

Computer Graphics

Camera & Projective Transformations

Philipp Slusallek

Motivation

- **Rasterization works on 2D primitives (+ depth)**
- **Need to project 3D world onto 2D screen**
- **Based on**
 - Positioning of objects in 3D space
 - Positioning of the virtual camera

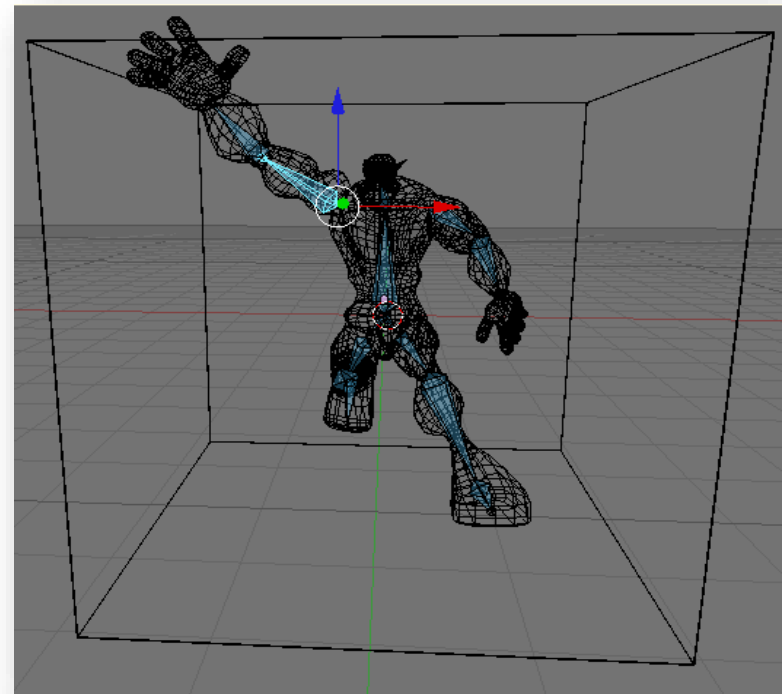
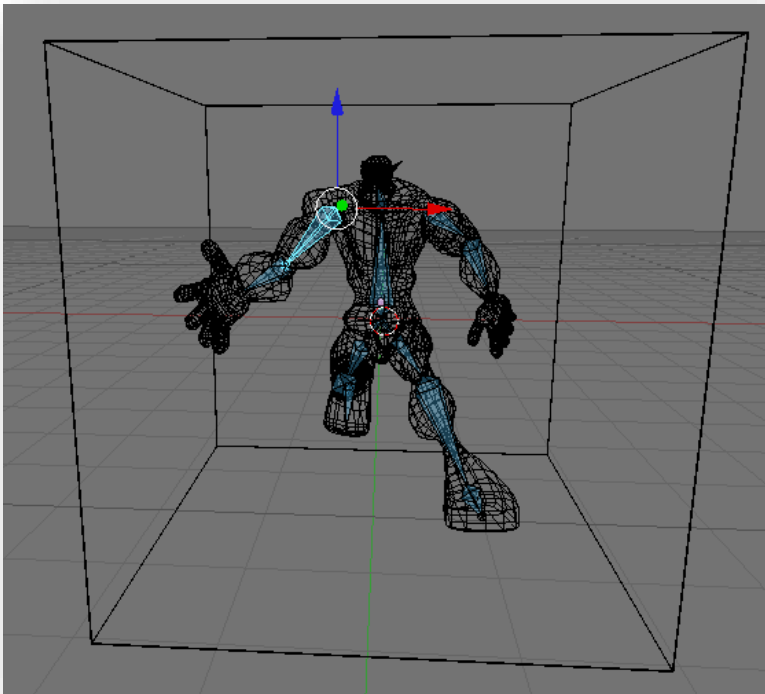
Coordinate Systems

- **Local (object) coordinate system (3D)**
 - Object vertex positions
 - Can be **hierarchically nested** in each other (scene graph, transf. stack)
- **World (global) coordinate system (3D)**
 - Scene composition and object placement
 - Mostly rigid objects: constant translation, rotation per object, (scaling)
 - Animated objects: time-varying transformation in world or local space
 - Illumination can be computed in this space
- **Camera/view/eye coordinate system (3D)**
 - Coordinates relative to camera pose (position & orientation)
 - Camera itself specified relative to world space
 - Illumination can also be done in this space
- **Normalized device coordinate system (2.5D)**
 - After *perspective transformation*, rectilinear, in $[0, 1]^3$
 - Normalization to view frustum (for rasterization and depth buffer)
 - Shading executed here (interpolation of color across triangle)
- **Window/screen (raster) coordinate system (2D)**
 - 2D transformation to place image in window on the screen

Hierarchical Coordinate Systems

- **Used in Scene Graphs**

- Group objects hierarchically
- Local coordinate system is relative to parent coordinate system
- Apply transformation to the parent to change the whole sub-tree (or sub-graph)



