Computer Graphics

- Camera Transformation -

Stefan Lemme

Overview

- **Last time**
  - Affine space $(\boldsymbol{A}, \boldsymbol{V}, \oplus)$
  - Projective space $\boldsymbol{P^n}(\mathbb{R})$
    - set of lines through origin
    - $[x, y, z, w] = [\lambda x, \lambda y, \lambda z, \lambda w] = \left[\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right]$
  - Normalized homogeneous coordinates
    - Points $(x, y, z, 1)$
    - Vectors $(x, y, z, 0)$
  - Affine transformations

$$\begin{bmatrix} a_{xx} & a_{xy} & a_{xz} & b_x \\ a_{yx} & a_{yy} & a_{yz} & b_y \\ a_{zx} & a_{zy} & a_{zz} & b_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Basic transformations
    - Translation, Scaling, Reflection, Shearing, Rotation
  - Transforming normals
    - $N = (M^{-1})^T$

Overview

- **Today**
  - How to use affine transformations
    - Coordinate spaces
    - Hierarchical structures
  - Camera transformations
    - Camera specification
    - Perspective transformation

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
    - Rigid objects: constant translation, rotation per object, (scaling)
    - Animated objects: time-varying transformation in world-space
  - Illumination can be computed in this space

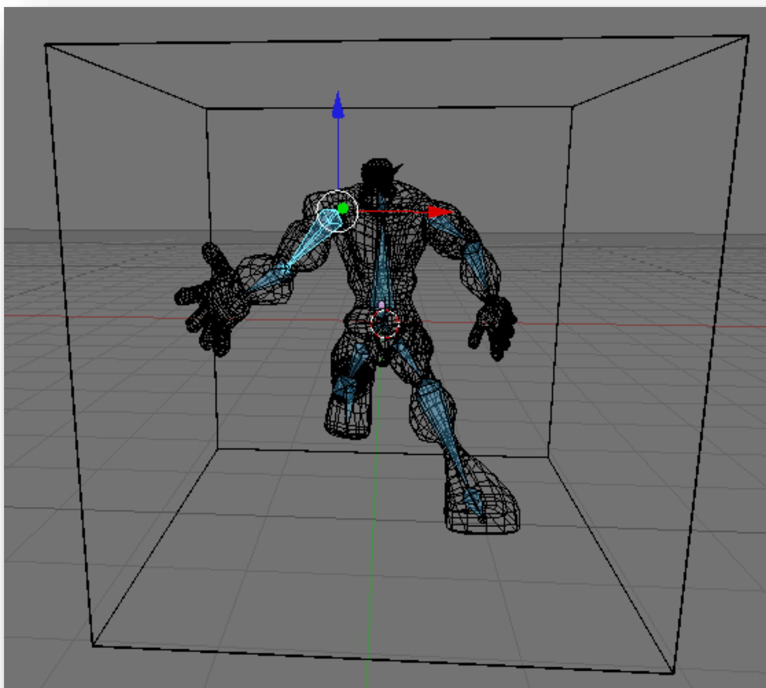
Hierarchical Coordinate Systems

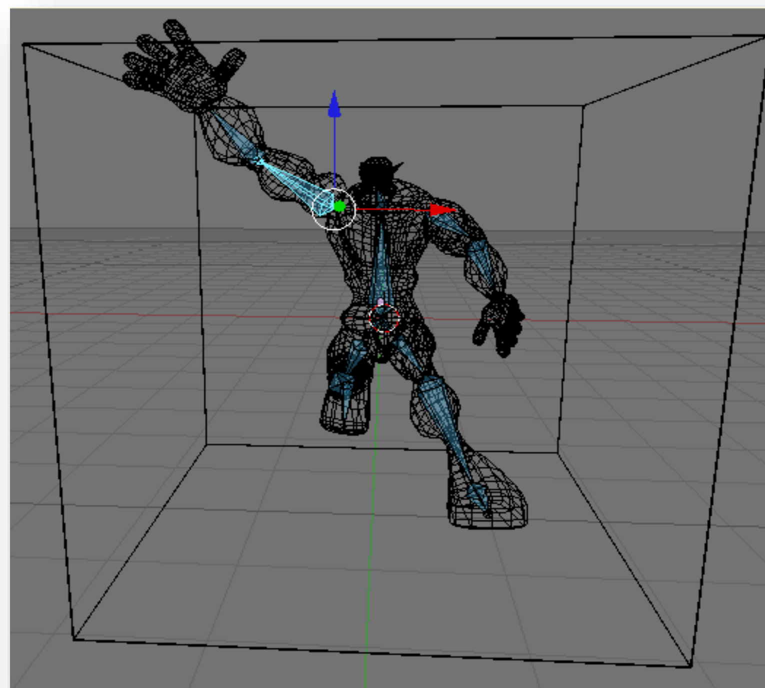
- **Hierarchy of transformations**

```
T_root                        //Position of the character in world
  T_ShoulderR                 //Right shoulder position
    T_ShoulderRJoint          //Shoulder rotation  <== User
      T_UpperArmR             //Elbow position
        T_ElbowRJoint         //Elbow rotation     <== User
          T_LowerArmR         //Wrist position
            T_WristRJoint     //Wrist rotation     <== User
              ...             //Hand and fingers...
  T_ShoulderL                 //Left shoulder position
    T_ShoulderLJoint          //Shoulder rotation  <== User
      T_UpperArmL             //Elbow position
        T_ElbowLJoint         //Elbow rotation     <== User
          T_LowerArmL         //Wrist position
            ...
```

