FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lecture #7

Viewing and Projection

Computer Graphics
Winter Term 2016/17

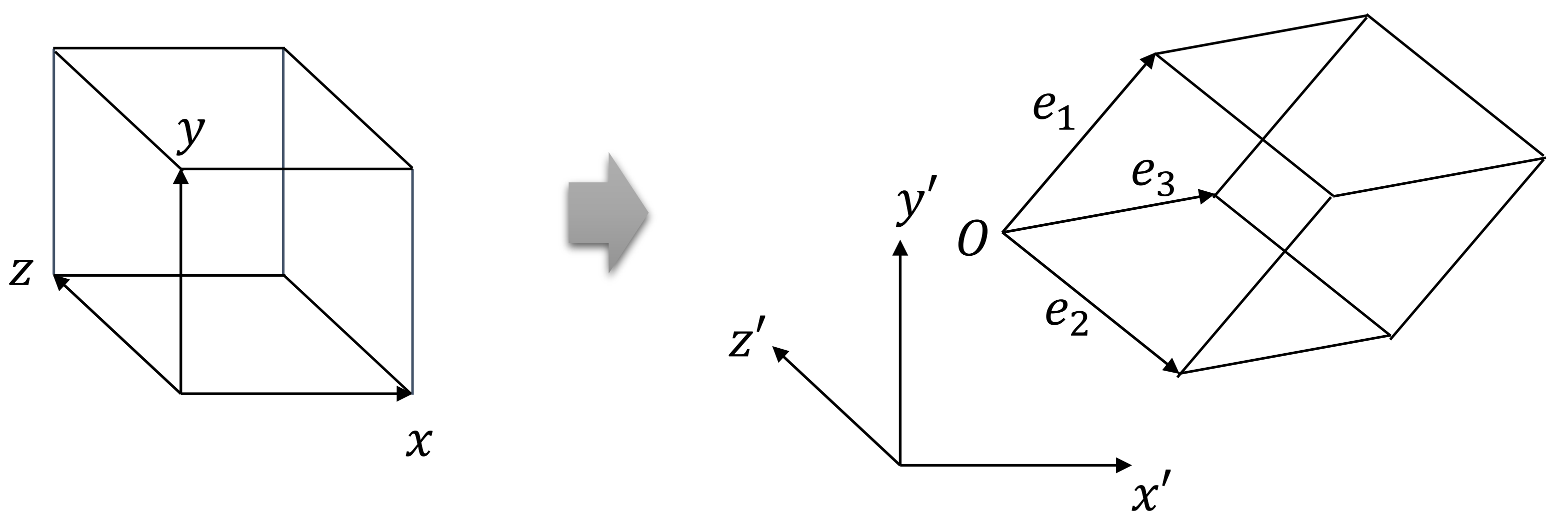
Marc Stamminger / Roberto Grosso

Remember

- Affine Transformations in 2D and 3D:
  - Coordinate system changes:

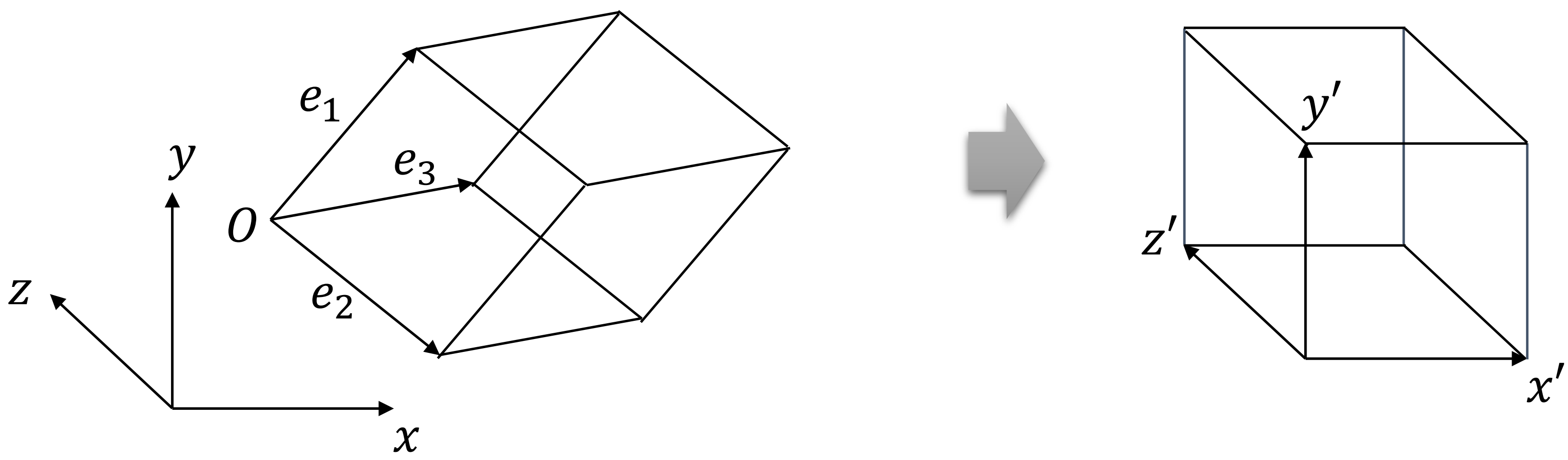
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & \vdots & O_1 \\ e_1 & e_2 & e_3 & O_2 \\ \vdots & \vdots & \vdots & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

  - Maps the unit cube to a parallelepiped



Remember

- Other direction: maps any parallelepiped to the unit cube



- use the inverse matrix:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & \vdots & O_1 \\ e_1 & e_2 & e_3 & O_2 \\ \vdots & \vdots & \vdots & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Remember

- Special case: orthonormal e_1, e_2, e_3:

$$e_1 \circ e_2 = e_1 \circ e_3 = e_2 \circ e_3 = 0, \qquad e_1 \circ e_1 = e_2 \circ e_2 = e_3 \circ e_3 = 1$$

- = Rigid Transformation: rotation + translation

- The inverse of rotation = transposed (for linear part)

$$\begin{pmatrix} \vdots & \vdots & \vdots & O_1 \\ e_1 & e_2 & e_3 & O_2 \\ \vdots & \vdots & \vdots & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} & R & & \vdots \\ & & & t \\ & & & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} & R^T & & \vdots \\ & & & -R^T t \\ & & & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Remember

- more general: orthogonal e_1, e_2, e_3:

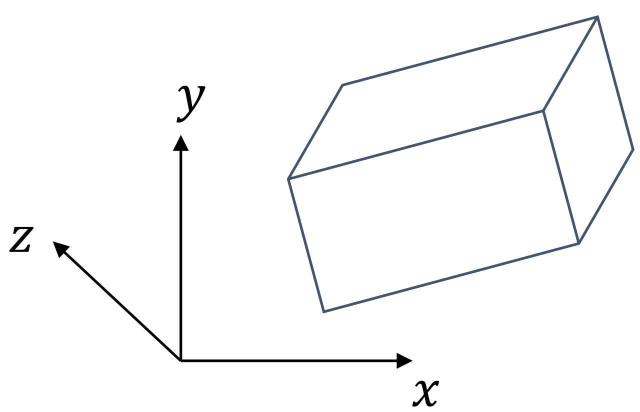
$$e_1 \circ e_2 = e_1 \circ e_3 = e_2 \circ e_3 = 0$$

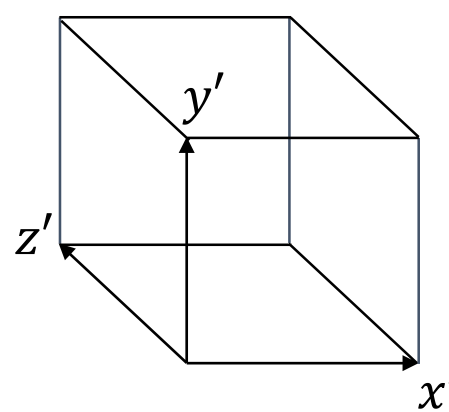
$$\mathrm{E} = \begin{pmatrix} \vdots & \vdots & \vdots \\ e_1 & e_2 & e_3 \\ \vdots & \vdots & \vdots \end{pmatrix}$$

- Then:

$$\begin{pmatrix} \vdots & \vdots & \vdots & O_1 \\ e_1 & e_2 & e_3 & O_2 \\ \vdots & \vdots & \vdots & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{\|e_1\|^2} & 0 & 0 \\ 0 & \frac{1}{\|e_2\|^2} & 0 \\ 0 & 0 & \frac{1}{\|e_3\|^2} \end{pmatrix} \begin{pmatrix} & E^T & & \vdots \\ & & & -E^T t \\ & & & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

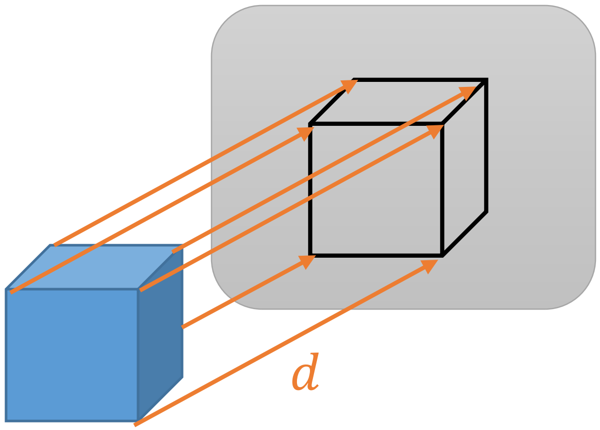
- Maps an arbitrary box to the unit cube