Hierarchical Coordinate Systems

- **Hierarchy of transformations**

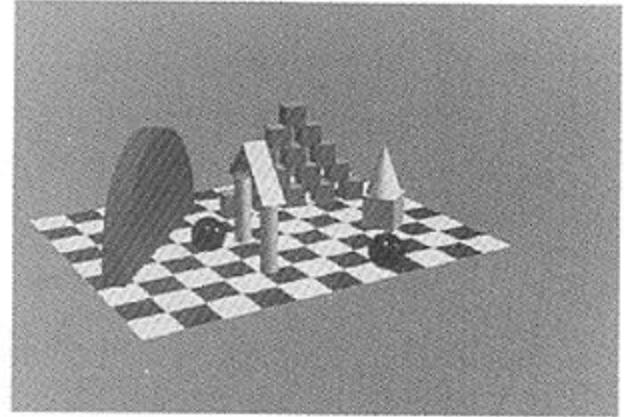
T_root	Positions the character in the world	
T_ShoulderR	Moves to the right shoulder	
T_ShoulderRJoint	Rotates in the shoulder	<== User
T_UpperArmR	Moves to the Elbow	
T_ElbowRJoint	Rotates in the Elbow	<== User
T_LowerArmR	Moves to the wrist	
T_WristRJoint	Rotates in the wrist	<== User
.....	Further for the right hand and the fingers	
T_ShoulderL	Moves to the left shoulder	
T_ShoulderLJoint	Rotates in the shoulder	<== User
T_UpperArmL	Moves to the Elbow	
T_ElbowLJoint	Rotates in the Elbow	<== User
T_LowerArmL	Moves to the wrist	
.....	Further for the left hand and the fingers	

- Each transformation is relative to its parent
 - Concatenated by multiplying and pushing onto a stack
 - Going back by popping from the stack
 - This transformation stack was so common, it was built into OpenGL
-

Coordinate Transformations

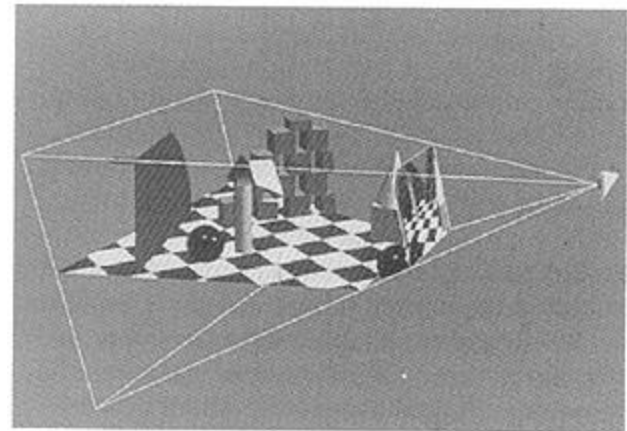
- **Model transformation**

- Object space to world space
- Can be hierarchically nested
- Typically an affine transformation



- **View transformation**

- World space to eye space
- Typically an affine transformation

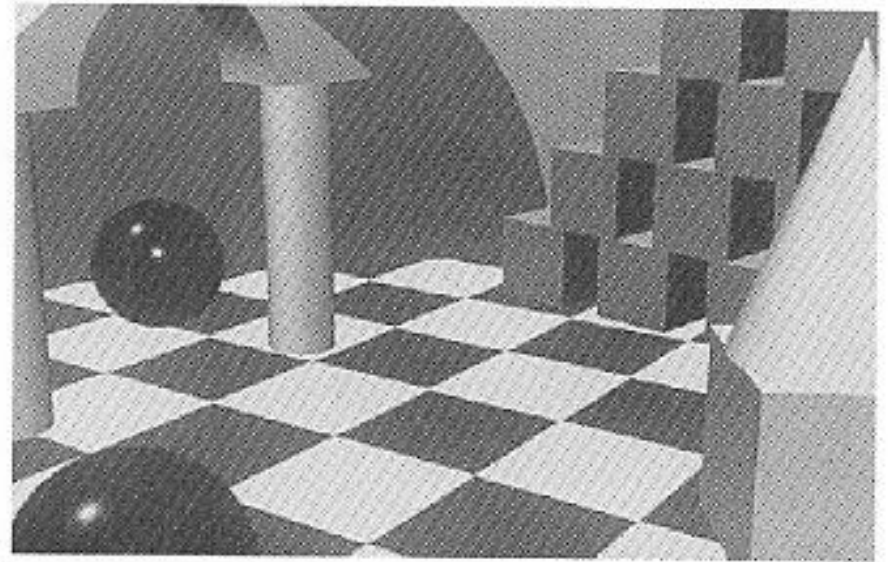
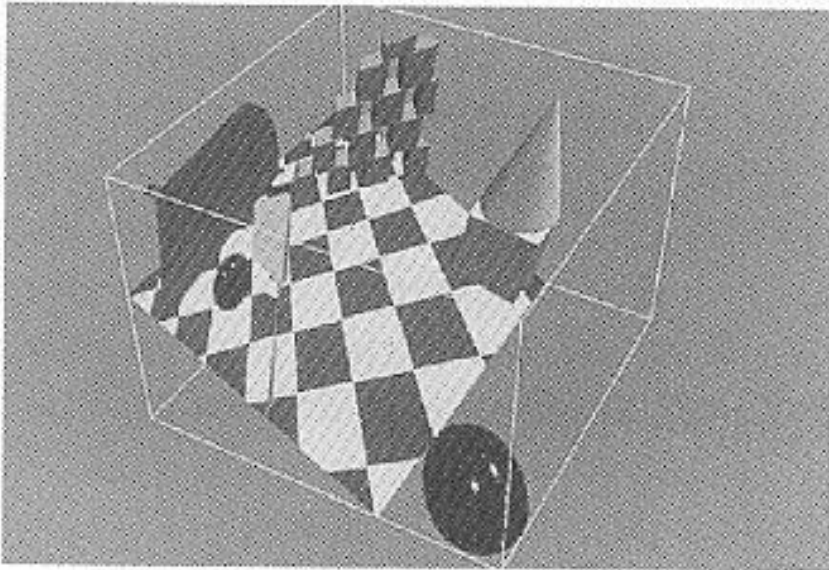


- **Combination: Modelview transformation**

- Used by *traditional* OpenGL (although world space is conceptually intuitive, it was not explicitly exposed in OpenGL)