Hierarchical Coordinate Systems

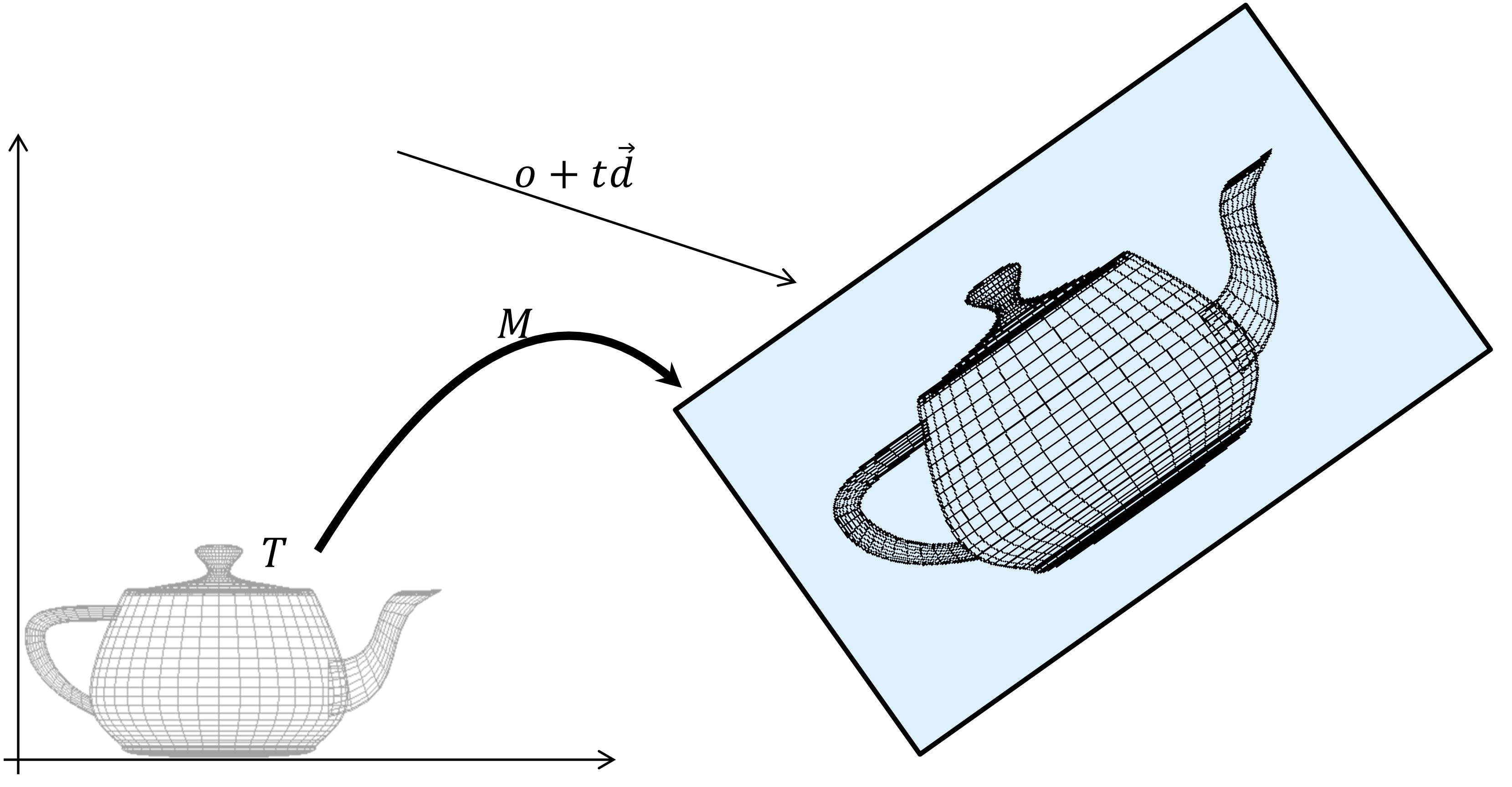
- **Used in Scene Graphs**
  - Group objects hierarchically
  - Local coordinate system is relative to parent coordinate system
  - Apply transformation to the parent to change the whole sub-tree (or sub-graph)