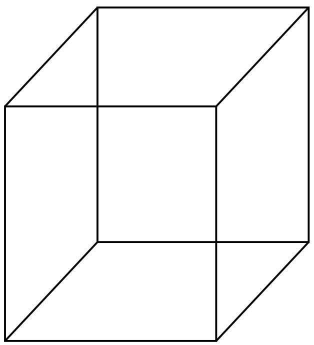




Remember

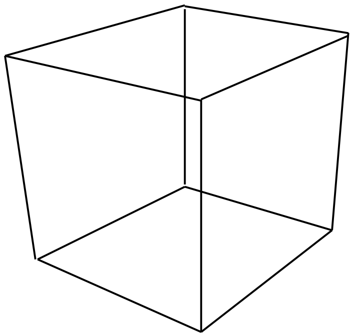
- Simple projection: parallel projection onto plane





orthographic projection

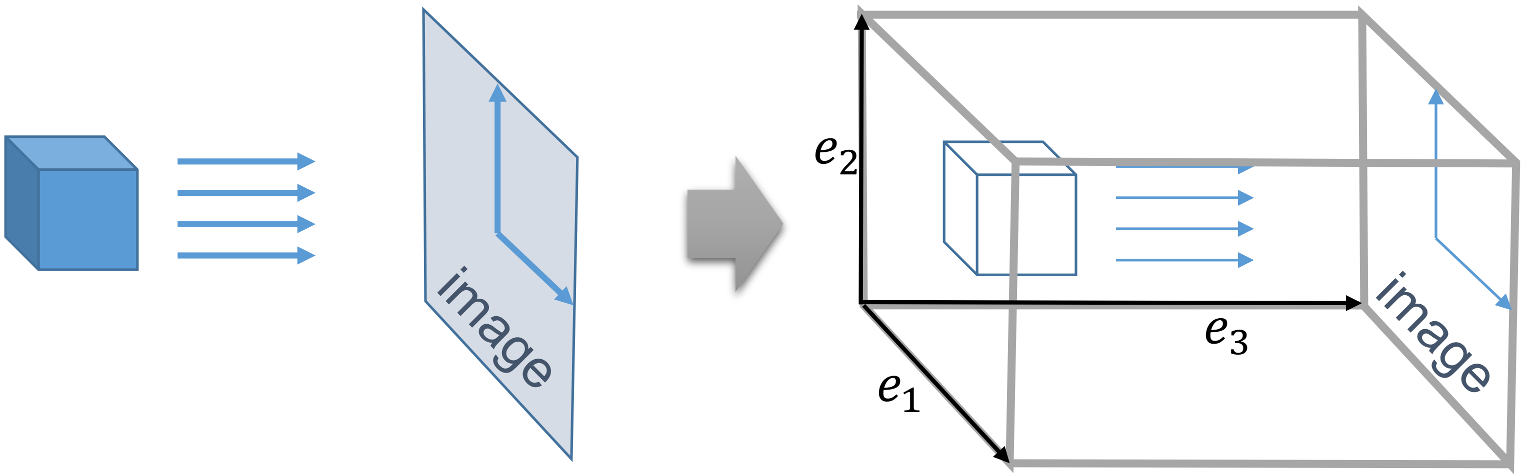
- Affine → parallel lines remain parallel
- no real perspective yet !



“real” perspective projection

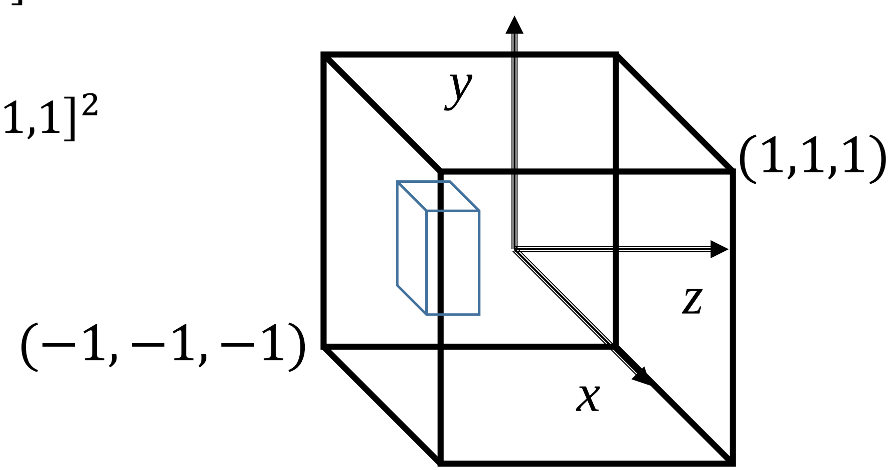
Orthographic Projection

- Alternative interpretation:
  - define a box in 3D



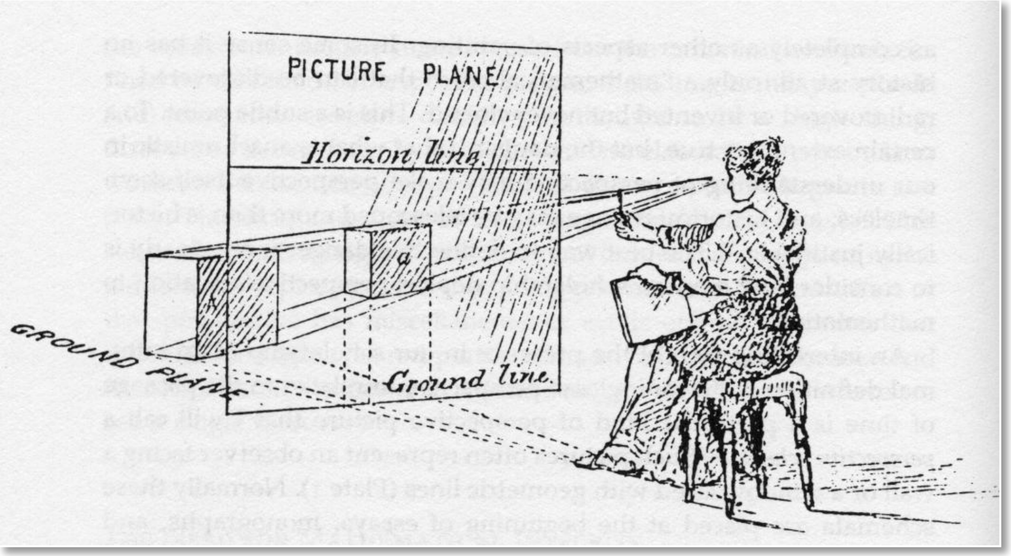
  - transform this box to unit cube $[-1,1]^3$

  - $\rightarrow (x, y)$ are image coordinates $\in [-1,1]^2$
    $\rightarrow z$ is normalized depth $\in [-1,1]$

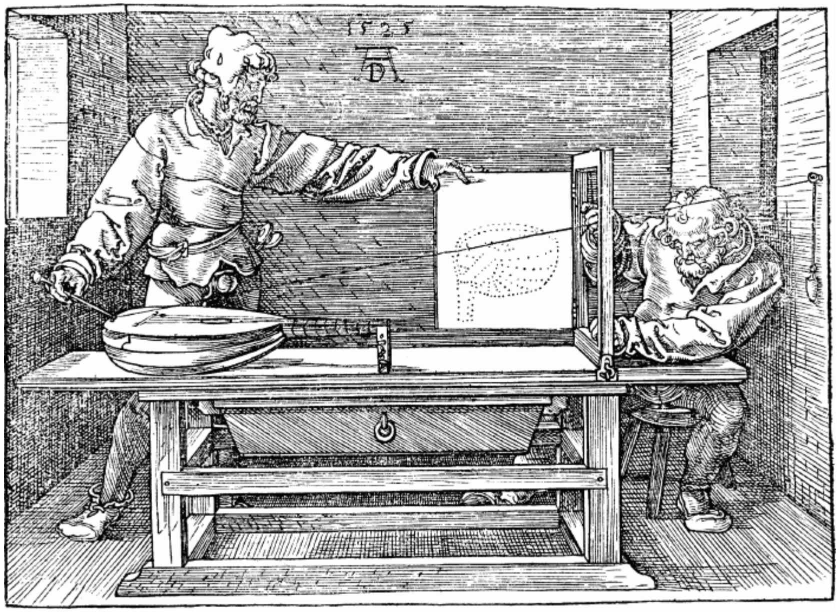


Perspective Projection

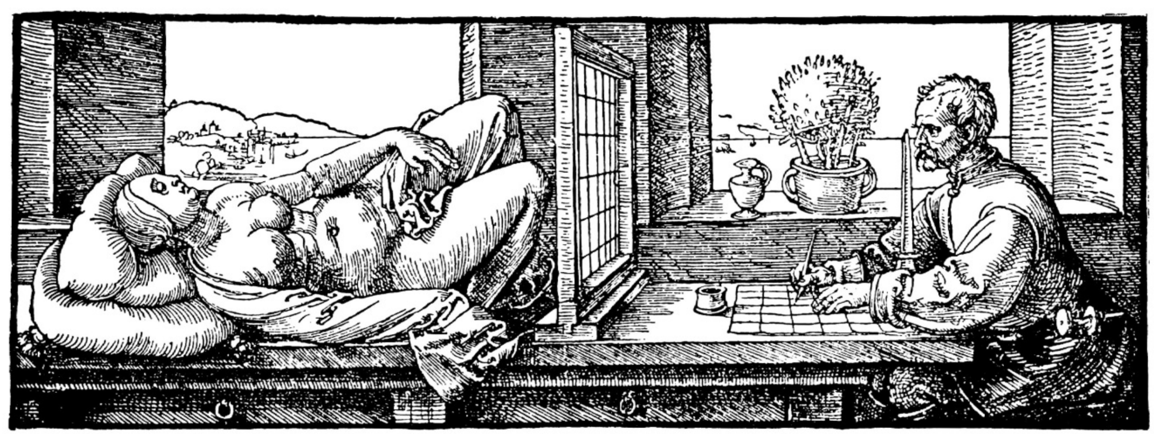
- Strategy based on simple mathematical rule
- Project objects directly towards the eye
- Draw object where they meet a view plane in front of the eye



