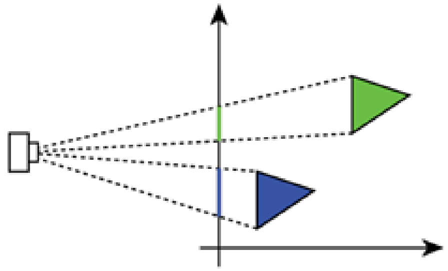
Perspective projection

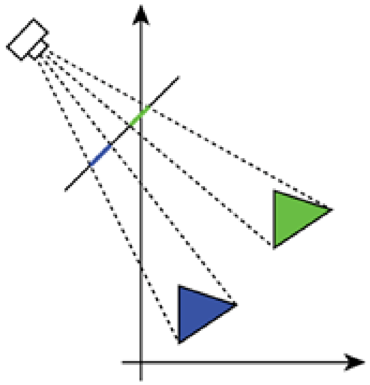
Maps 3D points to 2D by projecting them along lines that pass trought a single viewpoint until they hit an image plane

- distinction between oblique and non-oblique based on projection direction at the center of the image

Characteristics:

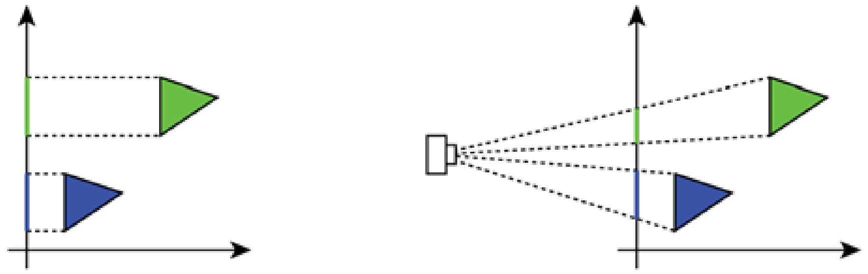
- objects farther from the viewpoint naturally become smaller





Parallel vs. perspective projection

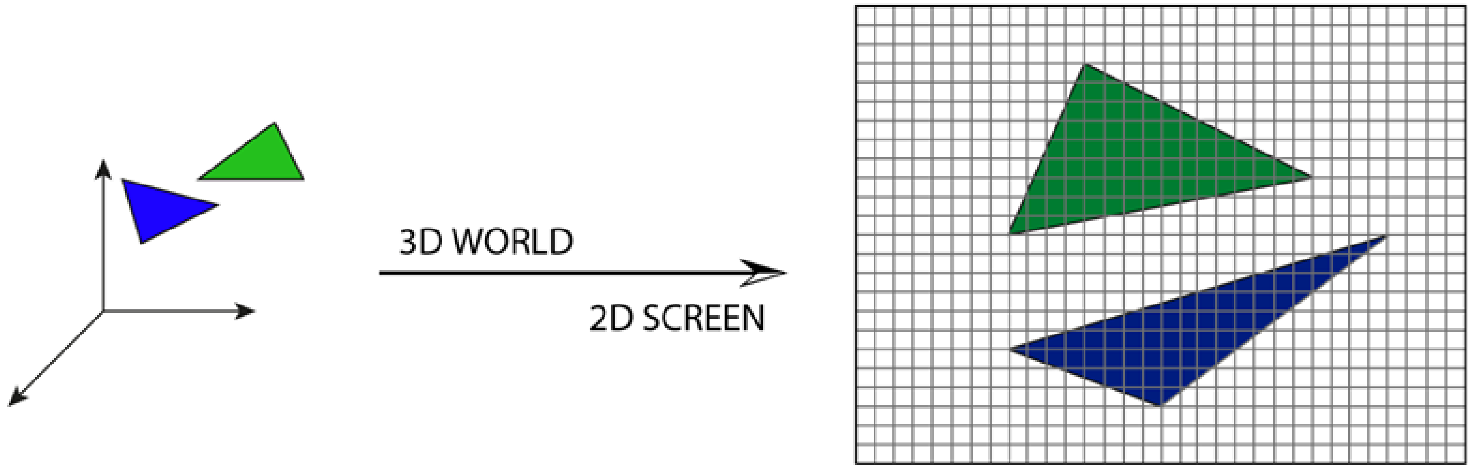
- Perspective projection: more natural and realistic
- Parallel: usage in mechanical and architectural drawings



- How to get 3D objects perspectively correct on 2D screen?
- Note: usually your API takes care of most of this, but it's good to know what's going on behind those function calls
- And it's a good opportunity to improve your maths skills ;)

Perspective projection

How to get 3D objects perspectively correct on 2D screen?

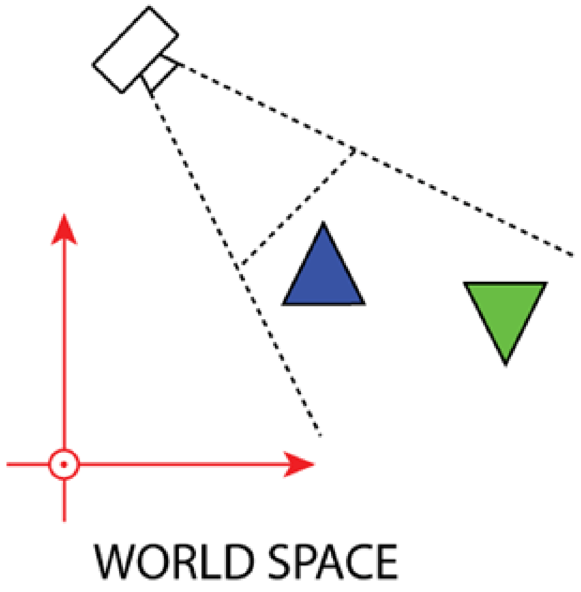


This task is best solved by splitting it in subtasks that in turn can be solved by matrix multiplication

Let's start with what we got ...

World space

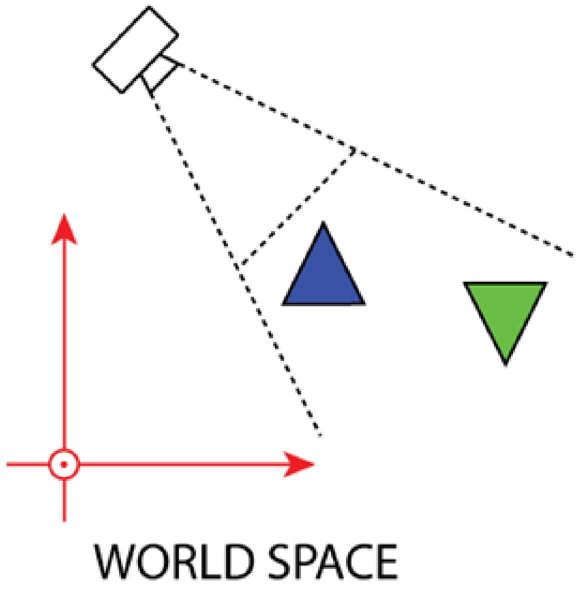
- Our 3D scene is given in world space, i.e. linear combinations of the base vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$
- Given an arbitrary camera position, we want to display our 3D world in a 2D image using perspective projection



Camera position

The camera position is specified by

- the eye vector $\vec{e}$
  (it's location)
- the gaze vector $\vec{g}$
  (it's direction)
- the image plane
  (it's field of view (FOV) and distance)

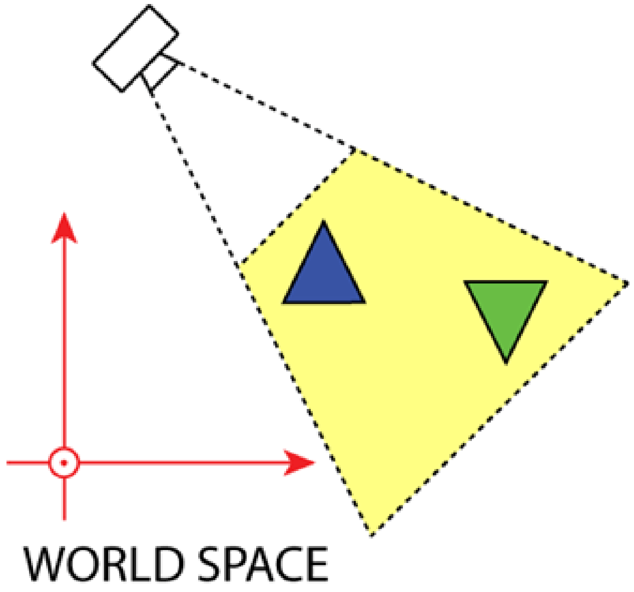


View frustum

The view frustum (aka view volume) specifies everything that the camera can see. It's defined by

- the left plane l
- the right plane r
- the top plane t
- the bottom plane b
- the near plane n
- the far plane f

Note: for now, let's assume all our objects are completely within the view frustum

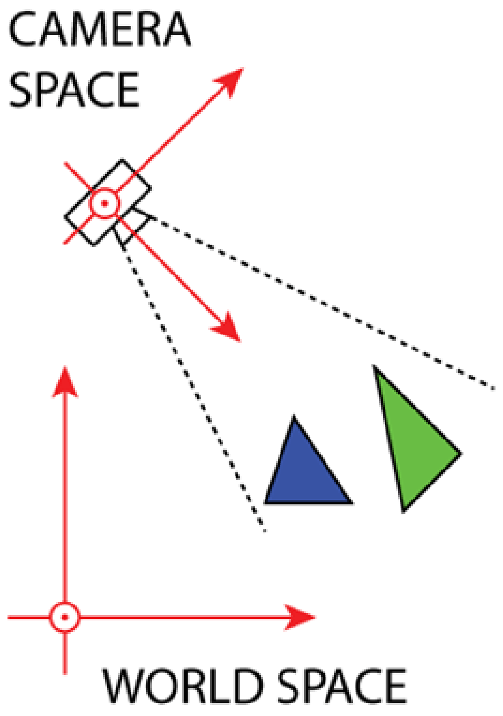


Camera transformation

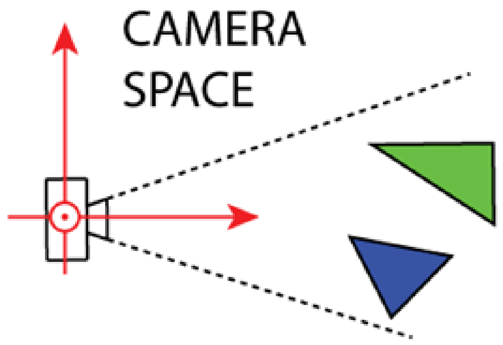
Hmm, it would be much easier if the camera were at the origin ...