Ray-tracing Transformed Objects

- Ray (world coordinates)
- T – set of triangles (local coordinates)
- M – transformation matrix (local-to-world)

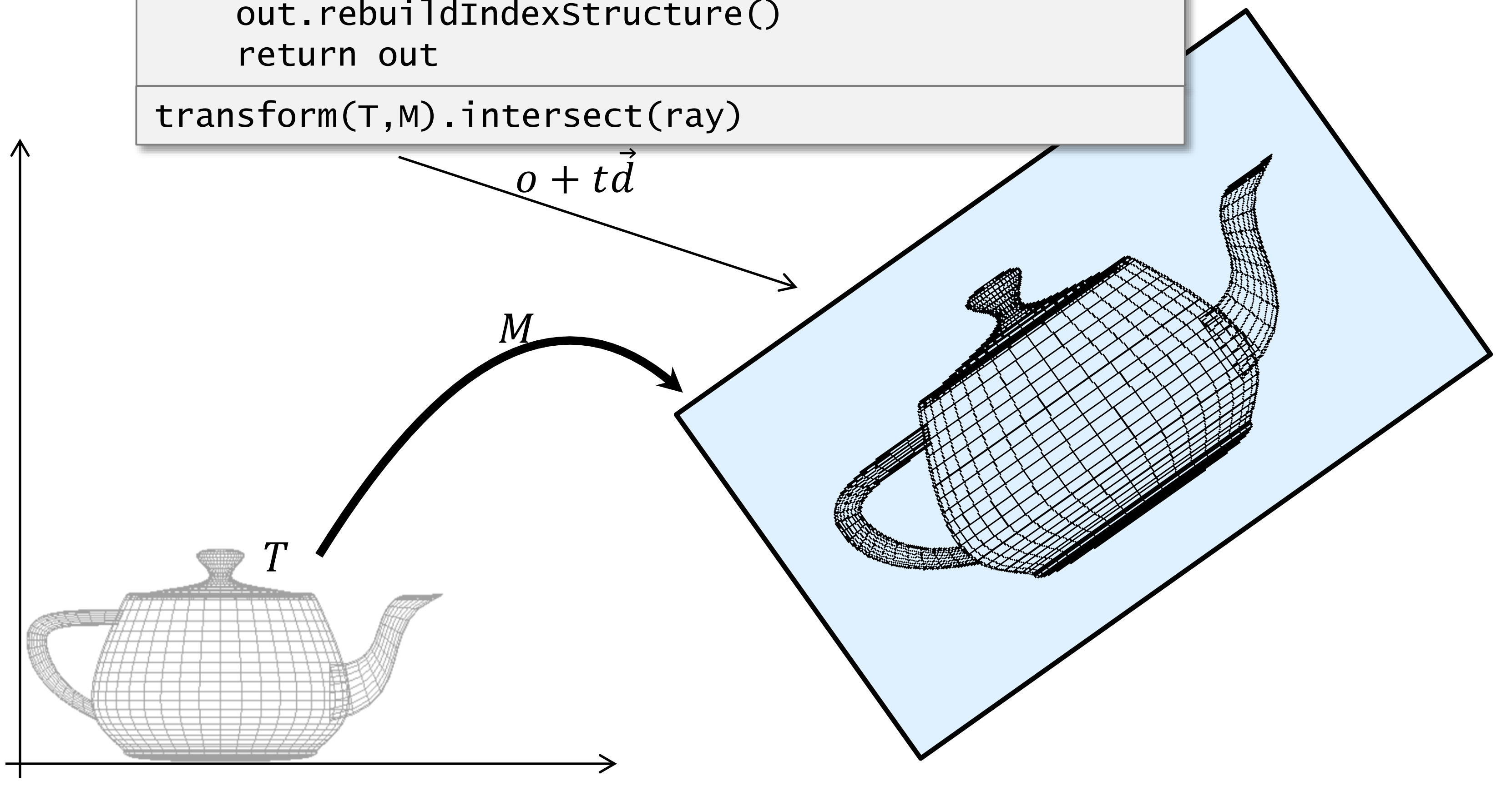


Ray-tracing Transformed Objects

- **Option 1: transform the triangles**

```
def transform(T,M)
    out = {}
    foreach p in T
        q = M*p
        out.insert(q)
    out.rebuildIndexStructure()
    return out
```