Coordinate Transformations

- **Projective transformation**
 - Eye space to normalized device space (defined by view frustum)
 - Parallel or perspective projection
 - 3D to 2D: Preservation of depth in Z coordinate
- **Viewport transformation**
 - Normalized device space to window (raster) coordinates



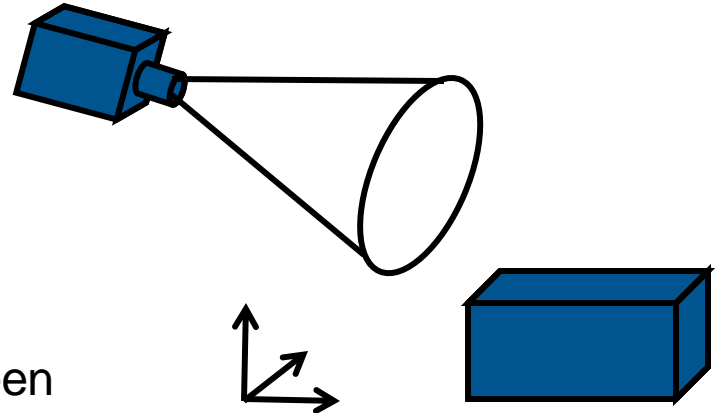
Camera & Perspective Transforms

- **Goal**

- Compute the transformation between points in 3D and pixels on the screen
- Required for rasterization algorithms (OpenGL)
 - They project all primitives from 3D to 2D
 - Rasterization happens in 2D (actually 2.5D, XY plus Z attribute)

- **Given**

- Camera pose (pos. & orient.)
 - *Extrinsic* parameters
- Camera configuration
 - *Intrinsic* parameters
- Pixel raster description
 - Resolution and placement on screen

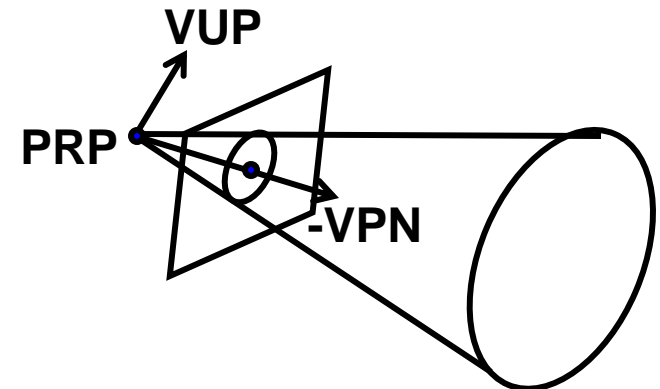


- **In the following: Stepwise Approach**

- Express each transformation step in homogeneous coordinates
- Multiply all 4x4 matrices to combine all transformations

Viewing Transformation

- **Need camera position and orientation in world space**
 - External (extrinsic) camera parameters
 - Center of projection: projection reference point (PRP)
 - Optical axis: view-plane normal (VPN)
 - View up vector (VUP)
 - Not necessarily orthogonal to VPN, but not co-linear
- **Needed Transformations**
 - 1) Translation of PRP to the origin (-PRP)
 - 2) Rotation such that viewing direction is along negative Z axis
 - 2a) Rotate such that VUP is pointing up on screen



Perspective Transformation

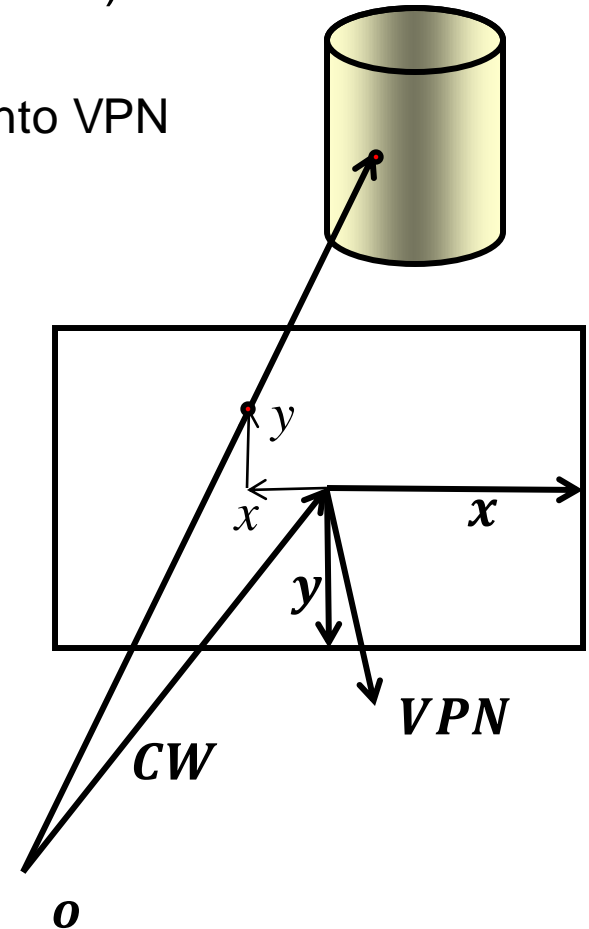
- **Define projection (perspective or orthographic)**
 - Needs internal (intrinsic) camera parameters
 - Screen window (Center Window (CW), width, height)
 - Window size/position on image plane (relative to VPN intersection)
 - Window center relative to PRP determines viewing direction (\neq VPN)
 - Focal length (f)
 - Distance of projection plane from camera along VPN
 - Smaller focal length means larger field of view
 - Field of view (fov) (defines width of view frustum)
 - Often used instead of screen window and focal length
 - Only valid when screen window is centered around VPN (often the case)
 - Vertical (or horizontal) angle plus aspect ratio (width/height)
 - Or two angles (both angles may be half or full angles, beware!)
 - Near and far clipping planes
 - Given as distances from the PRP along VPN
 - Near clipping plane avoids singularity at origin (division by zero)
 - Far clipping plane restricts the depth for fixed-point representation in HW

