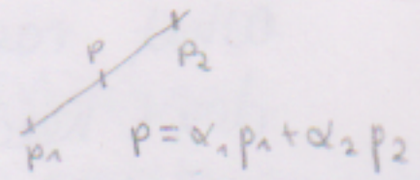
Transformations

Affine Transformations:

- an affine mapping or affine transformation is a "abbildung" between two affine spaces, where collinearity, parallelity and size ratios remain
- an affine space consists of points and is defined with an associated vectorspace V^3 → simple example: eukl. ~~vectorspace~~
- collinearity means: after an affine transformation, points that were on a line remain on a line
- parallelity means: after an affine transformation, parallel lines remain parallel
- size ratios mean: after an aff. transf. the ratios of surfaces/volumes/lines remains the same
- affine coordinates: · e.g. barycentric coordinates
  · weighting to define points



- if you now want to apply an affine transformation to points $T\bar{p} = A\bar{p} + t$ (linear transformation A (like rotations) + translation t) you can interchange applying the affine coordinates and the transformations → $T\bar{p} = T(\alpha_1 p_1 + \alpha_2 p_2) = A(\alpha_1 p_1 + \alpha_2 p_2) + t = \alpha_1 T\bar{p_1} + \alpha_2 T\bar{p_2}$
  → invariance of affine coordinates
- example: translation, rotation etc., <u>not</u>: perspective-transformations

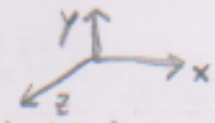
Homogeneous Coordinates

- expanding 3D-vectors into 4D-vectors $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{R}^3 \rightarrow \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \in \mathbb{R}^4$ with w mostly 1
- backtransform $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \in \mathbb{R}^4 \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \in \mathbb{R}^3$
- mathematical trick to use for 4x4 matrices to calculate transformations → later important for projections

Basic Transformations

- multiplication of a matrix with column vector $p' = Tp$
- composition: first T_1 then T_2: $p' = T_2 T_1 p = T_2(T_1 p) = (T_2 T_1)p = Tp$ (~~form~~
  → associative but not commutative (not always!)
- translation: $T = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$ shift the form by d_x in x-axis, d_y in pos. y-axis and d_z in pos. z-axis
  → attention: translation only acts on points, not on vectors
  → properties: additive, commutative, invertible

- rotation: rotation about θ around major axis

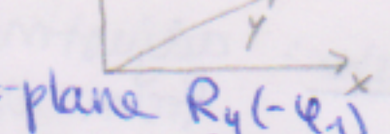


attention: here right-handed coord-system

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

→ properties: $-R^{-1}(\theta) = R(-\theta)$ $-R(\theta)R(\varphi) = R(\theta+\varphi)$ $-R_a(\theta)R_a(\varphi) = R_a(\varphi)R_a(\theta)$ but only around same axis

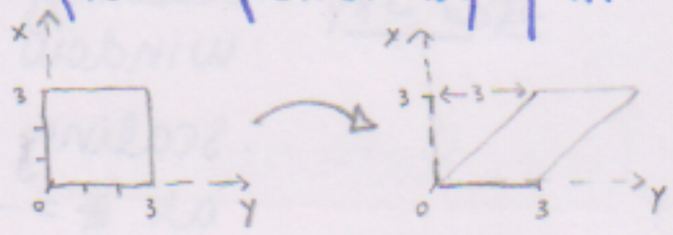
- scaling: $S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ scaling in x-dir by s_x, in y-dir by s_y, in z-dir by s_z



- reflection: at z $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = M_z$

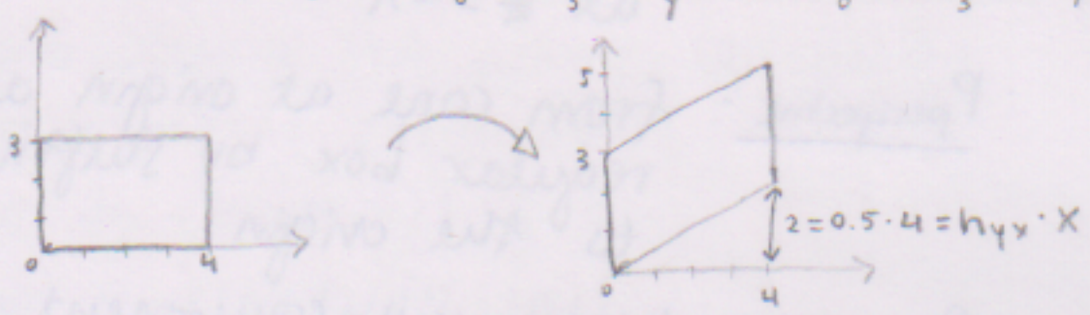
- shearing: $\begin{pmatrix} 1 & h_{xy} & h_{xz} & 0 \\ h_{yx} & 1 & h_{yz} & 0 \\ h_{zx} & h_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ where h_{xy} shifts a point by y in x-direction

e.g. $\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$:



skipped transform normals

e.g. $\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}_{2D}$



- rotation around arbitrary axis:

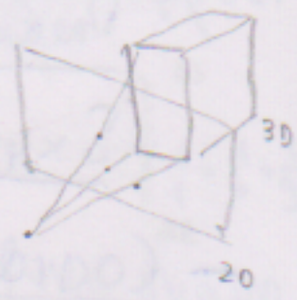
1) move base point to origin T(-o)
2) rotate around y-axis, so that r is in yz-plane $R_y(-\varphi_1)$ → r'
3) rotate around x-axis, so that r' is along z-axis $R_x(\varphi_2)$
4) rotate around z-axis with angle φ
5) rotate back around x-axis, around y-axis, translate back

$\tan(\varphi_1) = r_x / r_z$
$\tan(\varphi_2) = r'_y / r_z$

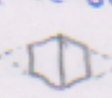
→ $R(\varphi, \vec{o}, \vec{r}) = T(o) R_y(\varphi_1) R_x(-\varphi_2) R_z(\varphi) R_x(\varphi_2) R_y(-\varphi_1) T(-o)$

Projections

- definition: mapping from 3D to 2D with a loss of information → not invertible

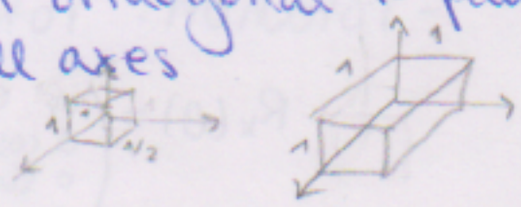


- perspective projection: projection along lines onto a proj. plane where the lines intersect in a single point
  → not affine

one point  two-point  three-point 

- parallel projection: · special case of perspective (proj. point at infinity)
  · axonometric projection
    isometric: same angle and length at every axis
    dimetric: same angle with 2 axis, 2 same length
    trimetric: same angle with 1 axis, 1 same length

- oblique/sheared projection (projection direct. not orthogonal to plane)
  cavalier projection: same length on all axes
  cabinet projection: shortening of 1/2

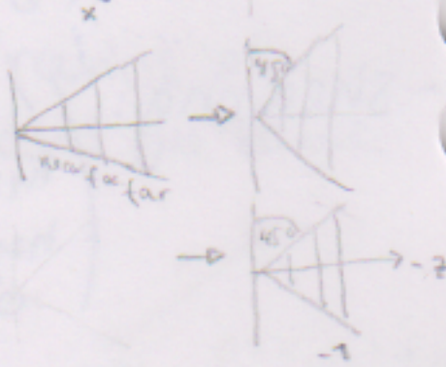


Camera Transformation "How to compute transf. between points in 3D and pixels on screen"

– $K = T_{raster}\, S_{raster}\, P_{parallel}\, P_{perspective}\, S_{far}\, S_{xy}\, H\, RT$ (perspective)

- RT: translation to the origin and rotation so that the viewing direction is along the neg. z-axis
- H: shear such that the viewing direction is along z-axis (window center on -z-axis)

- $S_{far}\, S_{xy}$: scaling in x and y such that screen window opens at 45 degrees
  scaling in z such that far plane is at $z = -1$

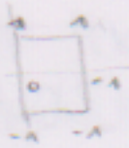


- $P_{perspective}$: from cone at origin around z-axis to regular box by shifting the near plane to the origin
- $P_{parallel}$: keeps x,y-component, cancels out z
  scaling by $\frac{1}{2}$ in x,y and translation by $\frac{1}{2}$ in x,y → 2D (0,0) to (1,1)
- S_{raster}: adjustment of the aspect ratio, not only from 0,0 to 1,1 but to dimension of image
- T_{raster}: positioning on the screen

} viewport transformation

– $K = T_{raster}\, S_{raster}\, P_{parallel}\, S_{xyz}\, T_{near}\, RT$ (orthographic)