Perspective Projection



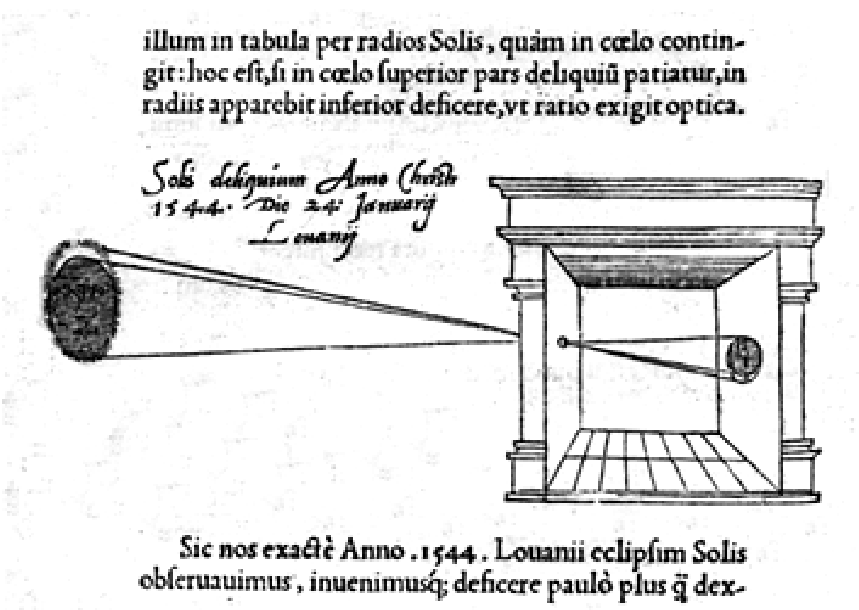
Albrecht Dürer
Der Zeichner der Laute
1512–1525



Albrecht Dürer
Der Zeichner des liegenden Weibes
1512–1525

Perspective Projection

- Linear Perspective Projection: The pinhole camera

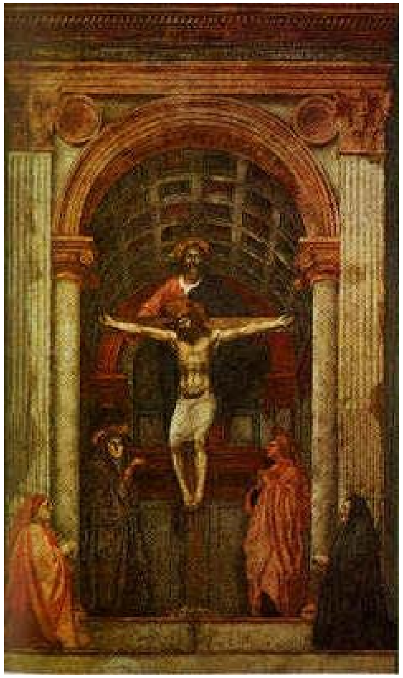


"When images of illuminated objects ... penetrate through a small hole into a very dark room ... you will see [on the opposite wall] these objects in their proper form and color, reduced in size ... in a reversed position, owing to the intersection of the rays".
Da Vinci

http://www.acmi.net.au/AIC/CAMERA_OBSCURA.html (Russell Naughton)

Perspective Projection

Pietro Perugino, fresco at the Sistine Chape (1481-82).
Source: http://en.wikipedia.org/wiki/Vanishing_point



Masaccio 1427, Trinitz with the Virgin, St. John and Donors.
First ever painting done in perspective.

Canaletto 1735-45. The Piazza of San Marco, Venice.
One point perspective

Source: http://www.siggraph.org/education/materials/HyperGraph/viewing/view3d/perspect.htm

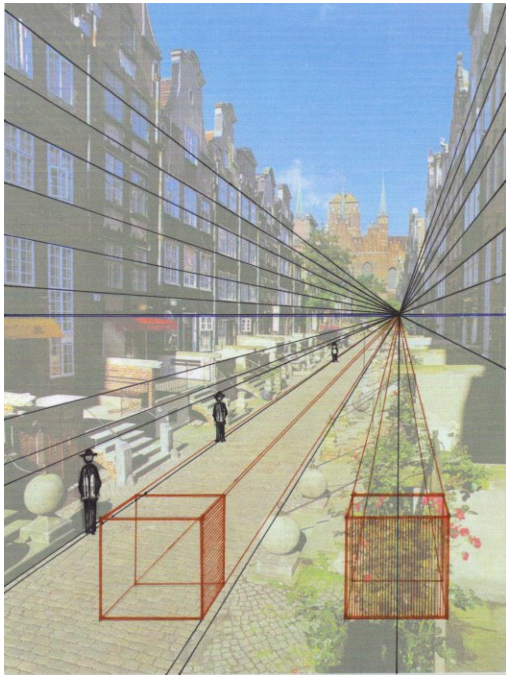
Perspective Projection

- Properties:
  - Objects appear smaller as their distance to the observer increases (foreshortening)
  - Vanishing points (Fluchtpunkte): Lines parallel in world converge to a single point in image space (rails of a railroad)
  - 1, 2 or 3-point perspective: Lines parallel to 1, 2 or 3 of the main axes converge in a vanishing point, the others remain parallel
- One point perspective
  - the image plane is orthogonal to one of the coordinate axis and parallel to the other two.
- Two point perspective
  - The image plane in parallel to one coordinate axis and intersect the other two.
- Three-point perspective
  - The image plane intersects all three coordinate axis.

Perspective Projection

- One-point perspective – one vanishing point

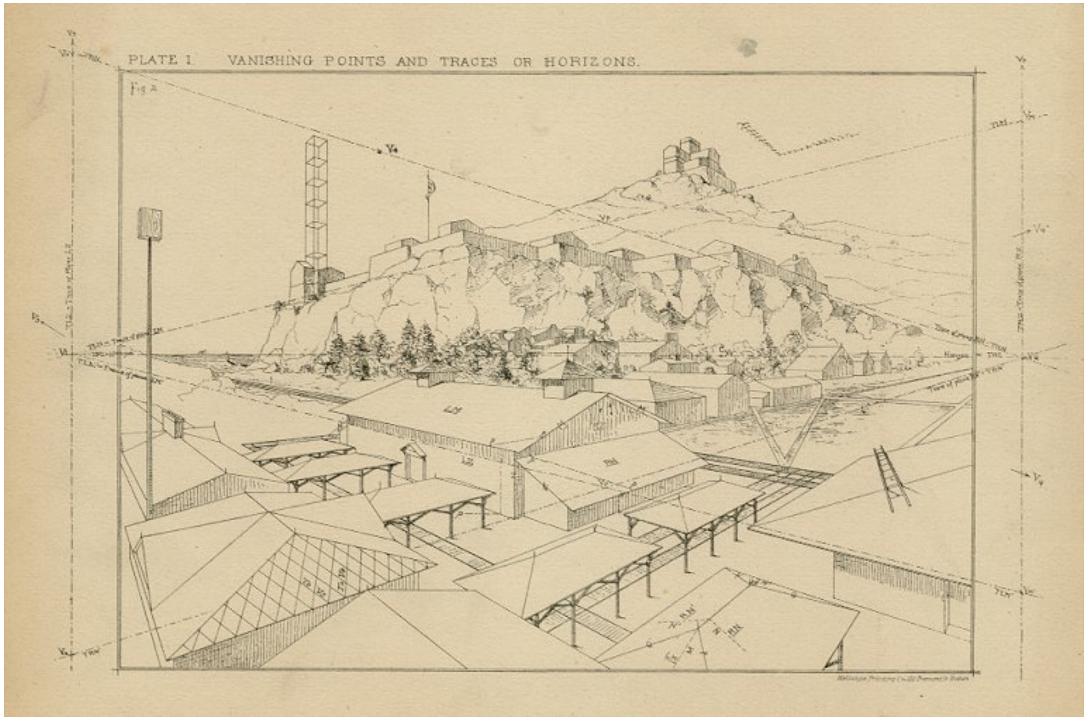
source: http://stevewebel.com/photographer/wp-content/uploads/2008/04/vanishing-point.jpg