Simple Camera Parameters

- **Camera definition (typically used in ray tracers)**

- $o \in \mathbb{R}^3$: center of projection, point of view (PRP)
- $CW \in \mathbb{R}^3$: vector to center of window
 - “Focal length”: projection of vector to CW onto VPN
 - $focal = |(CW - o) \cdot VPN|$
- $x, y \in \mathbb{R}^3$: span of half viewing window
 - $VPN = (\mathbf{y} \times \mathbf{x}) / |(\mathbf{y} \times \mathbf{x})|$
 - $VUP = -\mathbf{y}$
 - $width = 2|x|$
 - $height = 2|y|$
 - Aspect ratio: $camera_{ratio} = |x|/|y|$

PRP: Projection reference point
VPN: View plane normal
VUP: View up vector
CW: Center of window



Viewport Transformation

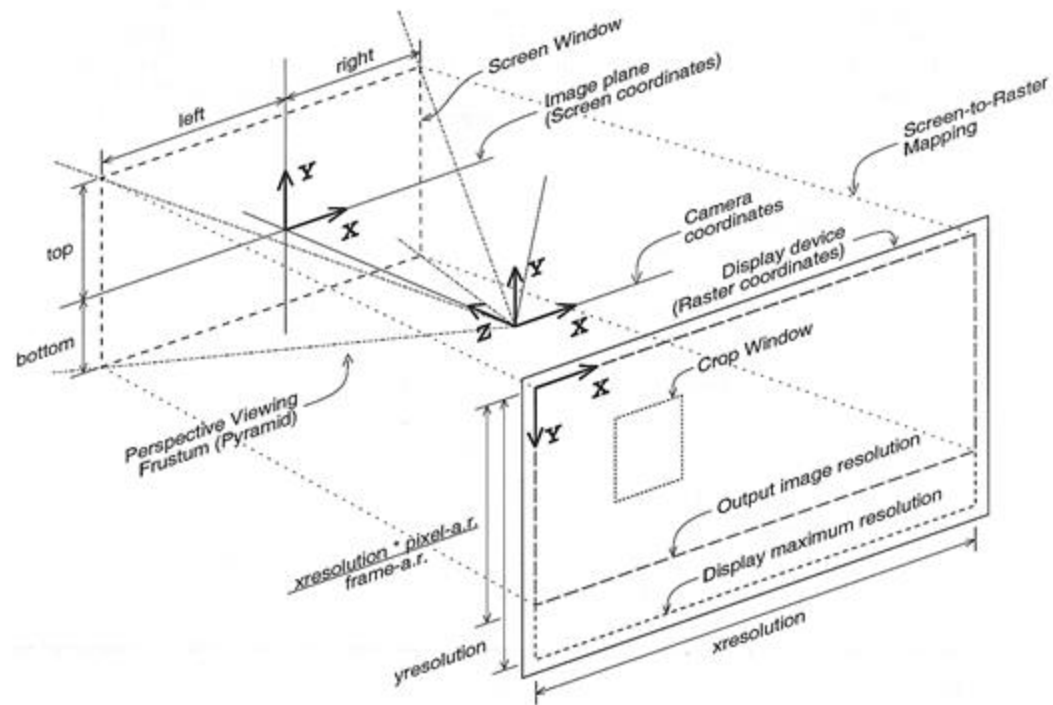
- **Normalized Device Coordinates (NDC)**
 - Intrinsic camera parameters transform to NDC
 - $[0,1]^2$ for x, y across the screen window
 - $[0,1]$ for z (depth)
- **Mapping NDC to raster coordinates on the screen**
 - $xres, yres$: Size of window in pixels
 - Should have same aspect ratios to avoid distortion
 - $camera_{ratio} = \frac{xres \text{ pixelspacing}_x}{yres \text{ pixelspacing}_y}$,
 - Horizontal and vertical pixel spacing (distance between centers)
 - Today, typically the same but can be different e.g. for some video formats
 - Position of window on the screen
 - Offset of window from origin of screen
 - $posx$ and $posy$ given in pixels
 - Depends on where the origin is on the screen (top left, bottom left)
 - “Scissor box” or “crop window” (region of interest)
 - No change in mapping but limits which pixels are rendered