We can do that by moving from world space coordinates to camera space coordinates.

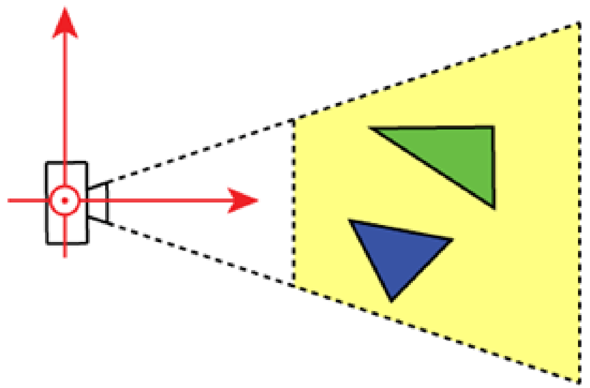
This is just a simple matrix multiplication (cf. later).



Camera transformation



Per convention, we look into the direction of the negative Z-axis

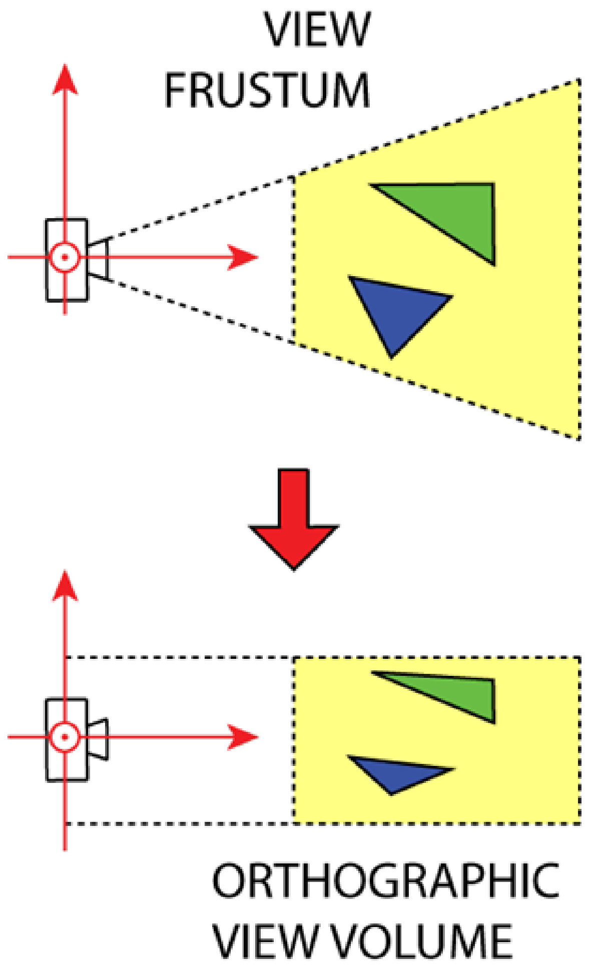


Orthographic projection

Hmm, it would be much easier if we could do parallel projection . . .

We can do that by transforming the view frustum to the orthographic view volume.

Again, this is just a matrix multiplication (but this time, it's not that simple, cf. later).

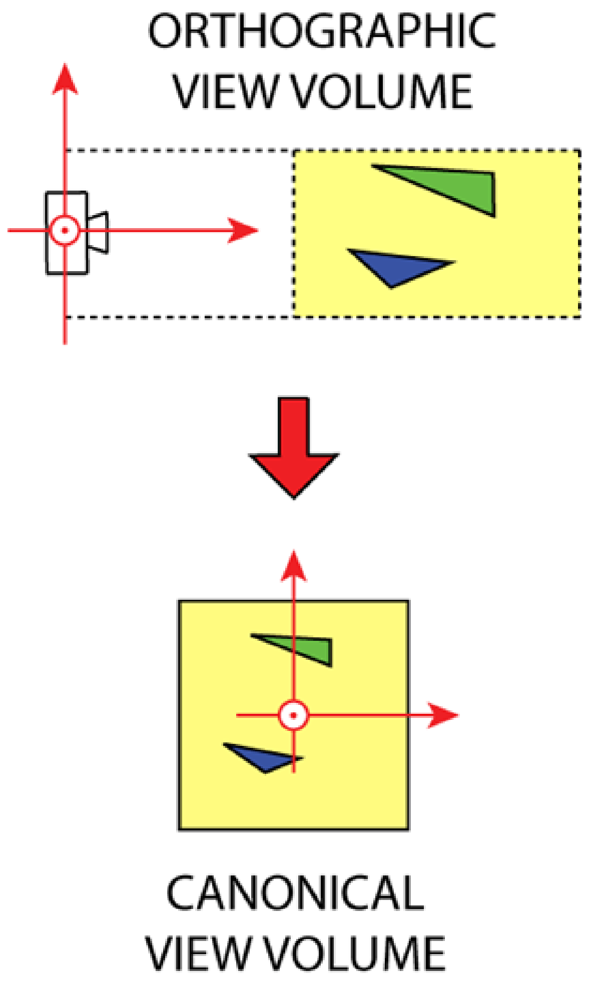


The canonical view volume

Hmm, it would be much easier if our values were between -1 and 1 . . .

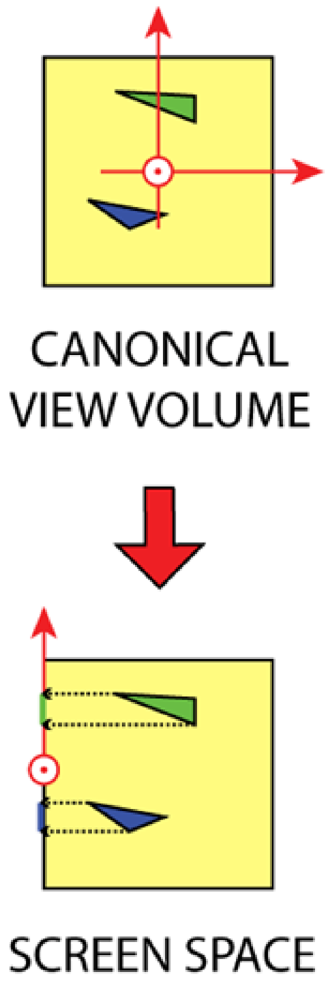
We can do that by transforming the orthographic view volume to the canonical view volume.

Again, this is just a (simple) matrix multiplication (cf. later).



Windowing transform

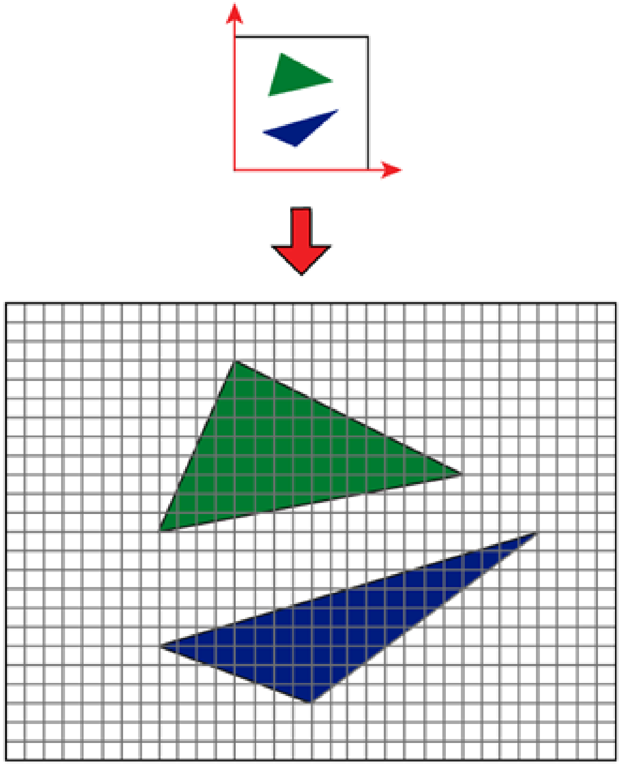
Now all that's left is a parallel projection along the Z-axis (every easy) and ...