source: http://cavespirit.com/CaveWall/5/vanishing_point_high_horizon.jpg

Perspective Projection

- Two-point perspective – two vanishing points



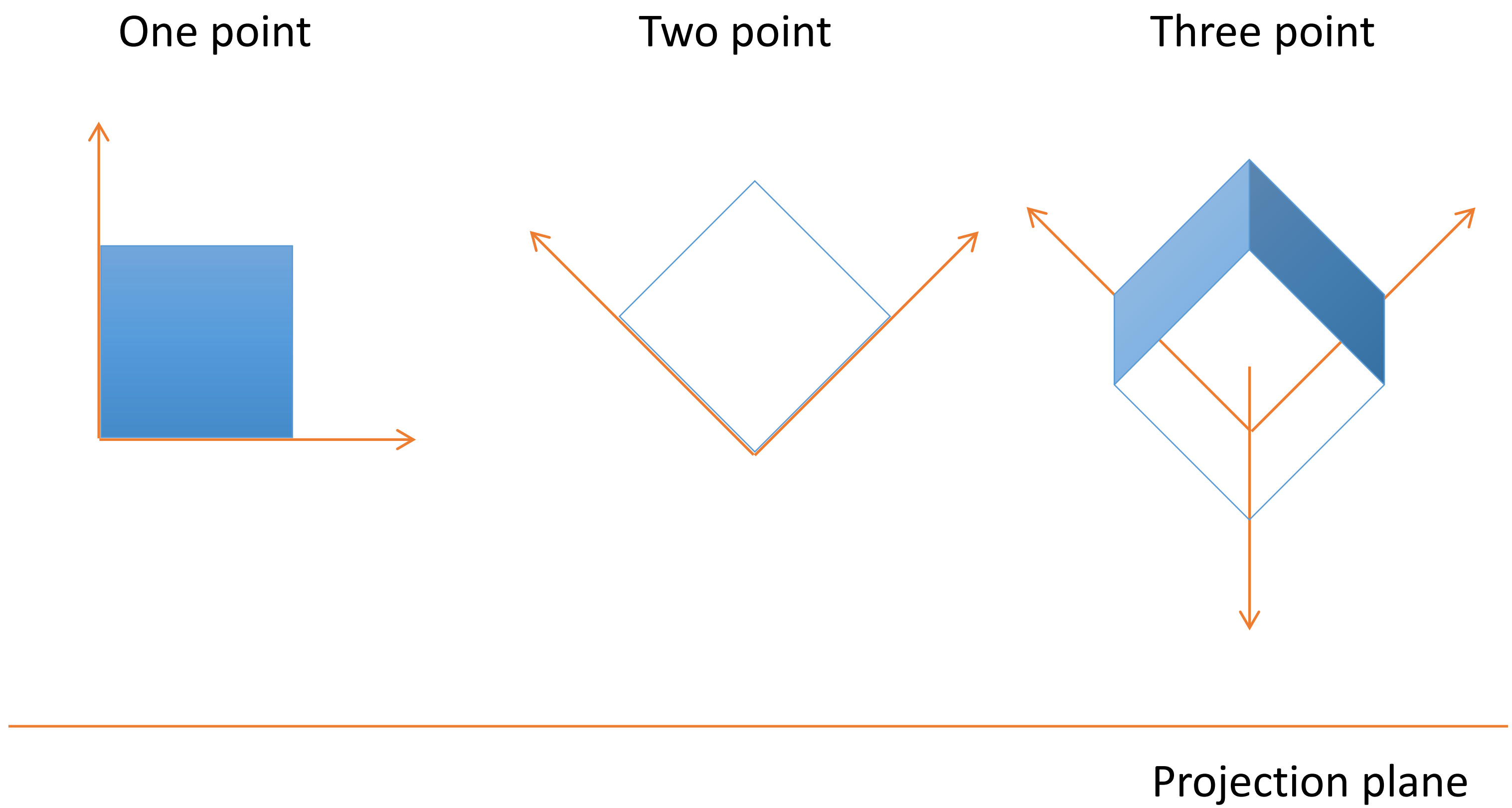
http://www.vintage-views.com/WaresModernPerspective/images/1219k6-Plate1.jpg

Perspective Projection

- Two-point perspective – two vanishing points

Sanaa-essen-Zollverein-School-of-Management-and-Design-220409-01.jpg
de.wikipedia.org

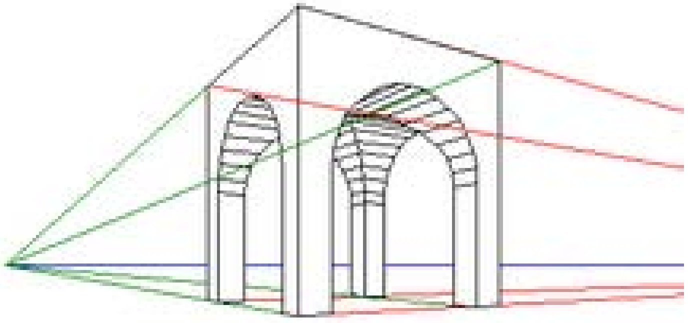
Perspective Projection



Projective Transformations

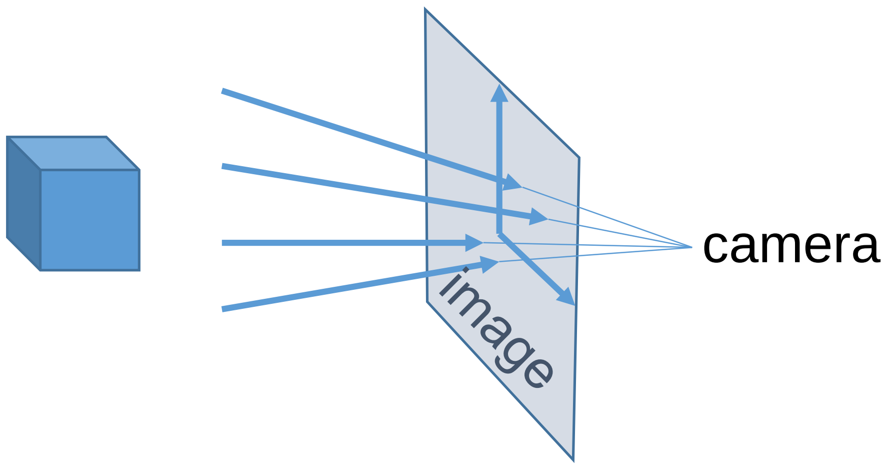
- The projective transformation maps points at infinity into regular points and it might map regular points into points at infinity.

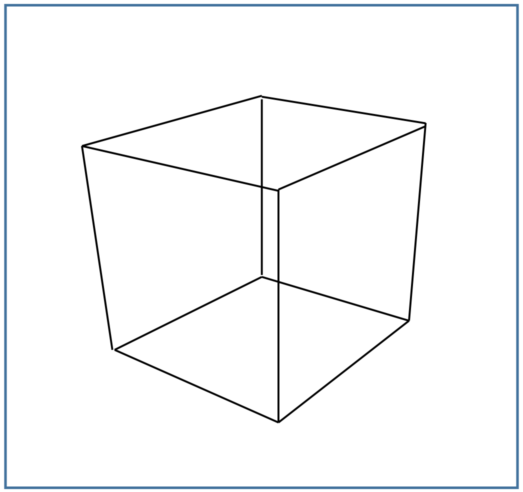
  → we will look at these weird properties in the next lecture



Perspective Projection

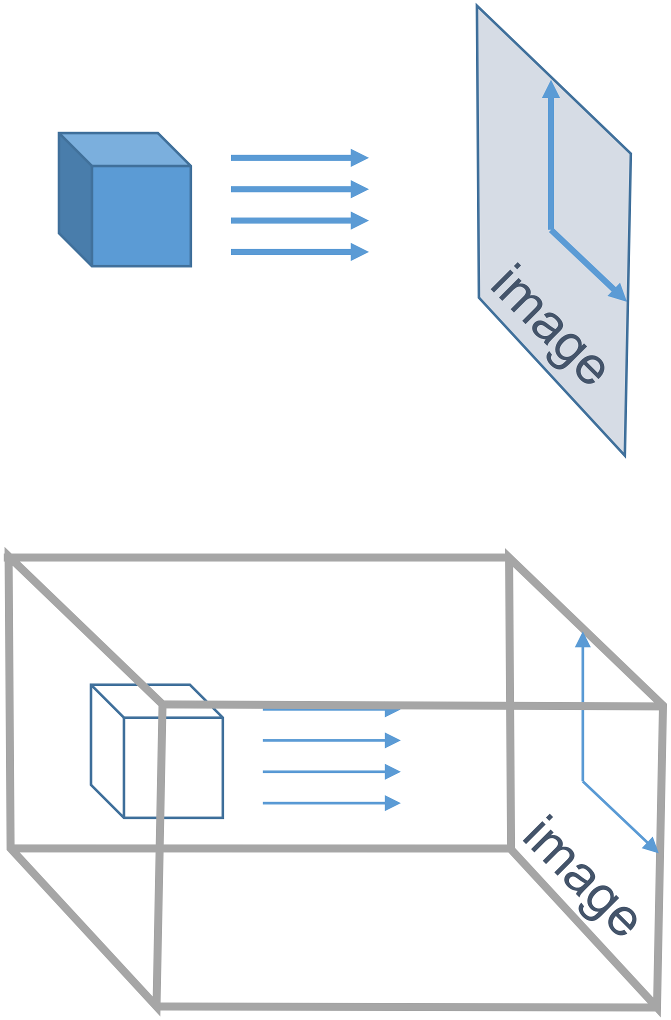
- Project scene onto image plane using point projection with camera as projection center