Camera Parameters: Rend.Man

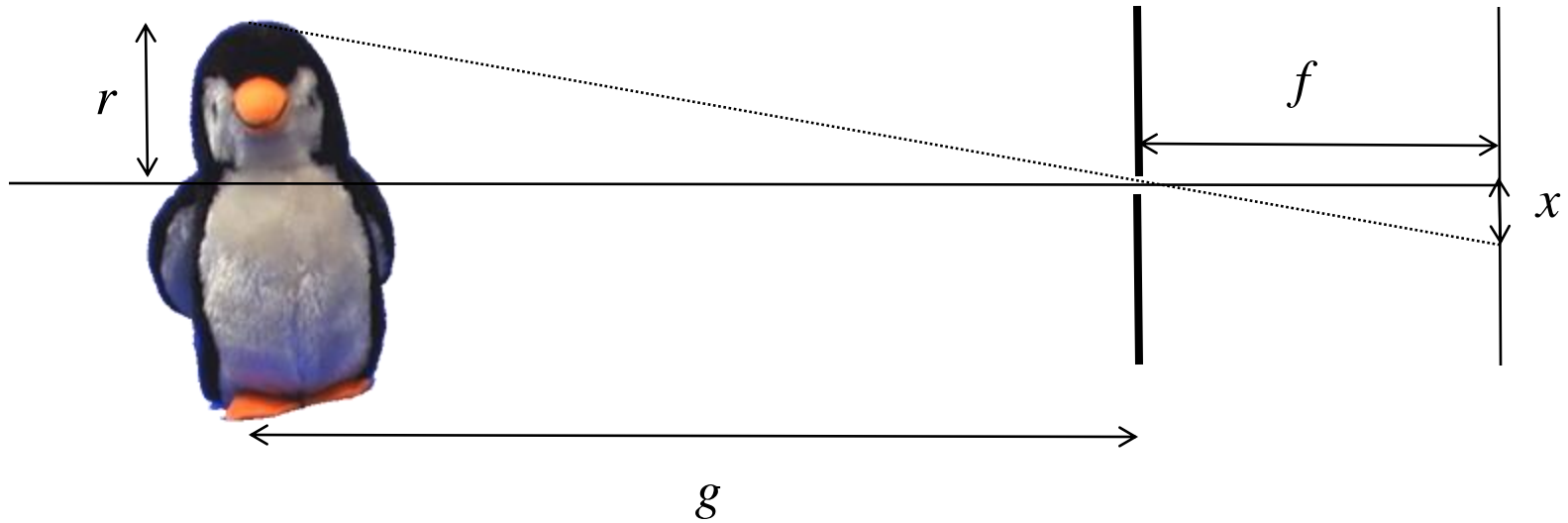
- **RenderMan camera specification**

- Almost identical to above description

- Distance of Screen Window from origin given by “field of view” (fov)
 - fov: Full angle of segment $(-1,0)$ to $(1,0)$, when seen from origin
- CW given implicitly
- No offset on screen



Pinhole Camera Model

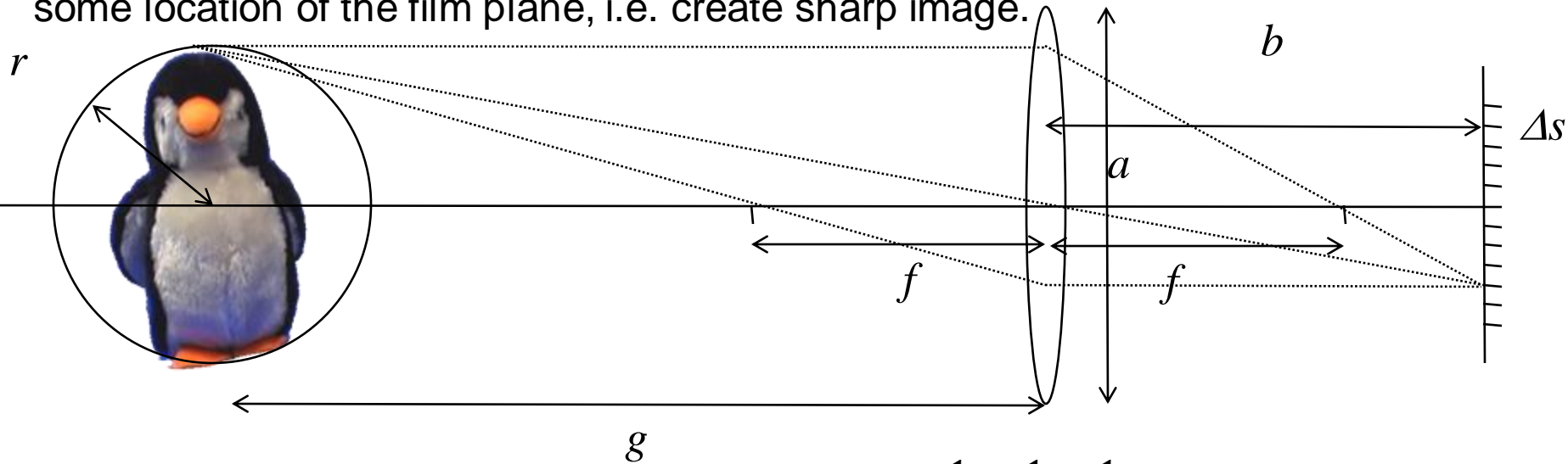


$$\frac{r}{g} = \frac{x}{f} \Rightarrow x = \frac{fr}{g}$$

- Infinitesimally small pinhole**
- ⇒ Theoretical (non-physical) model
 - ⇒ Sharp image everywhere
 - ⇒ Infinite depth of field
 - ⇒ Infinitely dark image in reality
 - ⇒ Diffraction effects in reality

Thin Lens Model

Lens focuses light from given position on object through finite-size aperture onto some location of the film plane, i.e. create sharp image.



Lens formula defines reciprocal focal length (focus distance from lens of parallel light)

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

Object center at distance g is in focus at

$$b = \frac{fg}{g - f}$$

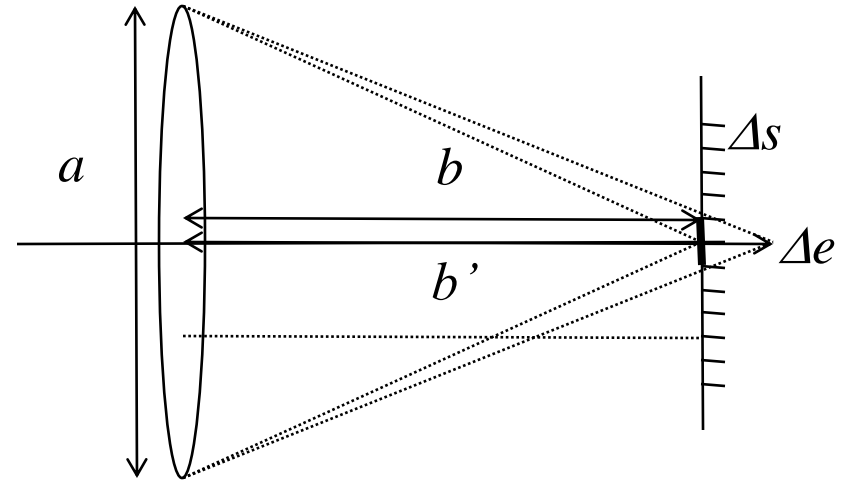
Object front at distance $g-r$ is in focus at