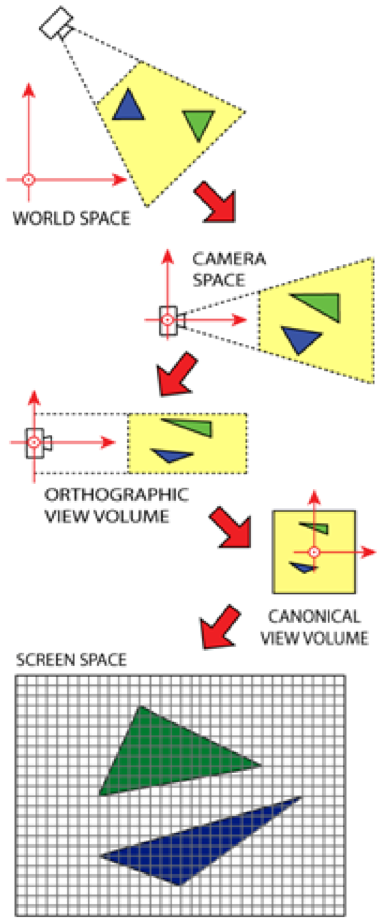
Windowing transform

. . . a windowing transformation in order to display the square $[-1, 1]^2$ onto an $n_x \times n_y$ image.

Again, these are just some (simple) matrix multiplications (cf. later).



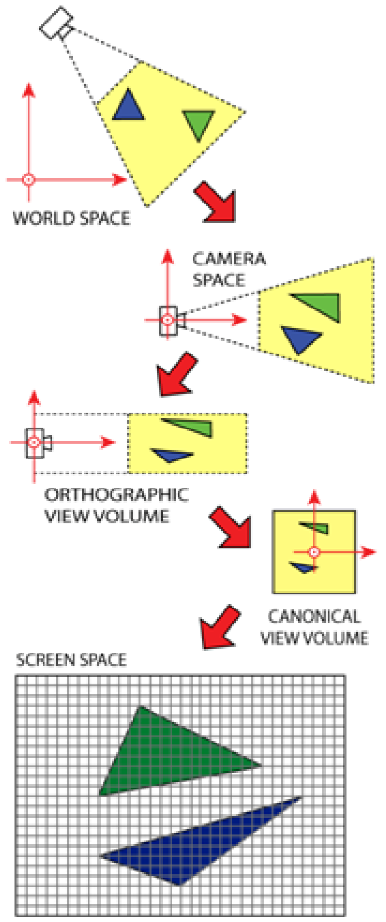
The graphics pipeline (part I)



Notice that every step in this sequence can be represented by a matrix operation, so the whole process can be applied by performing a single matrix operation! (well, almost . . .)

We call this sequence a graphics pipeline = a special software or hardware subsystem that efficiently draws 3D primitives in perspective.

Overview



Let's start with the easier stuff, e.g.

Windowing transformation
(aka viewport transformation)

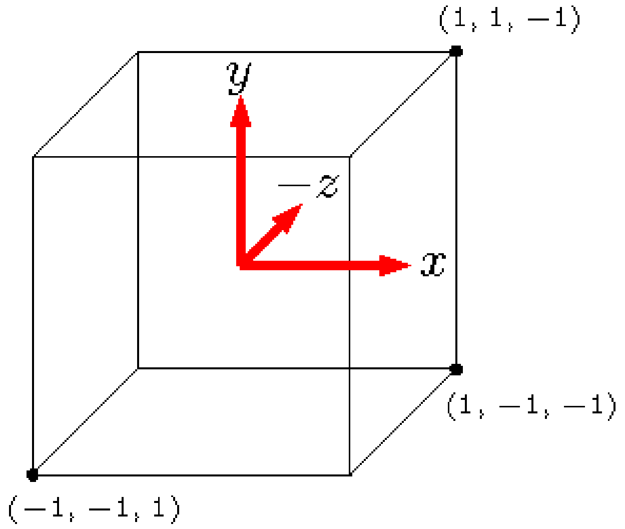
How do we get the data from the canonical view volume to the screen?

The canonical view volume

The canonical view volume is a $2 \times 2 \times 2$ box, centered at the origin.

The view frustum is transformed to this box (and the objects within the view frustum undergo the same transformation).

Vertices in the canonical view volume are orthographically projected onto an $n_x \times n_y$ image.

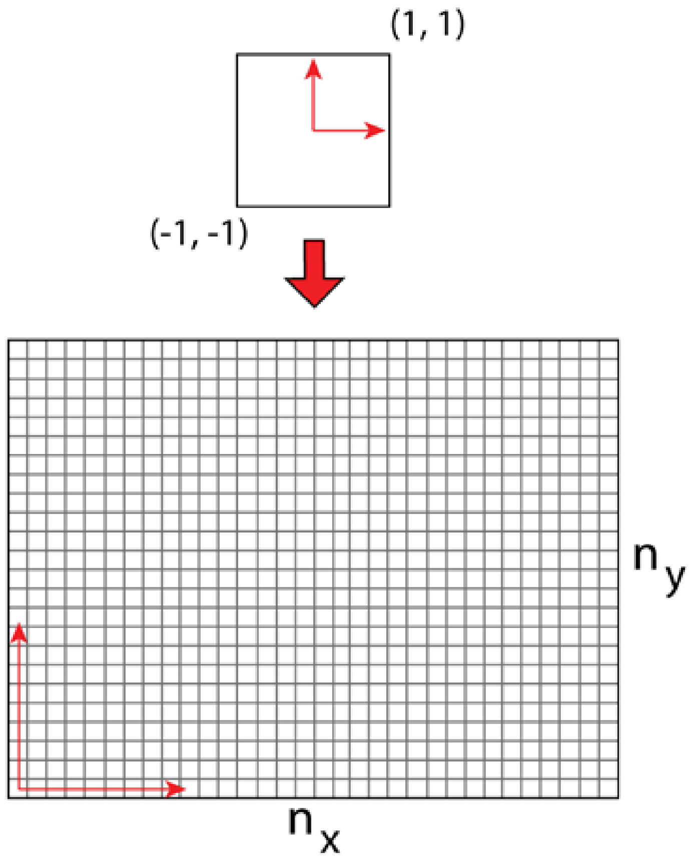


Mapping the canonical view volume

We need to map the square $[-1, 1]^2$ onto a rectangle $[0, n_x] \times [0, n_y]$.

The following matrix takes care of that:

$$\begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

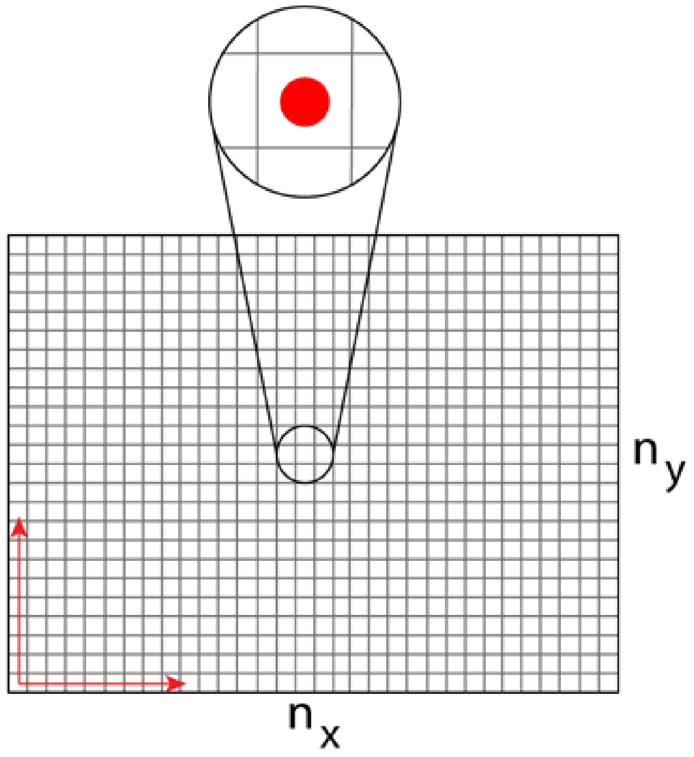


Mapping the canonical view volume

In practice, pixels represent unit squares centered at integer coordinates, so we actually have to map to the rectangle $[-\frac{1}{2}, n_x - \frac{1}{2}] \times [-\frac{1}{2}, n_y - \frac{1}{2}]$.

Hence, our matrix becomes:

$$\begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x}{2} - \frac{1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y}{2} - \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$



Mapping the canonical view volume

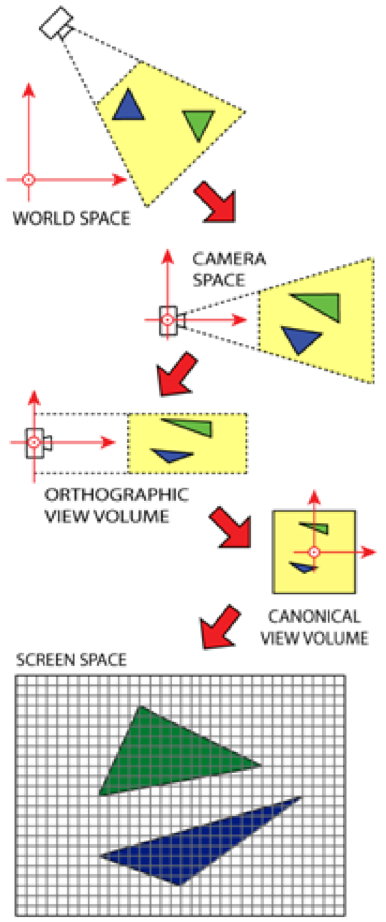
Notice that we did orthographic projection by "throwing away " the z-coordinate.

But since we want to combine all matrices in the end, we need a 4×4 matrix, so we add a row and colum that "doesn't change z".