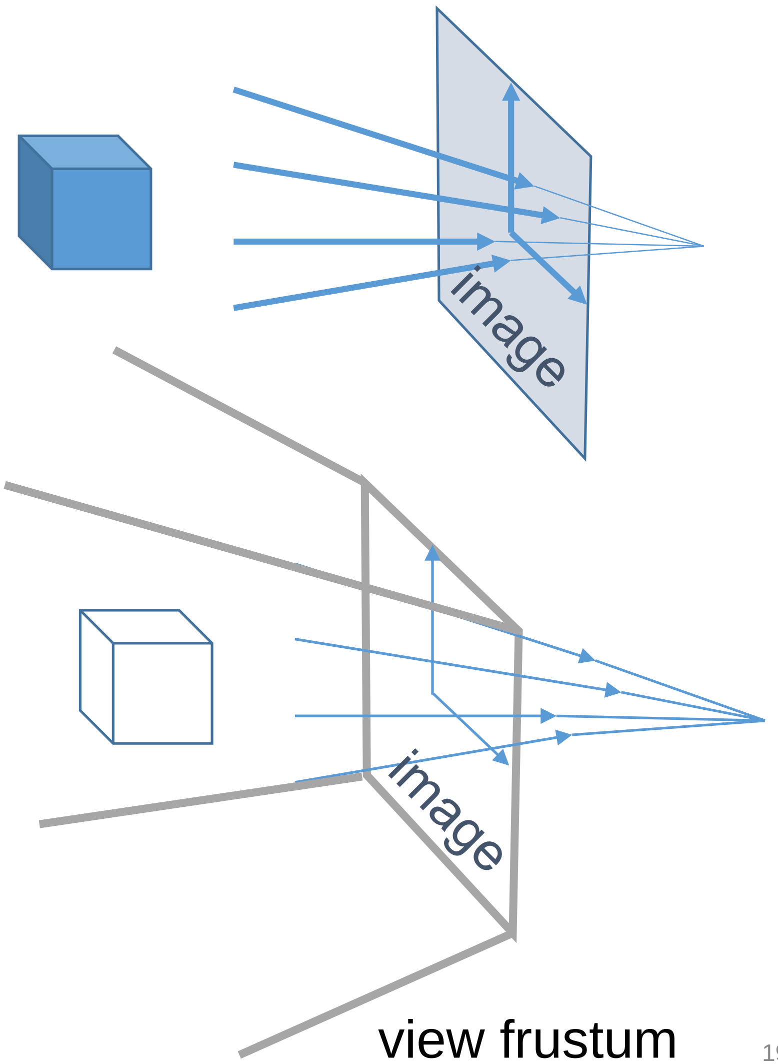
Perspective Projection

- orthographic projection



view box

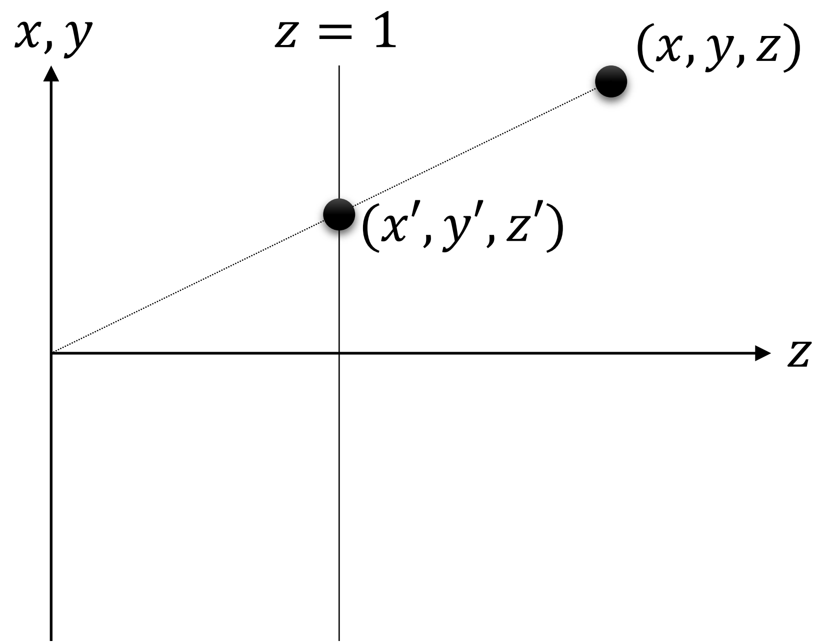
- perspective projection



view frustum

Perspective Projection

- How can we describe this projection?
- Look at special case:
  - camera in origin
  - looks into z-direction
  - projection onto $z = 1$ plane
- $\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ 1 \end{pmatrix}$
- Projection is division by z !



Perspective Projection

- How can we handle this ?
- Remember homogeneous coordinates:

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \xrightarrow{\cdot A} \begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} \rightarrow \begin{pmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{pmatrix}$$

- w is common divisor
- if we move z to w, the final division will generate perspective:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \xrightarrow{\cdot M} \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix} \quad \text{with } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Viewing and Projection

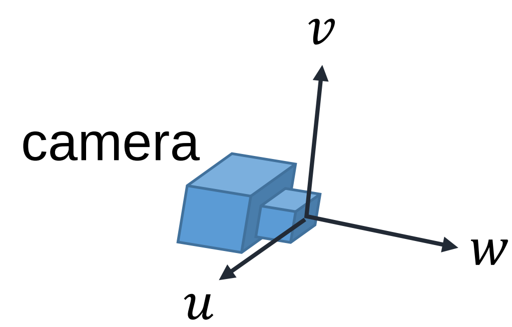
- How can we generalize all this?
  → To describe both orthogonal and perspective projection, we consider two separate steps, both described as matrices:

- First **Viewing**
  - defines camera position and view direction
  - rigid transformation
  - moves camera position to origin and aligns axes:
    → x-axis points in horizontal image direction
    → y-axis points in vertical image direction
    → z-axis points in view direction
  - to get a right-handed coordinate system, often $-z$ is view direction

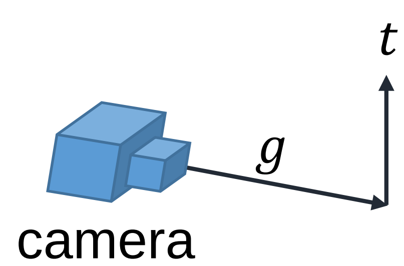
- Then **Projection**
  - then an orthogonal or perspective projection is performed
  - and a rectangular regions from the image plane mapped to the final image

Viewing Transformation

- Compute axes for viewing transformation
  → u-axis points in horizontal image direction
  → v-axis points in vertical image direction
  → w-axis points in view direction

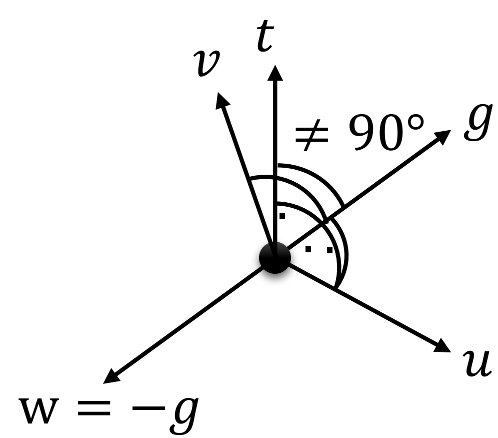


- We define these indirectly using the following three more intuitive vectors:
  - Eye position e: location of the eye / center of the lens
  - Gaze direction g: direction the viewer is looking
  - View-up vector t: bisects Viewers head, "Points to the sky"
    → typically $(0,1,0)$ → "y is up" or $(0,0,1)$ → "z is up"



Viewing Transformation

- Given: camera position e, view direction g and up-vector t
- Compute new basis: origin e and basis vectors (u, v, w)
- w
  - points opposite to gaze direction ("-z" convention): $w = -g/\|g\|$
- v
  - almost the same as t, but not always
  - if gaze direction is not perpendicular to t, then we have to rotate v away from t
  - v, t, and g should be in one plane
  - simple solution: first compute u, then v
- u
  - should be perpendicular to both g and t:
  $$u = \frac{t \times w}{\|t \times w\|}$$
  - then v is perpendicular to both u and w:
  $$\mathrm{v} = \mathrm{w} \times u$$

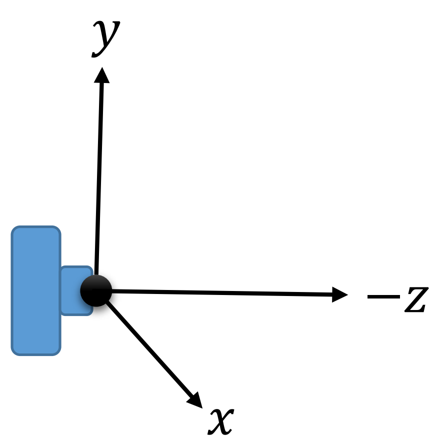


Viewing Transformation

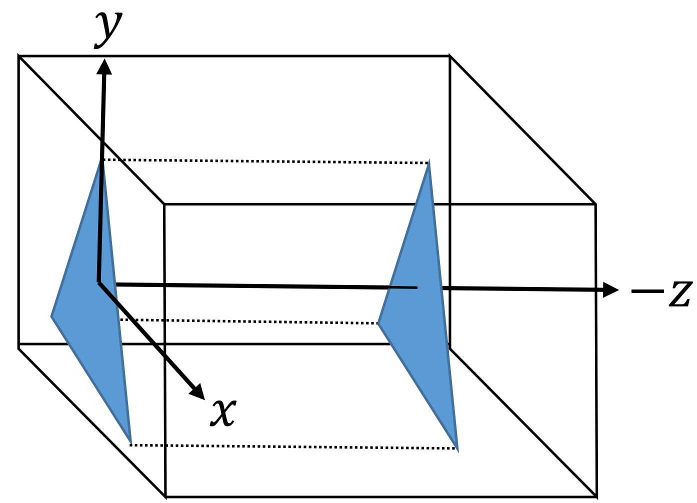
- Given: camera position e, view direction g and up-vector t
- Recipe
  - $w \; = \; -g/\|g\|$
  - $u = \frac{t \times w}{\|t \times w\|}$
  - $\mathrm{v} = \mathrm{w} \times u$
  - The viewing transformation is then (see intro slides; u, v, w are orthonormal):
  - $R = \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} \quad \mathrm{e} = \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix}$
  - $M_v = \begin{pmatrix} & R^T & & \vdots \\ & & & -R^T e \\ & & & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -u^T e \\ v_x & v_y & v_z & -v^T e \\ w_x & w_y & w_z & -w^T e \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Viewing → Projection

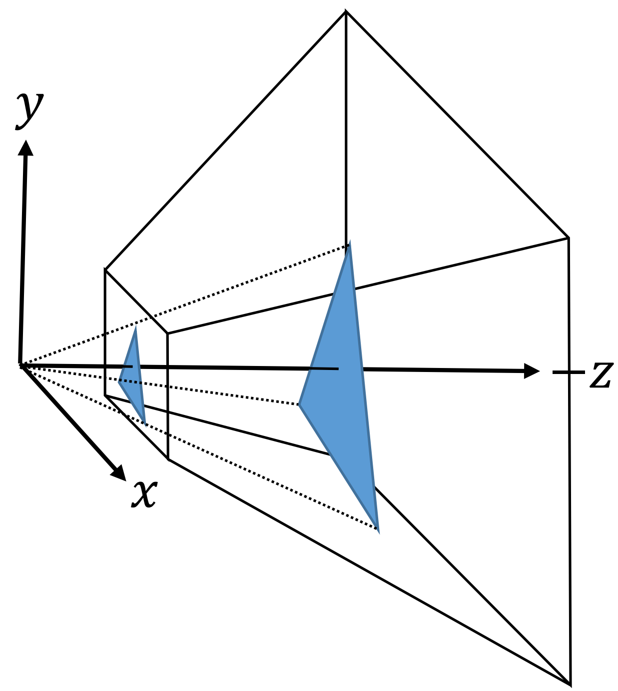
- When the coordinates are aligned with the camera, we have a much simpler situation:



orthogonal projection



perspective projection



Orthogonal Projection

- Projection onto image plane $z = 0$:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- This way, z (=depth) gets lost…
- so we keep z:

$$M_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = I$$

Perspective Projection

- For a perspective projection, we use the $z = 1$ image plane

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

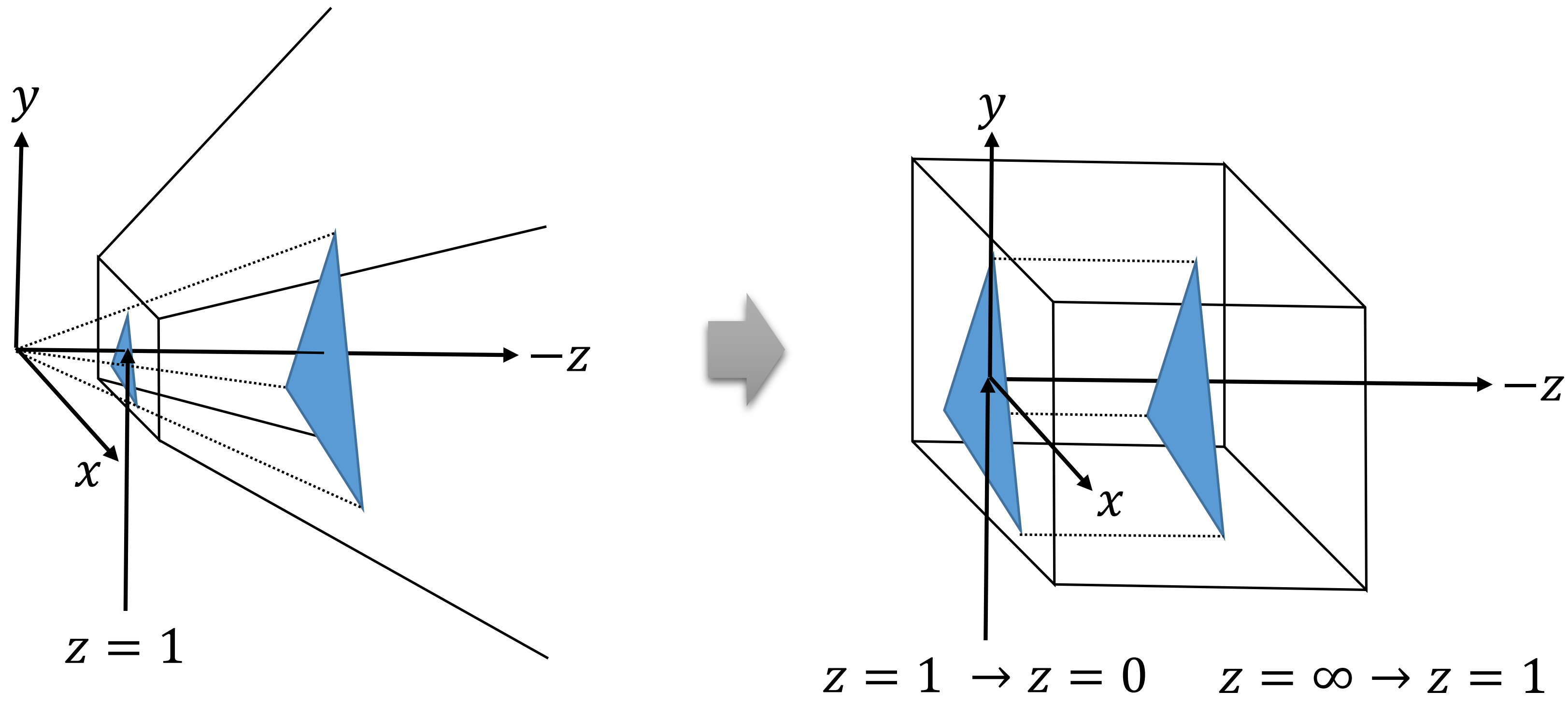
- Again, z gets lost ($z' = z/z = 1$)
- We thus use:

$$M_{perspective} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- now $z \rightarrow \frac{z-1}{z} = 1 - \frac{1}{z}$
- new depth not linear in z, but order is maintained

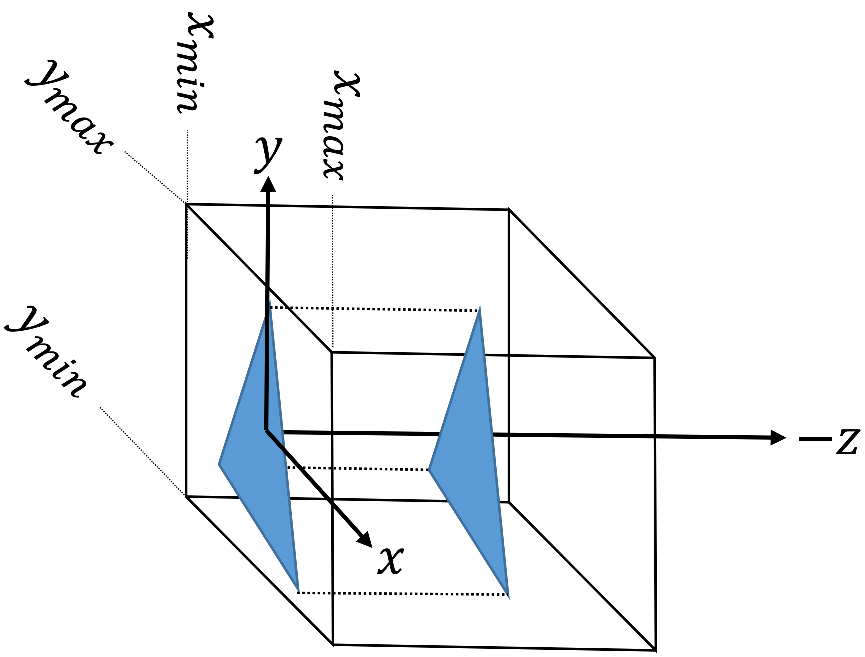
Perspective Projection

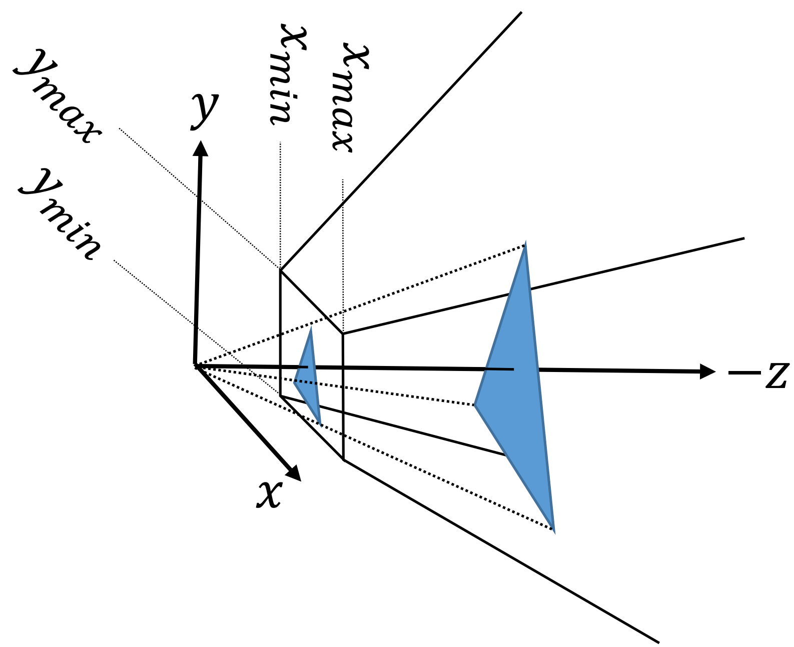
- Perspective matrix maps infinite view frustum to a box !
  - after this mapping, (x, y) are image coordinates and z is depth
  - z has non-linear to depth



Cropping

- After projection (both orthogonal and perspective)
  - x and y are image coordinates, z is depth
- Finally, we have to define
  - which window of this image plane becomes our final image
  - this image is a rectangular interval $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$
  - usually: $x_{min} = -x_{max}, y_{min} = -y_{max}$





Cropping

- to this end, we map $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ to $[-1,1]^2$:

$$M_c = \begin{pmatrix} \frac{2}{x_{max} - x_{min}} & 0 & 0 & -\frac{x_{max} + x_{min}}{x_{max} - x_{min}} \\ 0 & \frac{2}{x_{max} - x_{min}} & 0 & -\frac{y_{max} + y_{min}}{x_{max} - x_{min}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Depth

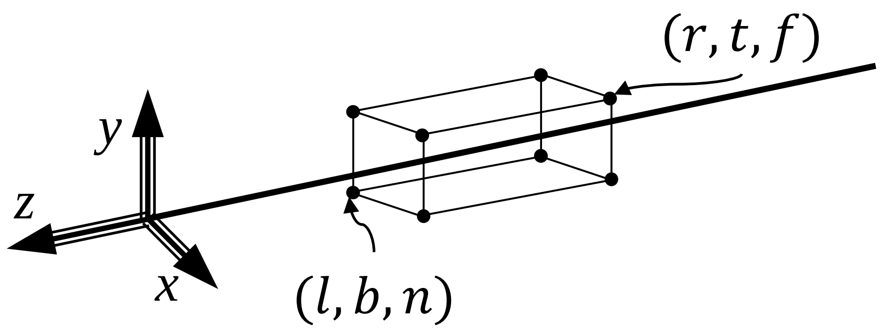
- finally, we also want to normalize depth to be in $[-1,1]$
- we define a **near plane** and a **far plane**: $z = -z_{near}$ and $z = -z_{far}$ (remember: „-z“-convention)
- linear mapping on z, such that $z_{near} \rightarrow -1$ and $z_{far} \rightarrow 1$:

$$M_c = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- choose A and B, such that
  - $z = -n$ gets mapped to $z = -1$ and
  - $z = -far$ gets mapped to $z = 1$