$$b' = \frac{f(g - r)}{(g - r) - f}$$

Thin Lens Model: Depth of Field

Circle of confusion
(CoC)

$$\Delta e = \left| a \left(1 - \frac{b}{b'} \right) \right|$$



Sharpness criterion based
on pixel size and CoC

$$\Delta s > \Delta e$$

DOF: Defined radius r , such that CoC smaller than Δs

Depth of field (DOF)

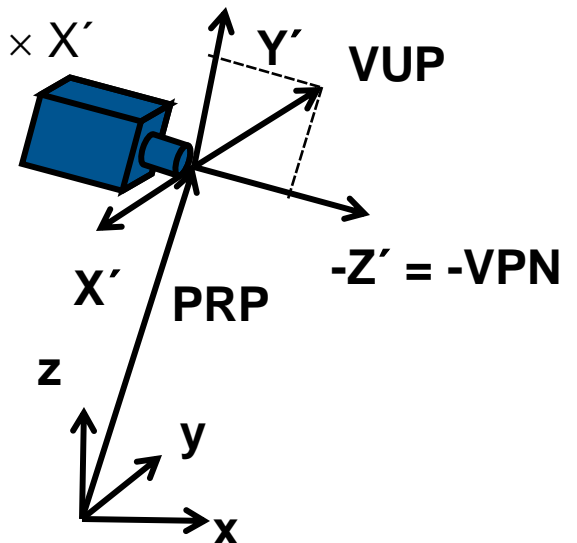
$$r < \frac{g\Delta s(g-f)}{af + \Delta s(g-f)} \Rightarrow r \propto \frac{1}{a}$$

The smaller the aperture, the larger the depth of field

Viewing Transformation

- **Let's put this all together**
- **Goal: Camera: at origin, view along $-Z$, Y upwards**
 - Assume right handed coordinate system
 - Translation of PRP to the origin
 - Rotation of VPN to Z-axis
 - Rotation of projection of VUP to Y-axis
- **Rotations**
 - Build orthonormal basis for the camera and form inverse
 - $Z' = \text{VPN}$, $X' = \text{normalize}(\text{VUP} \times \text{VPN})$, $Y' = Z' \times X'$
- **Viewing transformation**
 - Translation followed by rotation

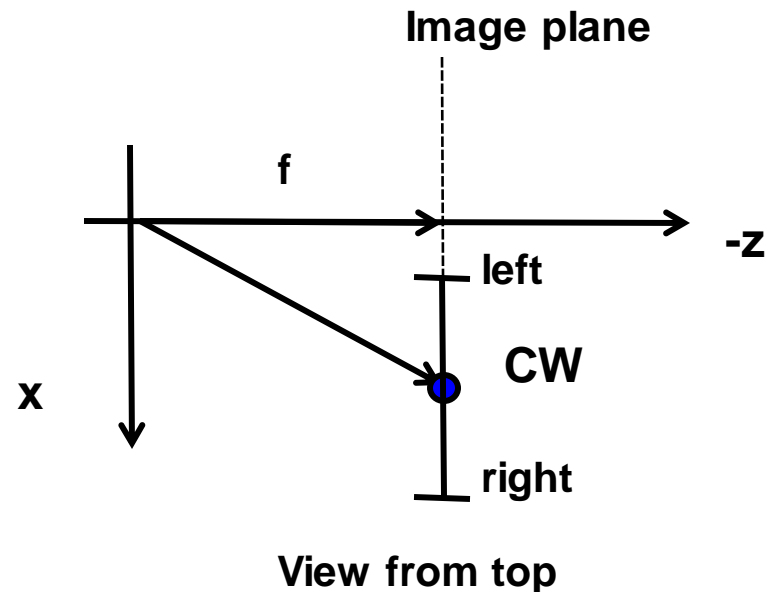
$$V = RT = \begin{pmatrix} X'_x & Y'_x & Z'_x & 0 \\ X'_y & Y'_y & Z'_y & 0 \\ X'_z & Y'_z & Z'_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^T T(-PRP)$$



Sheared Perspective Transformation

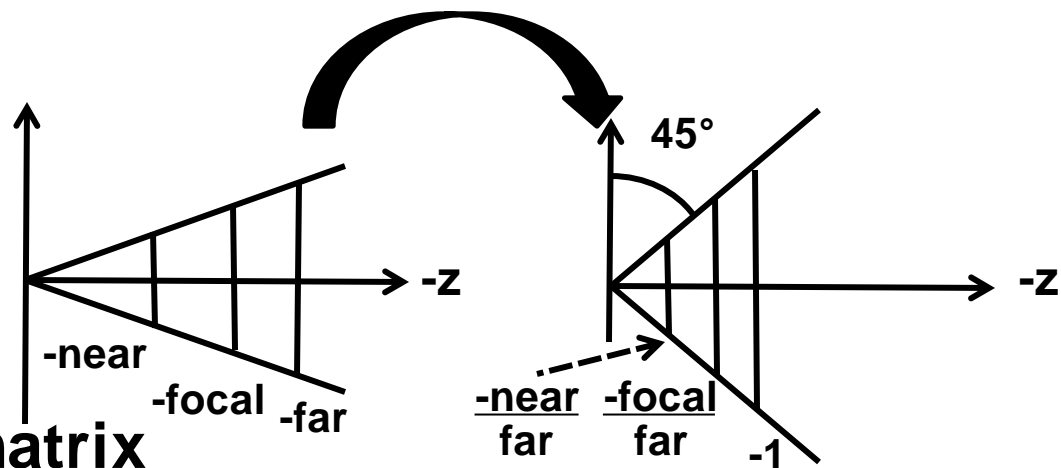
- **Step 1: VPN may not go through center of window**
 - Oblique viewing configuration
- **Shear**
 - Shear space such that window center is along Z-axis
 - Window center CW (in 3D view coordinates)
 - $CW = ((right+left)/2, (top+bottom)/2, -focal)^T$
- **Shear matrix**