Our final matrix for the windowing or viewport transformation is

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} - \frac{1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} - \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Overview



Hence, our last step will be

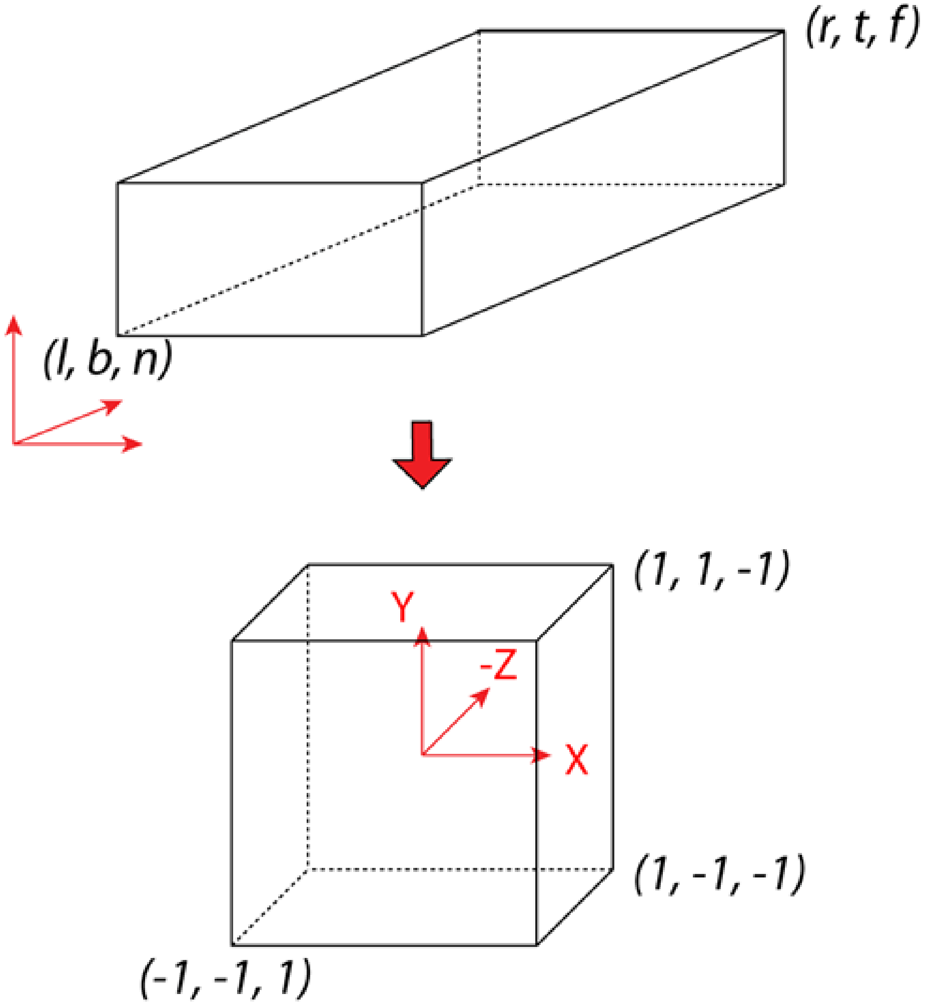
$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} \begin{pmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{pmatrix}$$

Ok, now let's work our way up:

How do we get the data from the orthographic view volume to the canonical view volume, i.e. ...

The orthographic view volume

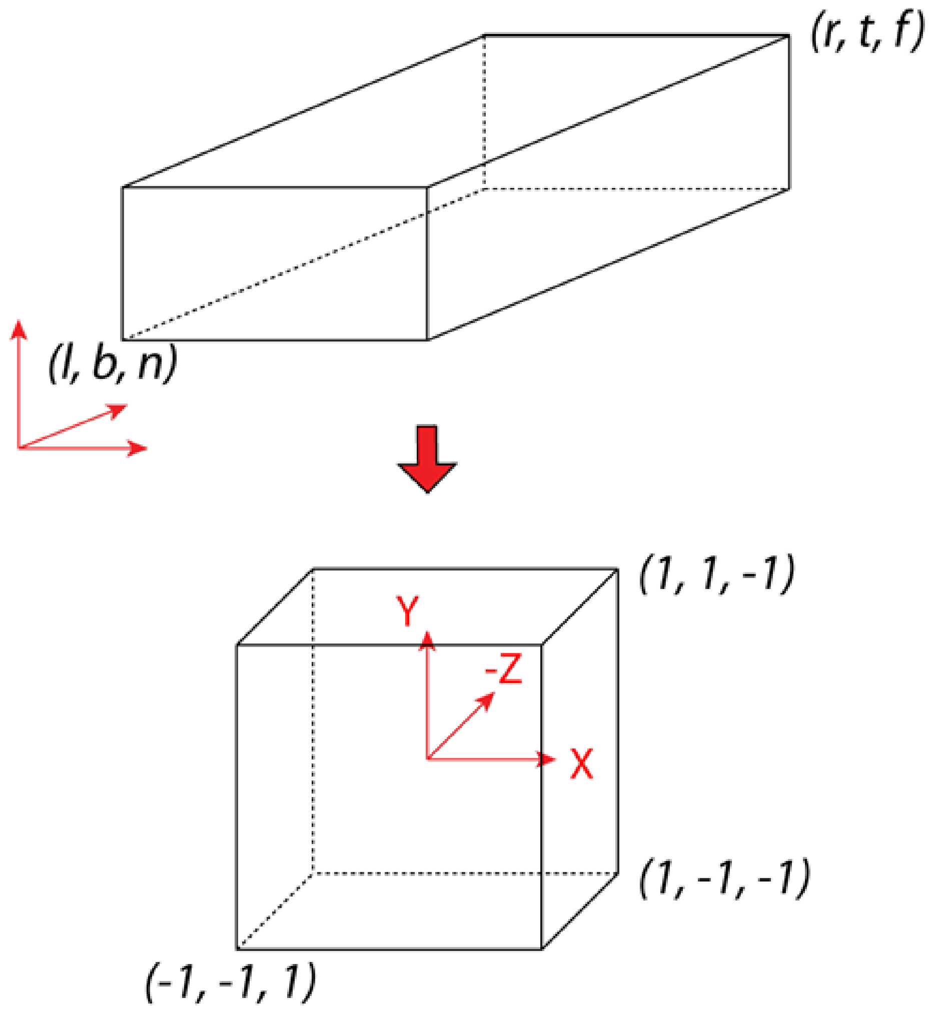
. . . how do we get the data from the axis-aligned box $[l, r] \times [b, t] \times [n, f]$ to a $2 \times 2 \times 2$ box around the origin?



The orthographic view volume

First we need to move the center to the origin:

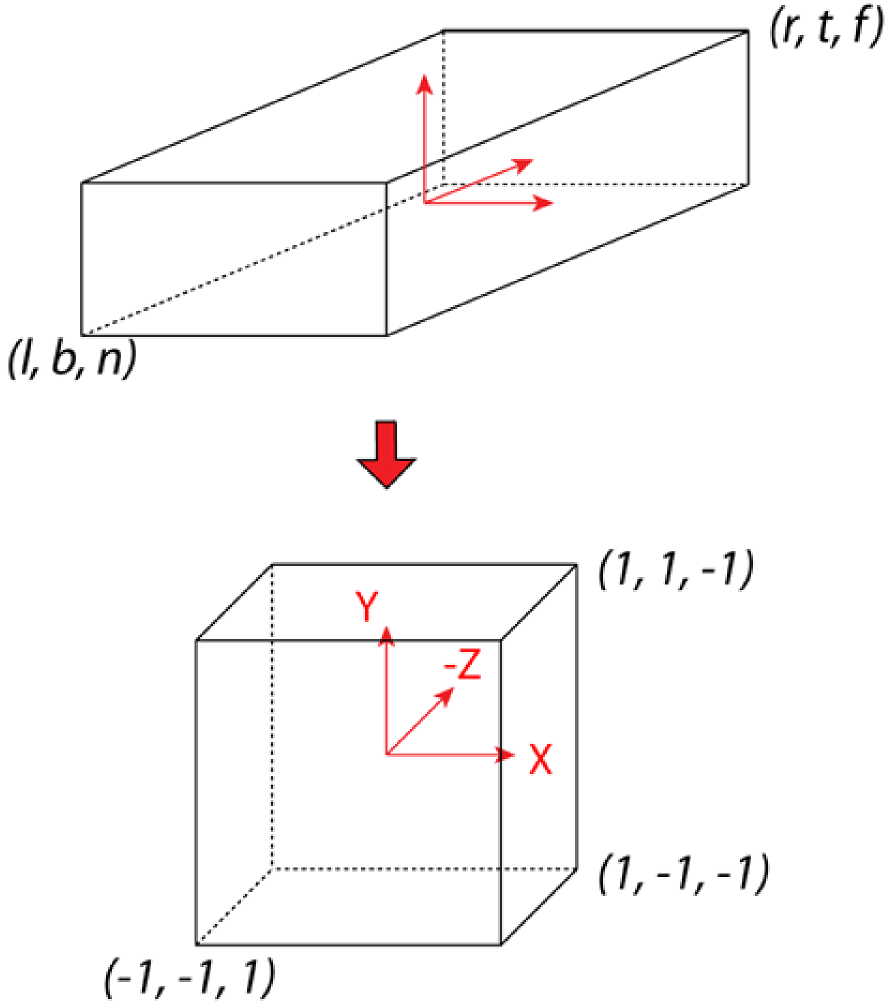
$$\begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



The orthographic view volume

Then we have to scale everything to $[-1, 1]$:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



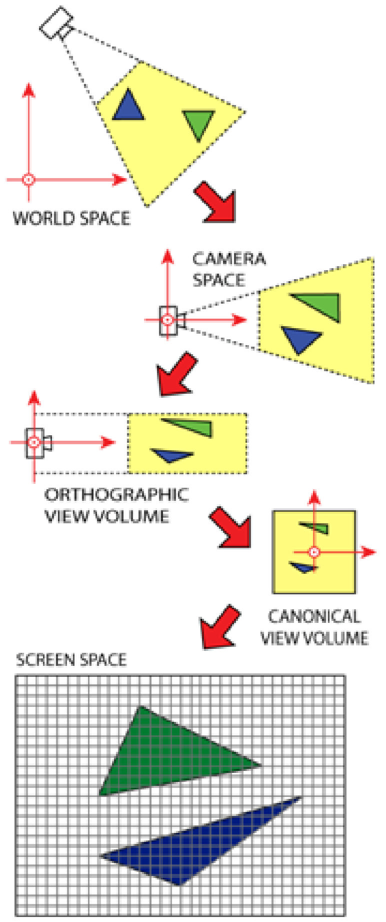
The orthographic view volume

Since these are just matrix multiplications (associative!), we can combine them into one matrix:

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

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

Overview



Hence, our last step becomes

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Now, how do we get the data in the orthographic view volume?