Projection Matrix

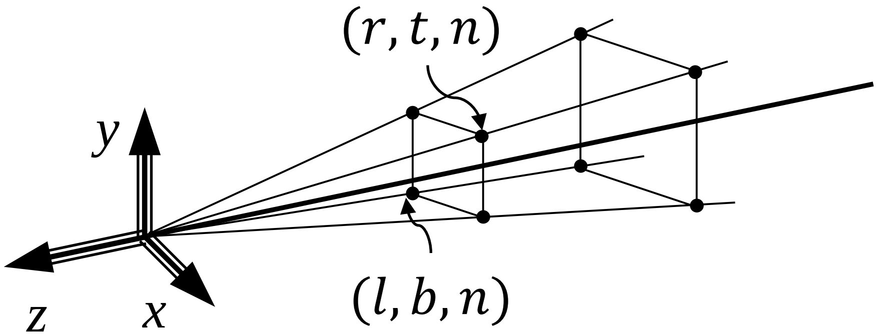
- Standard projection, cropping, and depth and merged to a single matrix, called the **Projection Matrix**
- **Orthogonal Projection:**
  In OpenGL / WebGL, the image window is defined by (l, r, b, t) (left,right,bottom,top) and the depth range by n and f



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

Projection Matrix

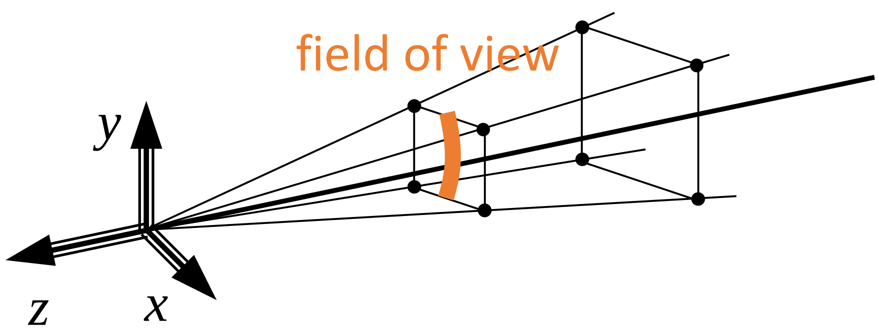
- **Perspective Projection:**
  In OpenGL / WebGL, the depth range goes from n to f (near to far), and the image window (l, r, b, t) is defined on the near-plane



- $M_{perspective}(l, r, b, t, n, f) = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$

Projection Matrix

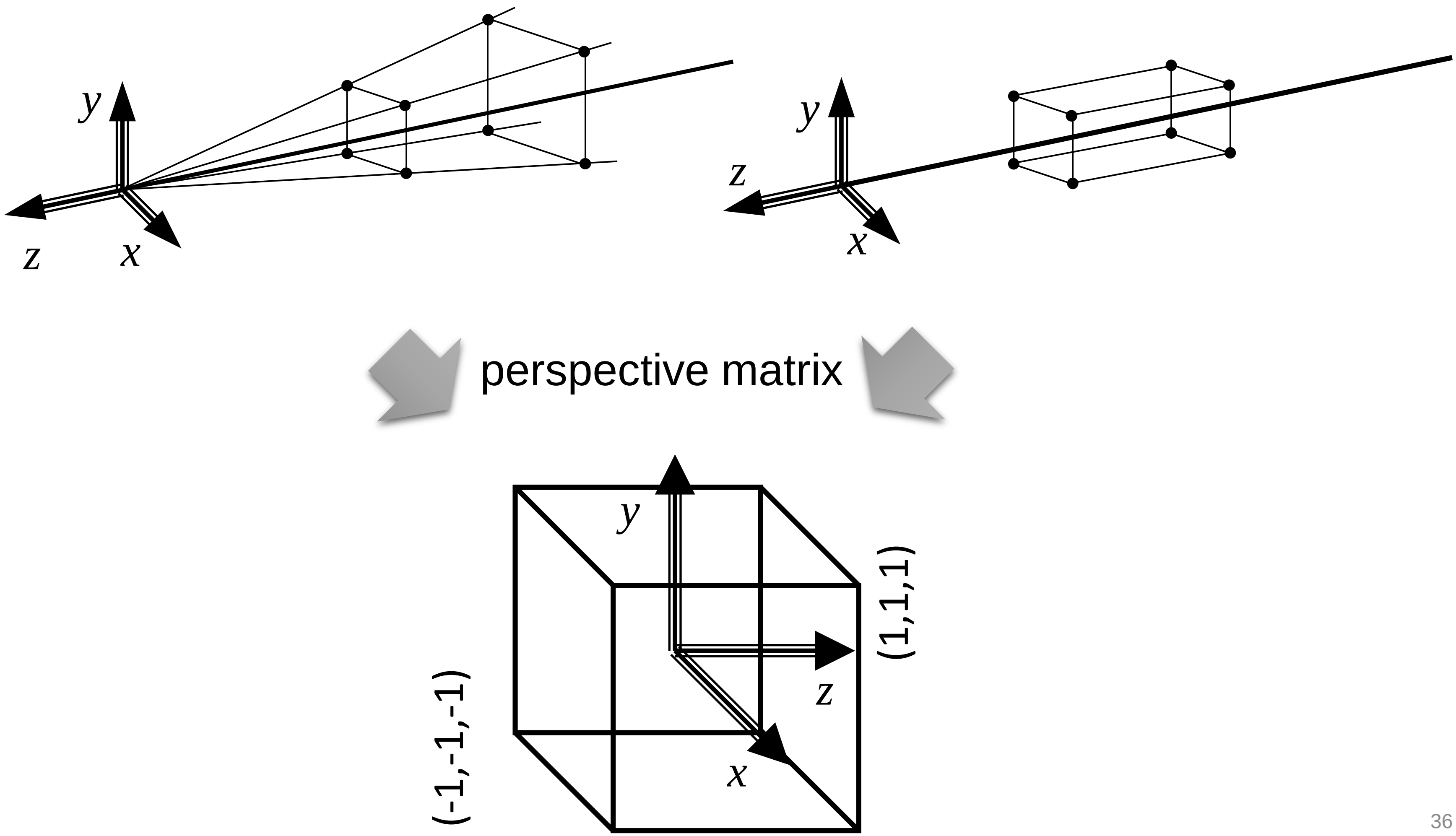
- Perspective Matrix usually defined by
  - opening angle in y-direction: field of view in y → $fovy$
  - aspect ratio: ration of width over height → $aspect$
  - near and far plane → n, f



  - $-l \;=\; r \;= aspect \cdot n \cdot \tan\frac{\mathrm{fovy}}{2}$
  - $-t = b = n \cdot \tan\frac{fovy}{2}$
- Large field of view corresponds to a wide angle lens, small field of view to a tele lens

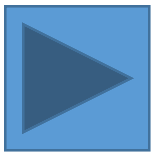
Projection

- The projection matrices transform the orthogonal and perspective view frustum into the **canonical view frustum** $[-\mathbf{1},\mathbf{1}]^{\mathbf{3}}$



Let’s play

- Viewing and projection



Viewport Transformation

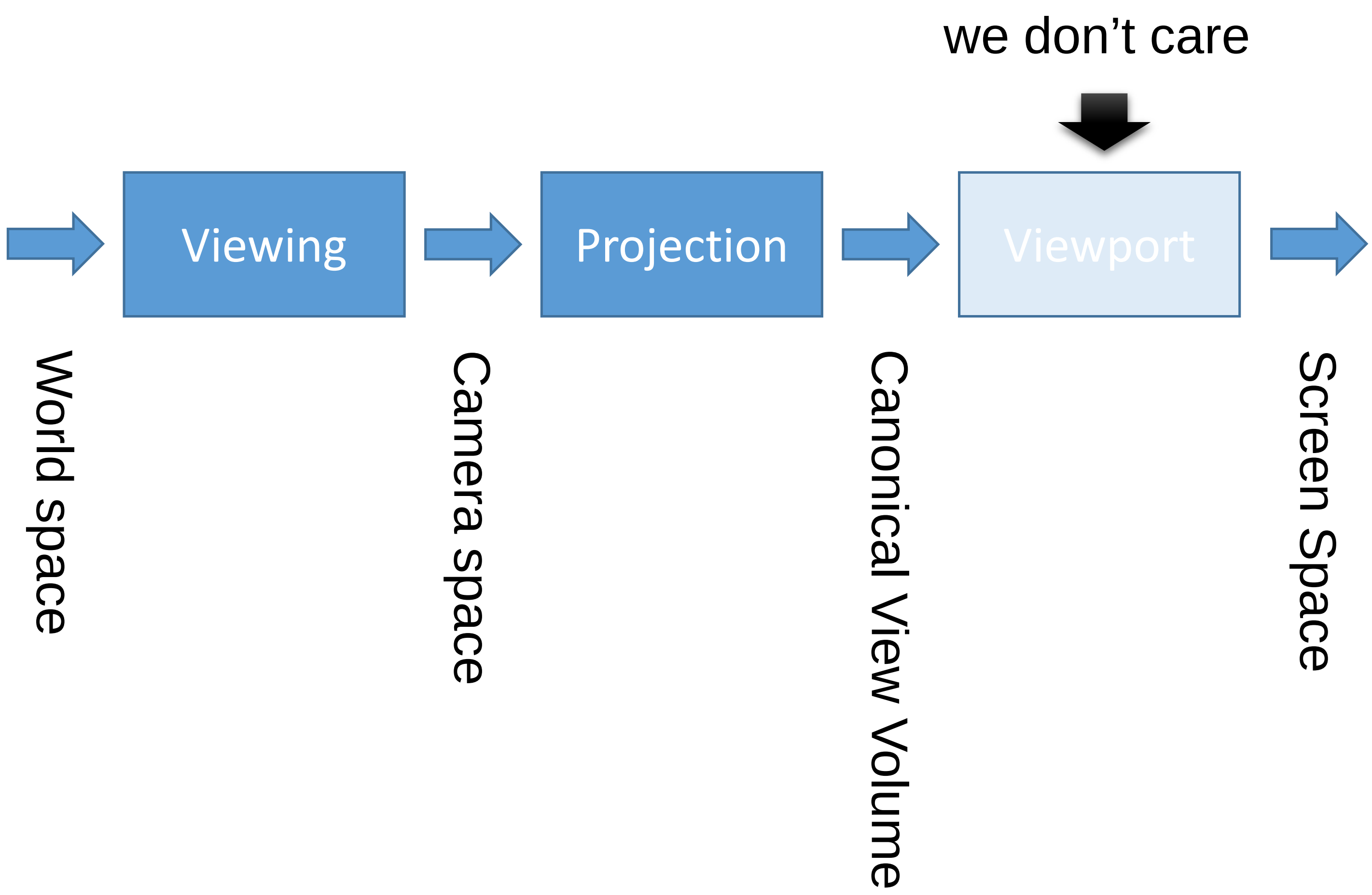
- The image has size n_x by n_y pixels and screen dimension $[-0.5, n_x - 0.5]$ in x and $[-0.5, n_y - 0.5]$ in y.
- Map points to screen coordinates
- Keep the canonical z coordinate for depth tests
- This **viewport transformation** in homogeneous coordinates is given by