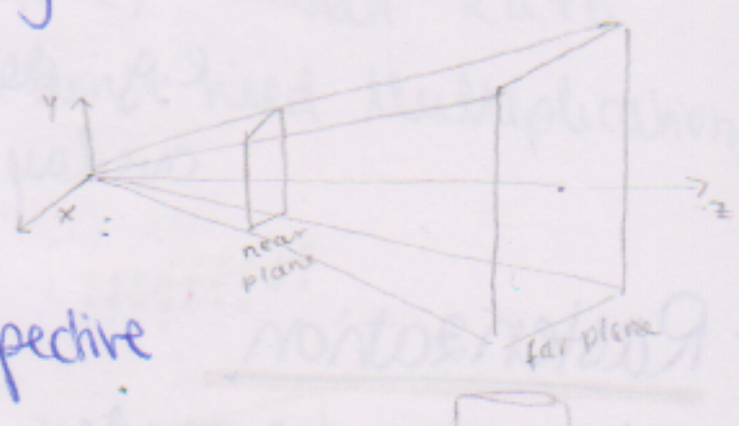
- T_{near}: translation to bring near plane into origin
- S_{xyz}: scaling to regular box



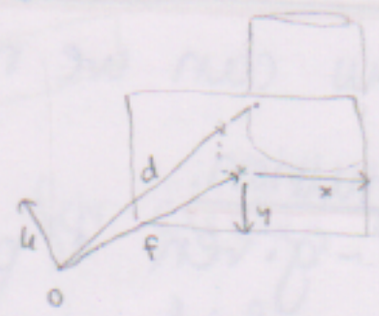
– Coordinate Systems:
- Local/Object coordinate system (3D)
  → defined by normals
  → by model transformation to world coords.
- World/Global coordinate system (3D)
  → scene composition and object placements with rigid and animated objects
  → by viewing transformation to camer. coords
- Camera/Viewing coordinate system (3D)
  → camera pos and direction
  → illumination & shading to be computed here
  → by perspective transformation to normalized coords

· Normalized device coordinate system (2D)
  → normalization to viewing frustum
  → rasterization

- Camera Model: · view frustum (seen right) is the region of space in the modeled world that may appear of view (perspective camera system)

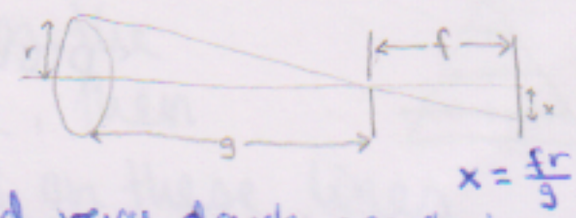


· camera definition in ray traces
  o is origin, f is focal length (vector to centr), u is up-vector (not necassarily parallel to plane), x, y is half of view window



exercise: derive the general equation for a ray that goes through an arbitrary pixel

· pinhole camera model
  · only small hole for ease of implementation → in real world very dark, many lenses.



$x = \frac{fr}{g}$

· lens camera model
  · like in real world, object in focus, if center of confusion smaller than the pixelsize (sharp if $\Delta s > \Delta e$) still sharp
  · the smaller the apature (a) the larger the depth of field (range of depth where objects are in focus).
  → therefore smartphone cameras have large dof

$\frac{1}{f} = \frac{1}{b} + \frac{1}{g}$

- Backface Culling: · check if polygons are seen from front or back
  front: $n \cdot v > 0$ with n polygon-normal & v sight-vector
  back: $n \cdot v < 0$
  → draw only the visible surfaces of closed objects
  → to reduce calculations

- limitations: . a pinhole-camera as used in image computation is missing some features of real-world-lens cameras
  . e.g.: depth-of-field, vignetting (fix with darker overlay), flare of the sun

Rasterization

"How can we render 2D or 3D graphic very time-efficient?"

Definition:
- given a primitive form (like a 2D line, circle, polygon), which pixels on a raster display are covered by this primitive → extension: which part of pixel
- it's used for realtime-rendering (e.g. for computer games)

Line Rasterization:

- assume: . pixels are points on 2D-integer-grid
  . endpoints of lines at pixel coordinates
  . slope (steigung) of lines is ≤ 1
  . line thickness is one pixel
- functions: . define lines with startpoint (x_0, y_0) & endpoint (x_e, y_e) through function $y = mx + b$ with $m = \frac{dy}{dx}$
  . algorithm: brute-force, that means iterate over x-values (x_0 to x_e) then calculate y
  . problem: m and y_i are floating-point values and multiplications are expensive
- digital differential analyzer (dda): . same definition of lines
  . algorithm: incremental, that means we increment $x + 1$ and $y + m$
  → avoiding multiplication
  → but rounding errors accumulate
- bresenham: . define lines implicitly: $F(x,y) = dy\,x - dx\,y + dx\,B = 0$
  → if $F(x,y) < 0$ point above line, $F(x,y) > 0$ below
  . algorithm: midpoint formulation, measure ther vertical distance of midpoint M to line $d_{i+1} = F(M_{i+1}) = F(x_i + 1, y_i + \frac{1}{2})$
  → increment $x + 1$ and if $d_i > 0$ also increment y