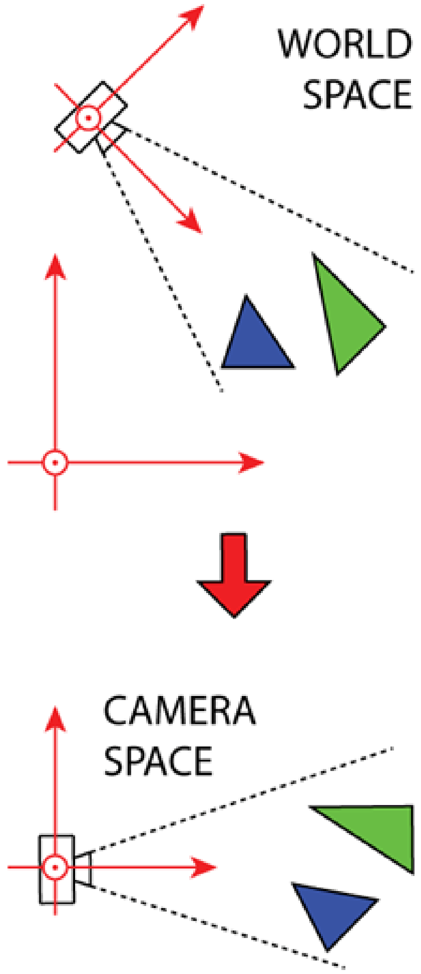
That's more difficult, so let's look at camera transformation first.

Aligning coordinate systems

How do we get the camera to the origin, i.e. how do we move from world space to camera space?

Remember:

- **world space** is expressed by the base vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$
- the **camera** is specified by eye vector $\vec{e}$ and gaze vector $\vec{g}$

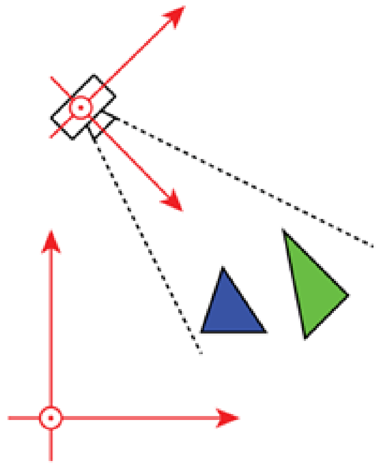


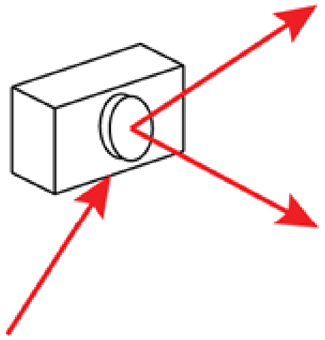
Aligning coordinate systems

To map one space to another, we need a coordinate system for both spaces.

We can easily get that using a view up vector $\vec{t}$, i.e. a vector in the plane bisecting the viewer's head into left and right halves and "pointing to the sky"

This gives us an orthonormal base $(\vec{u}, \vec{v}, \vec{w})$ of our camera coordinate system (how?)

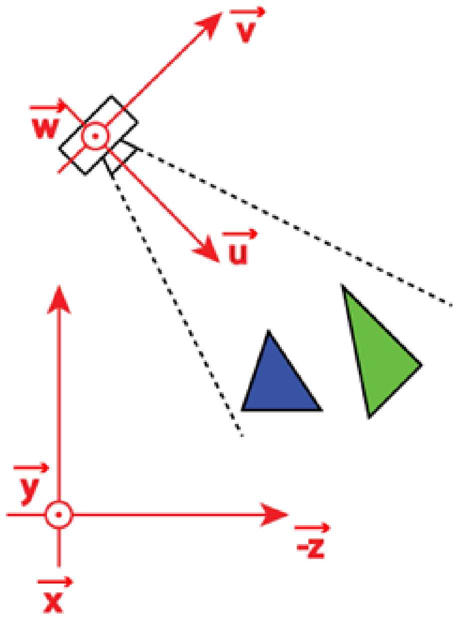




Aligning coordinate systems

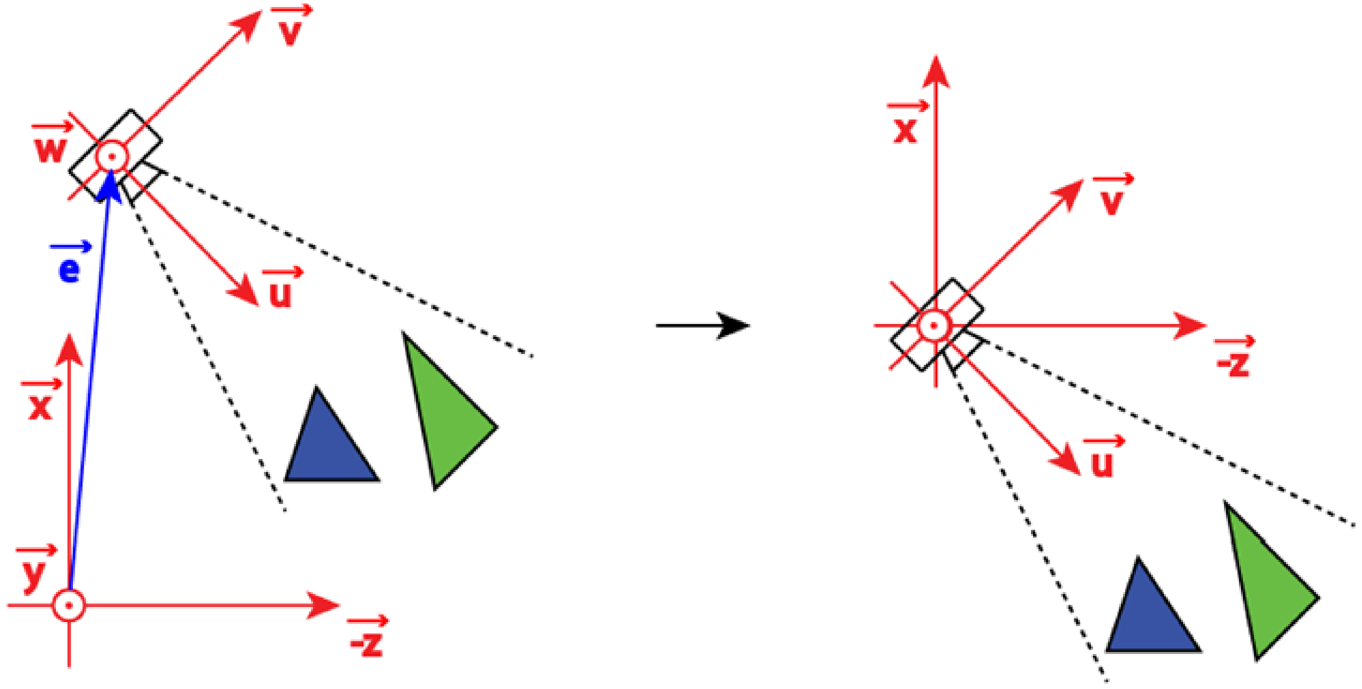
How do we align the two coordinate systems?

1. align the origins
2. align the base vectors



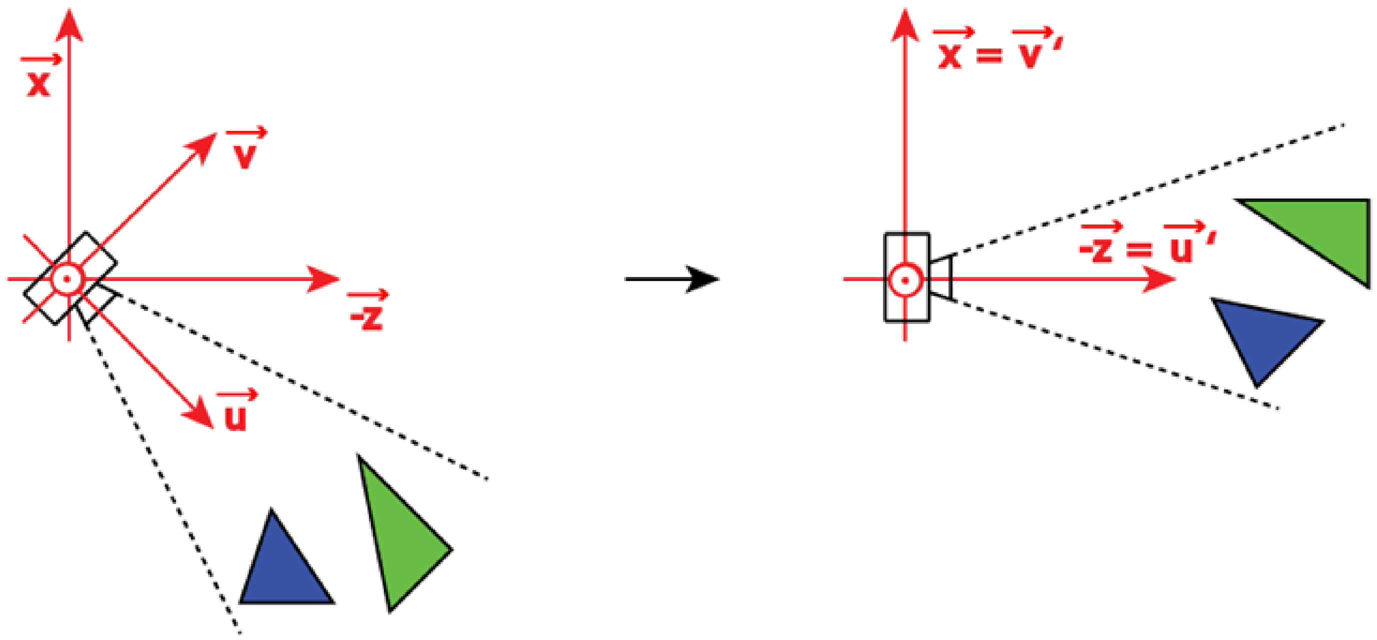
Aligning coordinate systems

Aligning the origins is a simple translation: $\begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$