$$H = \begin{pmatrix} 1 & 0 & -\frac{CW_x}{CW_z} & 0 \\ 0 & 1 & -\frac{CW_y}{CW_z} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Normalizing

- **Step 2: Scaling to canonical viewing frustum**
 - Scale in X and Y such that screen window boundaries open at 45 degree angles (at focal plane)
 - Scale in Z such that far clipping plane is at Z= -1



- **Scaling matrix**

$$- S = S_{far} S_{xy} = \begin{pmatrix} \frac{1}{far} & 0 & 0 & 0 \\ 0 & \frac{1}{far} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{2focal}{width} & 0 & 0 & 0 \\ 0 & \frac{2focal}{height} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Transformation

- **Step 3: Perspective transformation**

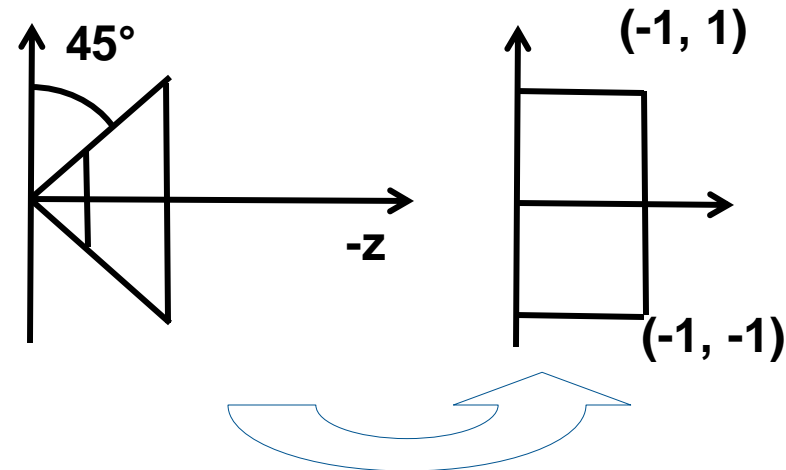
- From canonical perspective viewing frustum (= cone at origin around -Z-axis) to regular box $[-1 .. 1]^2 \times [0 .. 1]$

- **Mapping of X and Y**

- Lines through the origin are mapped to lines parallel to the Z-axis
 - $x' = x/-z$ and $y' = y/-z$ (coordinate given by slope with respect to z!)
- Do not change X and Y additively (first two rows stay the same)
- Set W to $-z$ so we divide when converting back to 3D
 - Determines last row

- **Perspective transformation**

- $$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \boxed{A} & \boxed{B} & \boxed{C} & \boxed{D} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$
 Still unknown



- **Note: Perspective projection = perspective transformation + parallel projection**

Perspective Transformation

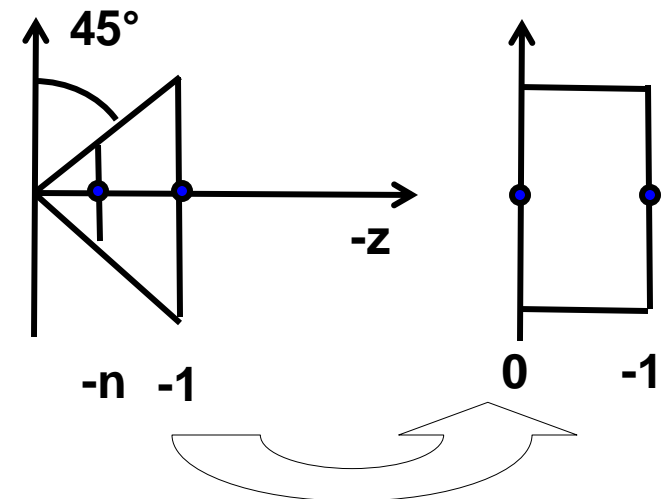
- **Computation of the coefficients A, B, C, D**
 - No shear of Z with respect to X and Y
 - $A = B = 0$
 - Mapping of two known points
 - Computation of the two remaining parameters C and D
 - $n = \text{near} / \text{far}$ (due to previous scaling by $1/\text{far}$)
 - Following mapping must hold
 - $(0,0,-1,1)^T = P(0,0,-1,1)^T$ and $(0,0,0,1) = P(0,0,-n,1)$

- **Resulting Projective transformation**

- $$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1-n} & \frac{n}{1-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- **Transforms Z non-linearly (in 3D)**

- $$z' = -\frac{z+n}{z(1-n)}$$



Parallel Projection to 2D

- **Parallel projection to $[-1 .. 1]^2$**
 - Formally scaling in Z with factor 0
 - Typically maintains Z in $[0, 1]$ for depth buffering
 - As a vertex attribute (see OpenGL later)
- **Transformation from $[-1 .. 1]^2$ to NDC ($[0 .. 1]^2$)**
 - Scaling (by $1/2$ in X and Y) and translation (by $(1/2, 1/2)$)
- **Projection matrix for combined transformation**
 - Delivers normalized device coordinates

$$\bullet P_{parallel} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 \text{ or } 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Viewport Transformation