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

- In OpenGL, we don't have to care about viewport transformations

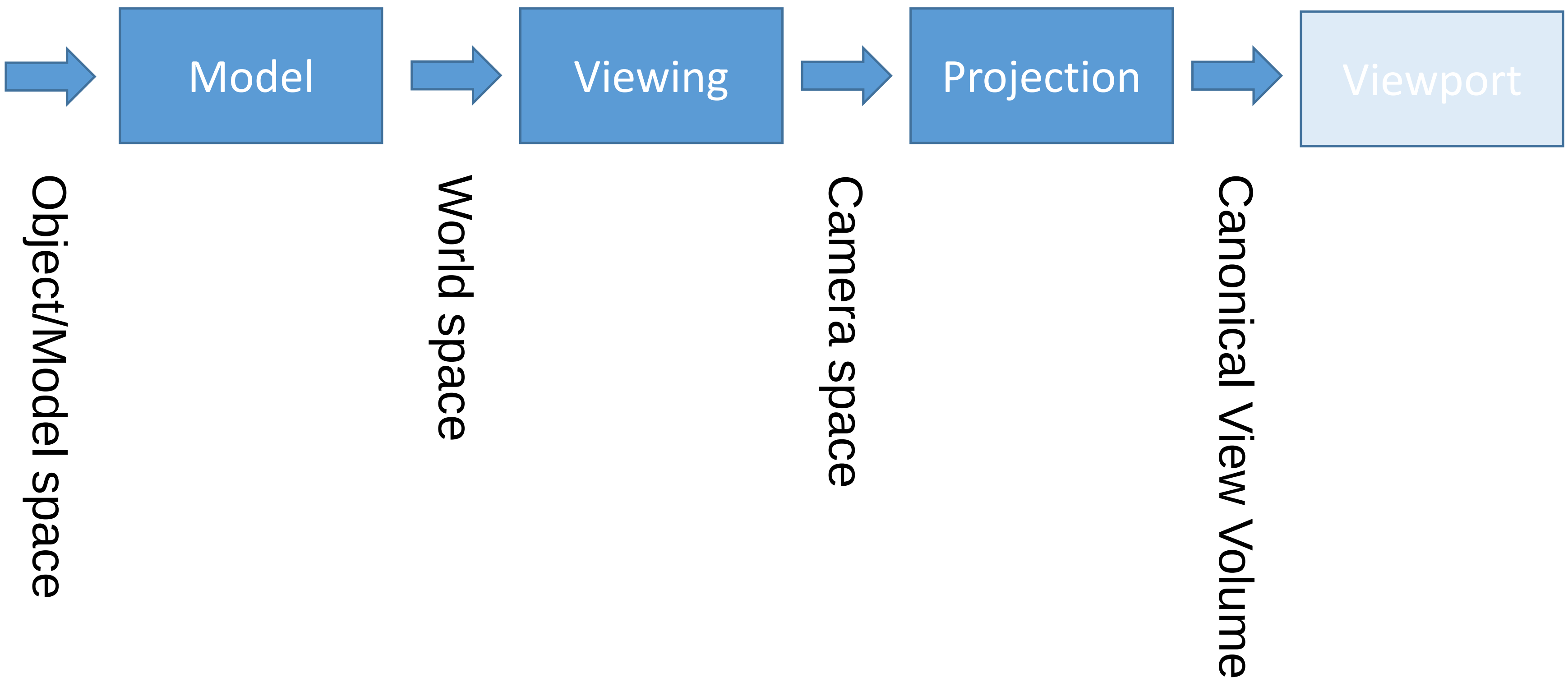
Pipeline

- Transformations in a pipeline



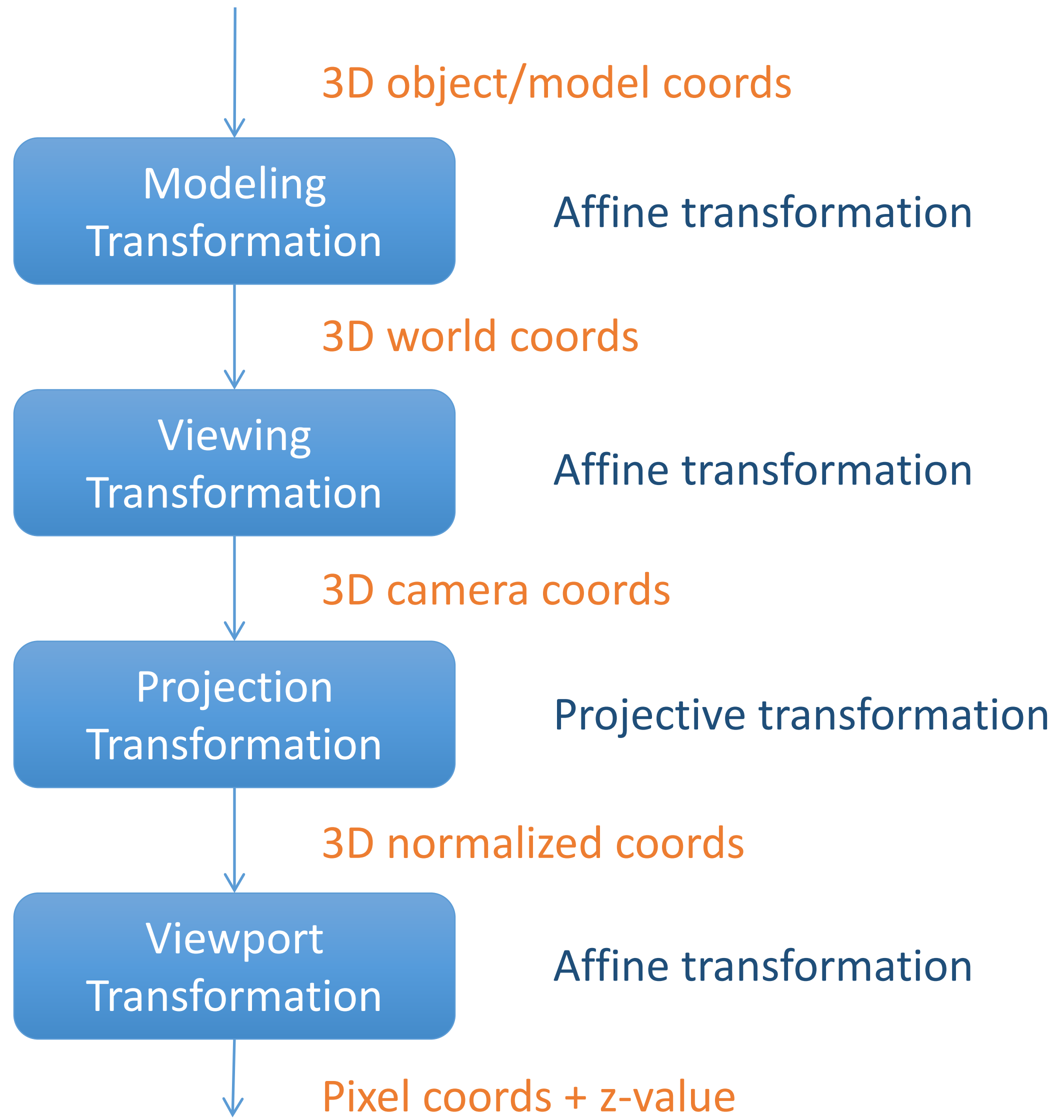
Pipeline

- Transformations in a pipeline
  - instead, we add a **model transformation**
  - this maps the local coordinates of an object to the world



Viewing Transformation

- several coordinate systems



In OpenGL / WebGL

- In old OpenGL versions, matrices where handled by OpenGL:
  - there isone matrix PROJECTION
    →orthogonal projection matrix set by:
    `glOrtho(left,right,bottom,top,near,far);`
    →perspective matrix set by
    `glFrustum(left,right,bottom,top,near,far);`
    → or by
    `gluPerspective(fovy,aspect,near,far);`

  - Viewing matrix and model matrix are stored as one MODELVIEW matrix
    → first, viewing is set using
    `gluLookAt(eyex,eyey,eyez,atx,aty,atz,upx,upy,upz);`
    where the view direction is set using a lookat point: $g = at - eye$
    → then modeling transformations can be appended, e.g. using
    `glTranslate(…), glRotate(…), glMultMatrix(…)`

In OpenGL / WebGL

- New OpenGL and WebGL do all this in the vertex shader
- So the matrix stuff must happen by the application
- In javascript: libraries, e.g. `gl-Matrix.js`
- and then upload the matrices as uniforms

Rigid Transformations

- The viewing transformation is a rigid transformation
  → rotation + translation
- The modeling transformation is usually a rigid transformation plus scale

- **Camera animation**:
  smooth transition from one viewing matrix to next one
- **Object animation**:
  smooth transition from one model matrix to next one

- Do not interpolate matrices (see previous lecture)
- instead use quaternions !
  → supported by most matrix libraries

Next Lecture

- special properties of perspective matrix