Aligning coordinate systems

Aligning the axes is a simple rotation, if you remember that the columns of our matrix are just the images of the base vectors under the linear transformation.



Aligning coordinate systems

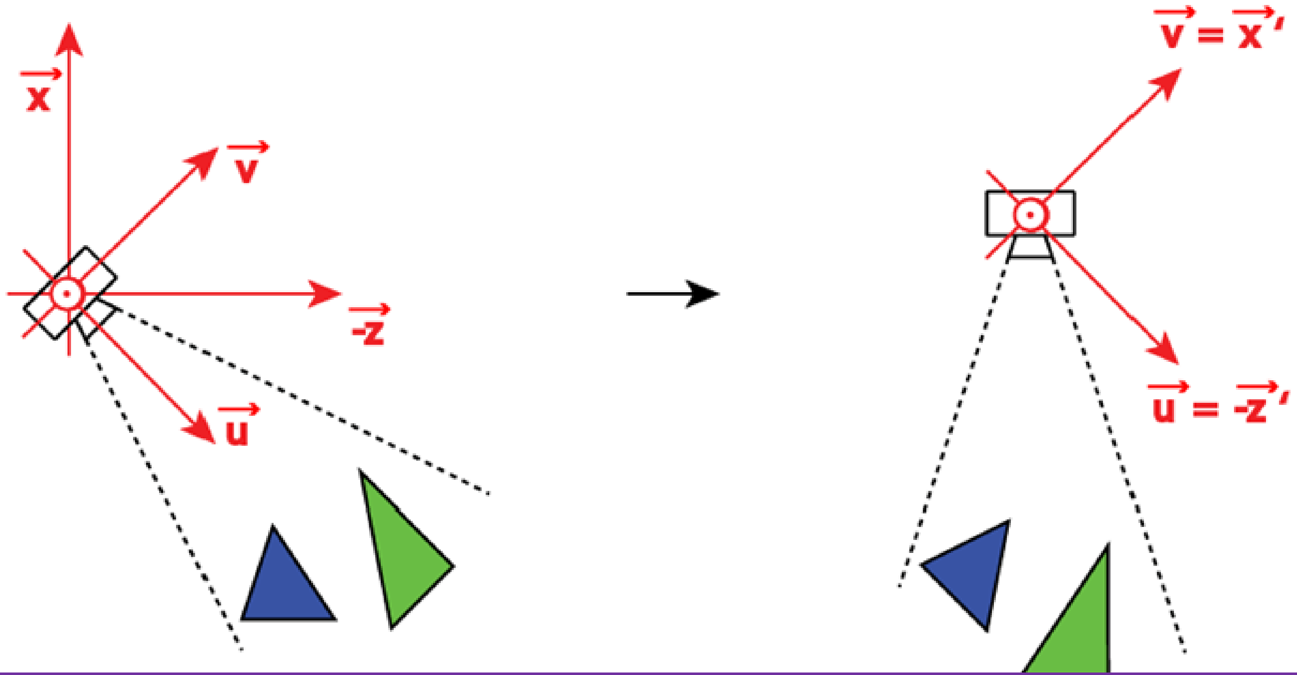
Camera Spec:
e = eye position.
g = gaze direction
t = Up Vector
e, g, t are in x,y,z coordinates.

Basis Vectors
w = -g/||g||
u = (t x w)/||t x w||
v = w x u

w, u, v in x,y, z coordinates.

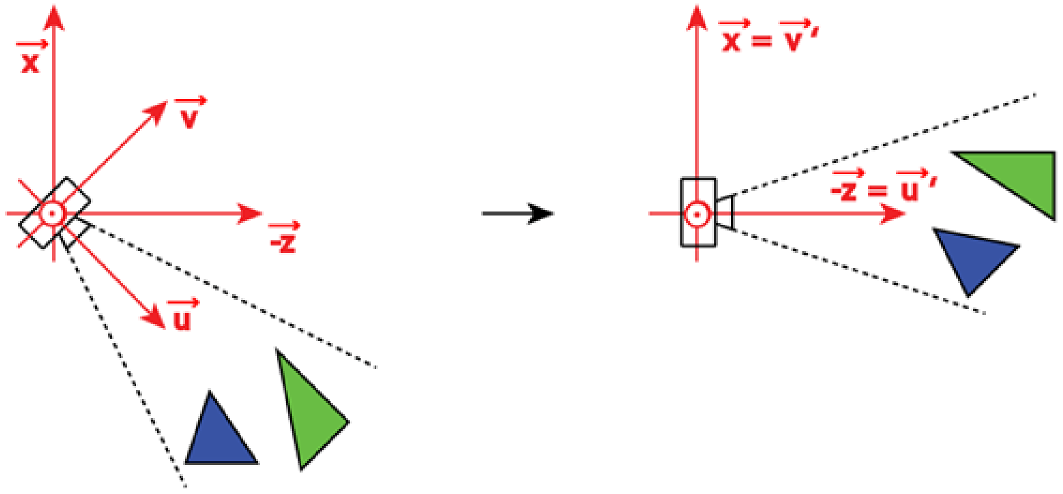
Aligning coordinate systems

These are easy to find for the reverse rotation: $\begin{pmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$



Aligning coordinate systems

Hence, our rotation matrix is: $\begin{pmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

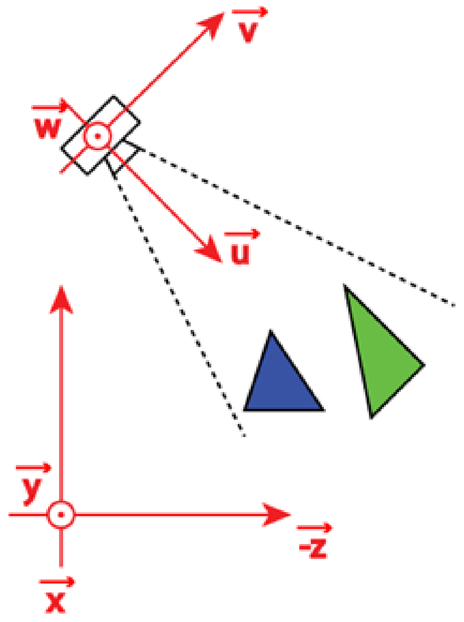


(Remember: the inverse of an orthogonal matrix is always its transposed)

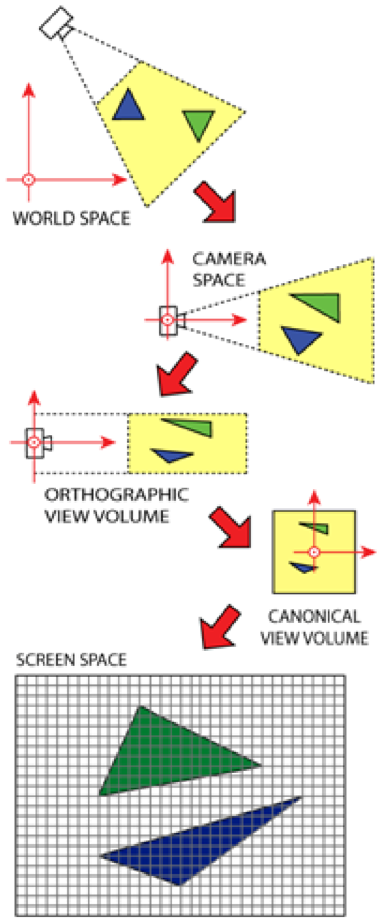
Aligning coordinate systems

For the total transformation we get

$$M_{cam} = \begin{pmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Overview



If it wasn't for perspective projection, we'd be done:

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

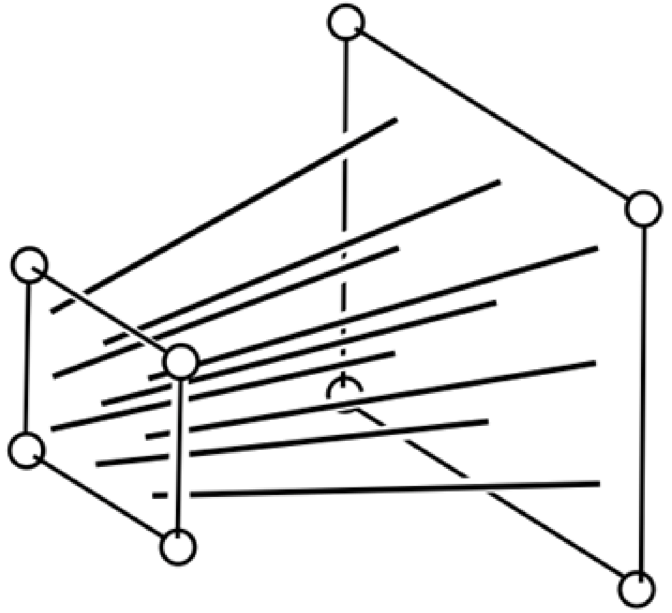
Now, let's put things into perspective . . .