- **Scaling and translation in 2D**

- Scaling matrix to map to entire window on screen

- $S_{raster}(xres, yres)$

- No distortion if aspects ration have been handled correctly earlier

- Sometime need to reverse direction of y

- Some formats have origin at bottom left, some at top left

- Needs additional translation

- Positioning on the screen

- Translation $T_{raster}(xpos, ypos)$

- May be different depending on raster coordinate system

- Origin at upper left or lower left

Orthographic Projection

- **Step 2a: Translation (orthographic)**
 - Bring near clipping plane into the origin
- **Step 2b: Scaling to regular box $[-1 .. 1]^2 \times [0 .. -1]$**
- **Mapping of X and Y**

$$- P_o = S_{xyz}T_{near} = \begin{pmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & near \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Camera Transformation

- **Complete transformation (combination of matrices)**

- Perspective Projection

- $T_{camera} = T_{raster} S_{raster} P_{parallel} P_{persp} S_{far} S_{xy} H R T$

- Orthographic Projection

- $T_{camera} = T_{raster} S_{raster} P_{parallel} S_{xyz} T_{near} H R T$

- **Other representations**

- Other literature uses different conventions

- Different camera parameters as input

- Different canonical viewing frustum

- Different normalized coordinates

- $[-1 .. 1]^3$ versus $[0 .. 1]^3$ versus ...

- ...

- *Results in different transformation matrices – so be careful !!!*

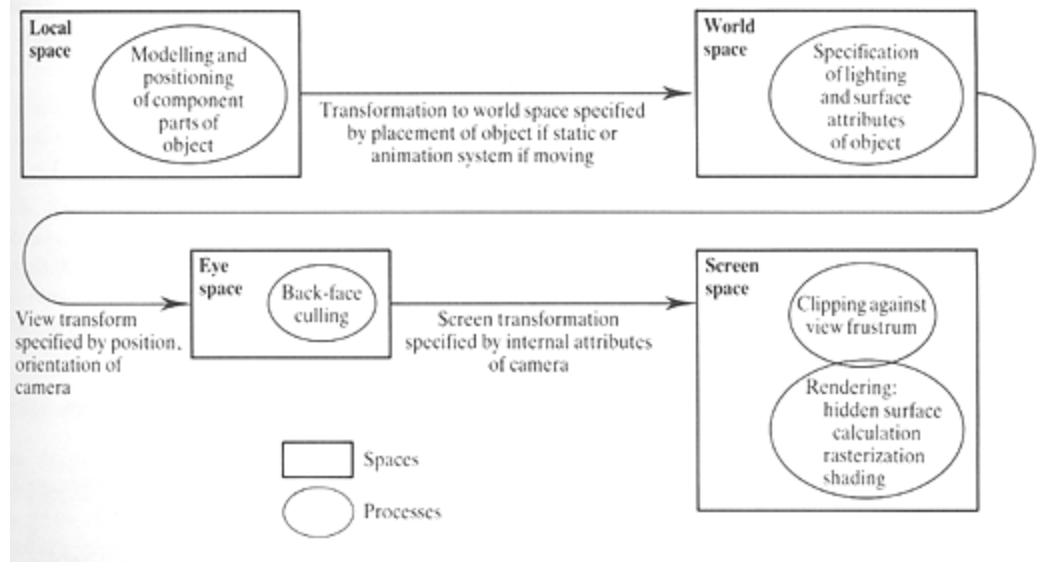
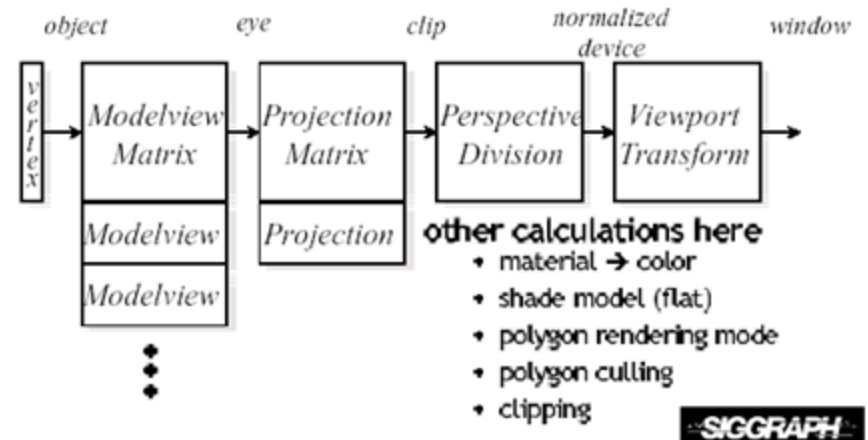
Per-Vertex Transformations

- **Traditional OpenGL pipeline**

- Hierarchical modeling
 - Modelview matrix stack
 - Projection matrix stack
- Each stack can be independently pushed/popped
- Matrices can be applied/multiplied to top stack element

- **Today**

- Arbitrary matrices as attributes to vertex shaders that apply them as they wish (later)
- All matrix stack handling must now be done by application



OpenGL

- **Modern OpenGL**
 - Transformation provided by app, applied by vertex shader
 - Vertex or Geometry shader must output clip space vertices
 - Clip space: Just before perspective divide (by w)
- **Viewport transformation**
 - `glViewport(x, y, width, height)`
 - Now can even have multiple viewports
 - `glViewportIndexed(idx, x, y, width, height)`
 - Controlling the depth range (after Perspective transformation)
 - `glDepthRangeIndexed(idx, near, far)`