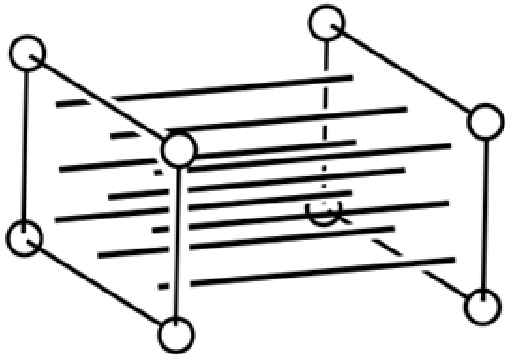
Transforming the view frustum

cf. book, fig. 7.13 (3rd ed.) or 7.12 (2nd ed.)

View frustum

Orthographic view volume



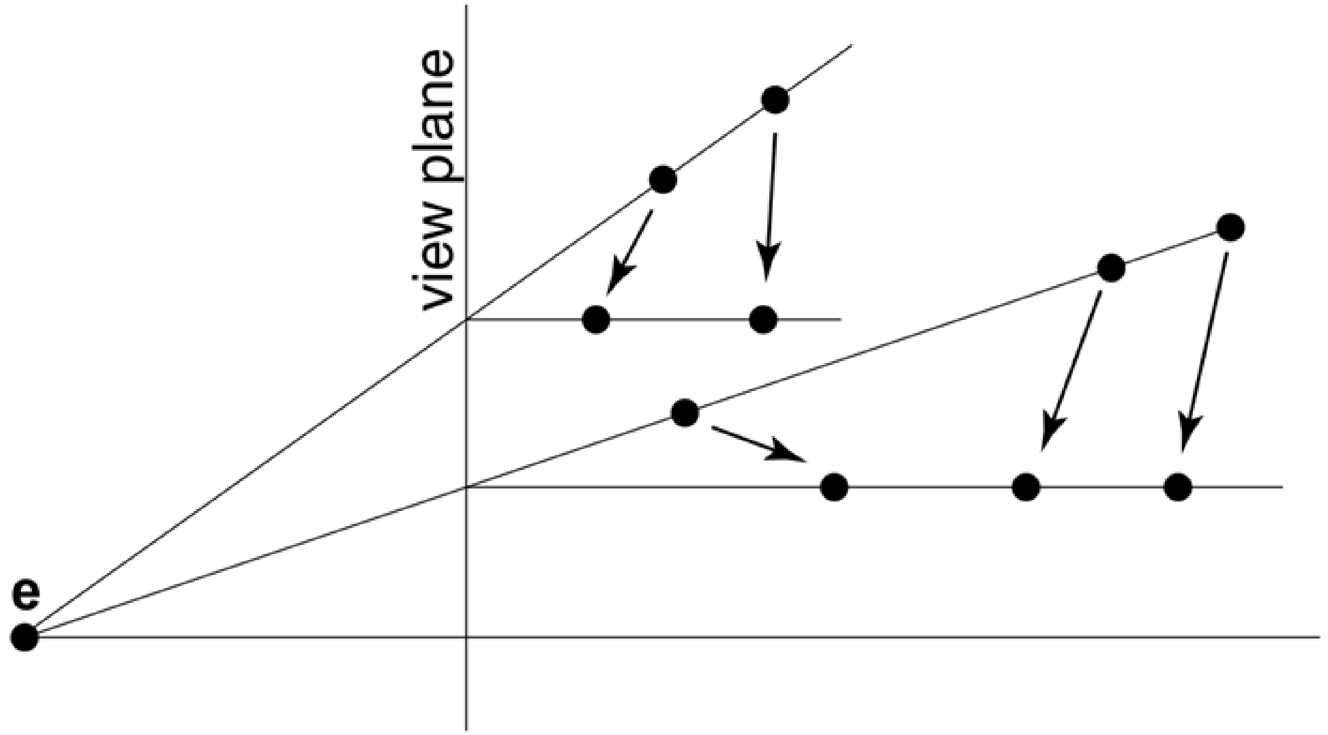


Perspective projection

Parallel/orthographic projection

Transforming the view frustum

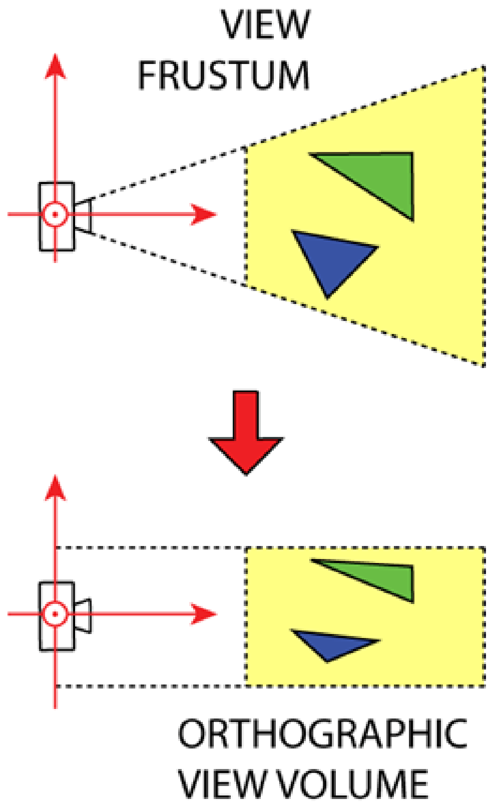
cf. book, fig. 7.10 (2nd ed.; not in 3rd one)



Transforming the view frustum

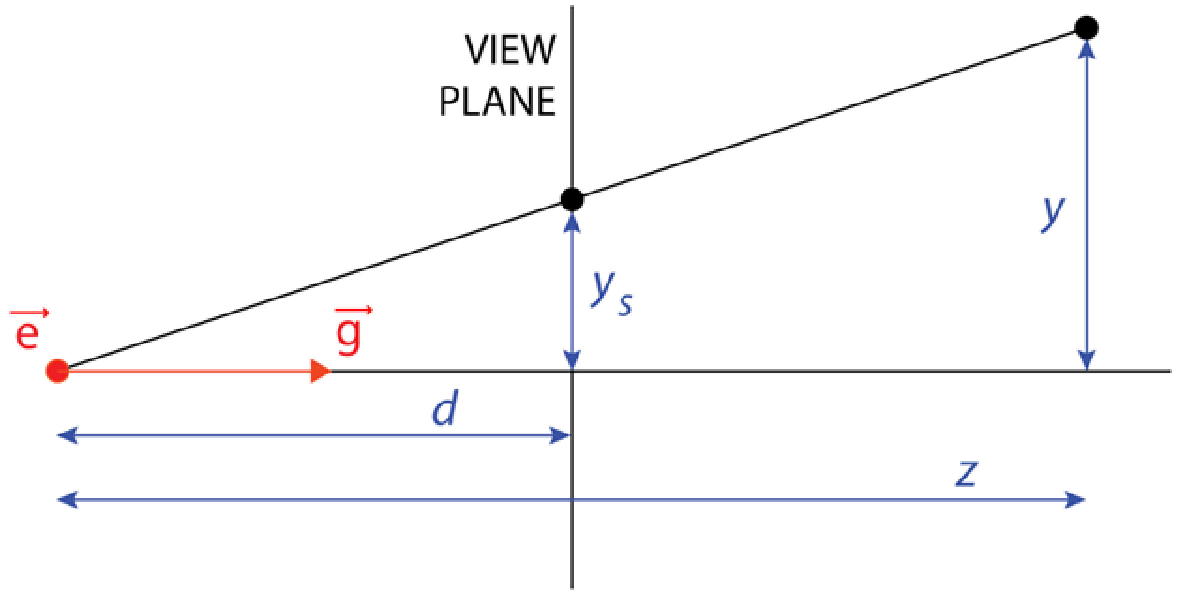
We have to transform the view frustum into the orthographic view volume. The transformation needs to

- Map lines through the origin to lines parallel to the z axis
- Map points on the viewing plane to themselves.
- Map points on the far plane to (other) points on the far plane.
- Preserve the near-to-far order of points on a line.



Transforming the view frustum

How do we calculate this? (cf. book, fig. 7.8/7.9 (3rd/2nd ed.))



From basic geometry we know:

$$\frac{y_s}{y} = \frac{d}{z} \qquad \text{and thus} \qquad y_s = \frac{d}{z} y$$

Transforming the view frustum

So we need a matrix that gives us

- $x_s = \frac{dx}{z}$
- $y_s = \frac{dy}{z}$

and a z-value that

- stays the same for all points on the near and fare planes
- does not change the order along the z-axis for all other points

Problem: we can't do division with matrix multiplication

Extending homogeneous coordinates

Remember: matrix multiplication is a linear transformation, i.e. it can only produce values such as:

$$x' = a_1x + b_1y + c_1z$$

Introducing homogeneous coordinates and representing points as $(x, y, z, 1)$, enables us to do affine transformations, i.e. create values such as:

$$x' = a_1x + b_1y + c_1z + d_1$$

Now we introduce projective transformation (aka homography) that allows us to create values such as:

$$x' = \frac{a_1x+b_1y+c_1z+d_1}{ex+fy+gz+h}$$

Extending homogeneous coordinates

How can we transform

$$\text{a vector} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ to a vector } \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \frac{a_1 x + b_1 y + c_1 z + d_1}{ex + fy + gz + h} \\ \frac{a_2 x + b_2 y + c_2 z + d_2}{ex + fy + gz + h} \\ \frac{a_3 x + b_3 y + c_3 z + d_3}{ex + fy + gz + h} \end{pmatrix}$$

using matrix multiplication?

We do this by replacing "the one" in the 4th coordinate with a value w that serves as denominator.

Extending homogeneous coordinates

With homogeneous coordinates, the vector

$$(x, y, z, 1) \quad \text{represents the point} \quad (x, y, z).$$

Now we extend this in a way that the homogeneous vector

$$(x, y, z, w) \quad \text{represents the point} \quad (x/w, y/w, z/w).$$

And matrix transformation becomes:

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Extending homogeneous coordinates

Notice that this doesn't change our existing framework (i.e. all affine transformations "still work").

We just have to set

$$e = f = g = 0 \text{ and } h = 1.$$

Then our resulting vector

$$(\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \text{ becomes } (\tilde{x}, \tilde{y}, \tilde{z}, 1),$$

and it represents the point

$$(\tilde{x}/\tilde{w}, \tilde{y}/\tilde{w}, \tilde{z}/\tilde{w}) = (\tilde{x}/1, \tilde{y}/1, \tilde{z}/1)$$

Extending homogeneous coordinates

With this extension, we do matrix multiplication:

$$\begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a_1x + b_1y + c_1z + d_1 \\ a_2x + b_2y + c_2z + d_2 \\ a_3x + b_3y + c_3z + d_3 \\ ex + fy + gz + h \end{pmatrix} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{pmatrix}$$

Followed by a step called homogenization:

$$\begin{pmatrix} a_1x + b_1y + c_1z + d_1 \\ a_2x + b_2y + c_2z + d_2 \\ a_3x + b_3y + c_3z + d_3 \\ ex + fy + gz + h \end{pmatrix} \xrightarrow{\text{homogenize}} \begin{pmatrix} \frac{a_1x+b_1y+c_1z+d_1}{ex+fy+gz+h} \\ \frac{a_2x+b_2y+c_2z+d_2}{ex+fy+gz+h} \\ \frac{a_3x+b_3y+c_3z+d_3}{ex+fy+gz+h} \\ \frac{ex+fy+gz+h}{ex+fy+gz+h} \end{pmatrix}$$

Perspective transform matrix

So, by multiplication with this matrix

$$\begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{pmatrix}$$

and homogenization,
we can create this vector

$$\begin{pmatrix} \frac{a_1x+b_1y+c_1z+d_1}{ex+fy+gz+h} \\ \frac{a_2x+b_2y+c_2z+d_2}{ex+fy+gz+h} \\ \frac{a_3x+b_3y+c_3z+d_3}{ex+fy+gz+h} \\ \frac{ex+fy+gz+h}{ex+fy+gz+h} \end{pmatrix}$$

Q: how do we chose the a_i, b_i, c_i, d_i and e, f, g, h to get what we want for perspective projection, i.e. the vector

$$\begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ z^* \\ 1 \end{pmatrix}$$

(z^* denotes a z-value fulfilling the conditions that we specified)

Perspective transform matrix

The following matrix will do the trick:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Notice that

- we are looking in negative Z-direction
- n, f denote the near and far plane of the view frustum
- n serves as projection plane

Let's verify that ...

Perspective transform matrix

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z\frac{n+f}{n} - f \\ \frac{z}{n} \end{pmatrix} \xrightarrow{\text{homogenize}} \begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{pmatrix}$$

Indeed, that gives the correct values for x_s and y_s.

But what about z? Remember our requirements for z:

- stays the same for all points on the near and fare planes
- does not change the order along the z-axis for all other points

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- values on the near plane stay on the near plane, i.e. if $z = n$, then $z_s = n$:

$$z_s = n + f - f = n = z$$

- values on the far plane stay on the far plane, i.e. if $z = f$, then $z_s = n$:

$$z_s = n + f - n = f = z$$

and ...

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- values within the view frustum stay within the view frustum, i.e. if $z > n$ then $z_s > n$:

  $$z_s = n + f - \frac{fn}{z} > n + f - \frac{fn}{n} = n$$

  and if $z < f$ then $z_s < f$:

  $$z_s = n + f - \frac{fn}{z} < n + f - \frac{fn}{f} = f$$

and ...

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- the order along the Z-axis is preserved,
  i.e. if $z_1 > z_2$ then $z_{1s} > z_{2s}$:

$$z_{1s} = n + f - \frac{fn}{z_1} > n + f - \frac{fn}{z_2} = z_{2s}$$

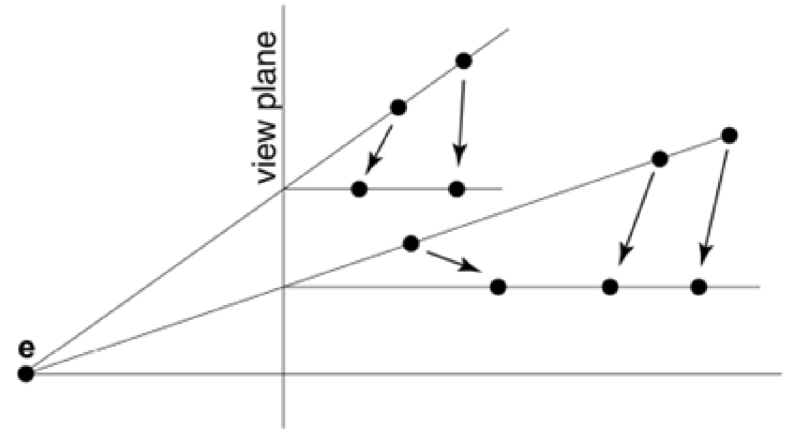
$$-\frac{1}{z_1} < -\frac{1}{z_2}$$

$$z_1 > z_2$$

and we are done.

Homogeneous coordinates and perspective transformation

Hence, the order is preserved. But how?



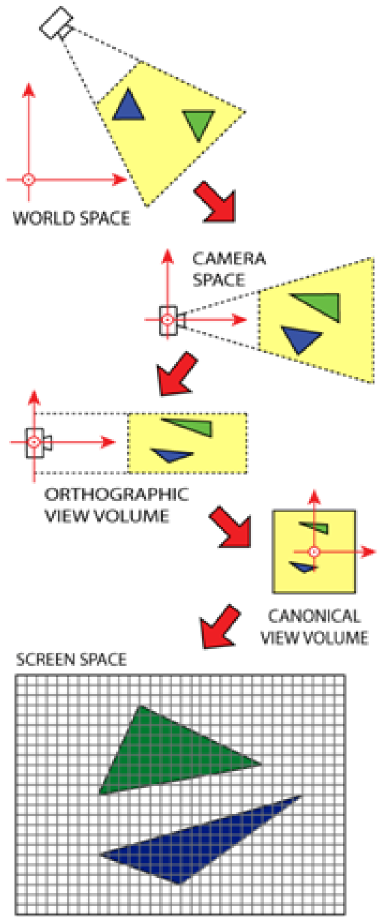
$z_s = n + f - \frac{fn}{z}$,
so z_s is proportional to $-\frac{1}{z}$

Perspective transform matrix

With this, we got our final matrix P. To map the perspective view frustum to the orthographic view volume, we need to combine it with the orthographic projection matrix M_{orth}, i.e. $M_{per} =$

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

Overview



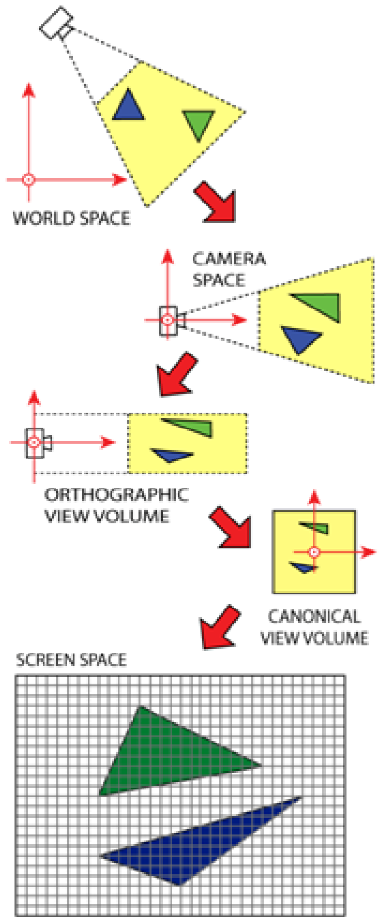
The following achieved parallel projection:

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

And if we replace M_{orth} with M_{per} we get perspective projection:

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{per} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Wrap-up



To draw lines on the screen, we can use the following pseudo code:

```
compute $\mathbf{M_{vp}}$ ///view port
compute $\mathbf{M_{per}}$ ///persp.  proj.
compute $\mathbf{M_{cam}}$ ///camera space
$\mathbf{M} = \mathbf{M_{vp}M_{per}M_{cam}}$
for each line segment $(\mathbf{a_i}, \mathbf{b_i})$ do
  $\mathbf{p} = \mathbf{Ma_i}$
  $\mathbf{q} = \mathbf{Mb_i}$
  drawline$(x_p/w_p, y_p/w_p, x_q/w_q, y_q/w_q)$
```