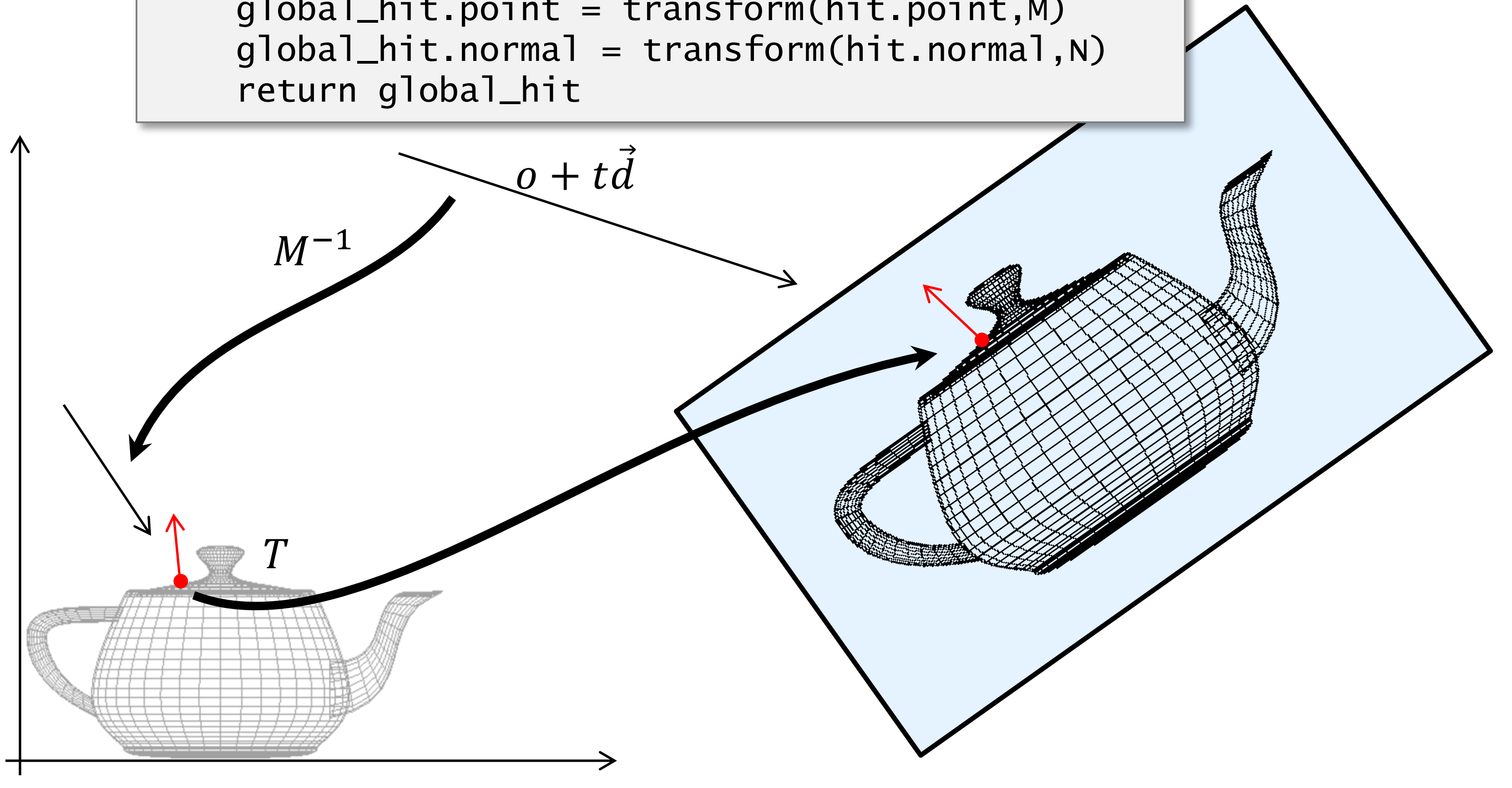
```
transform(T,M).intersect(ray)
```



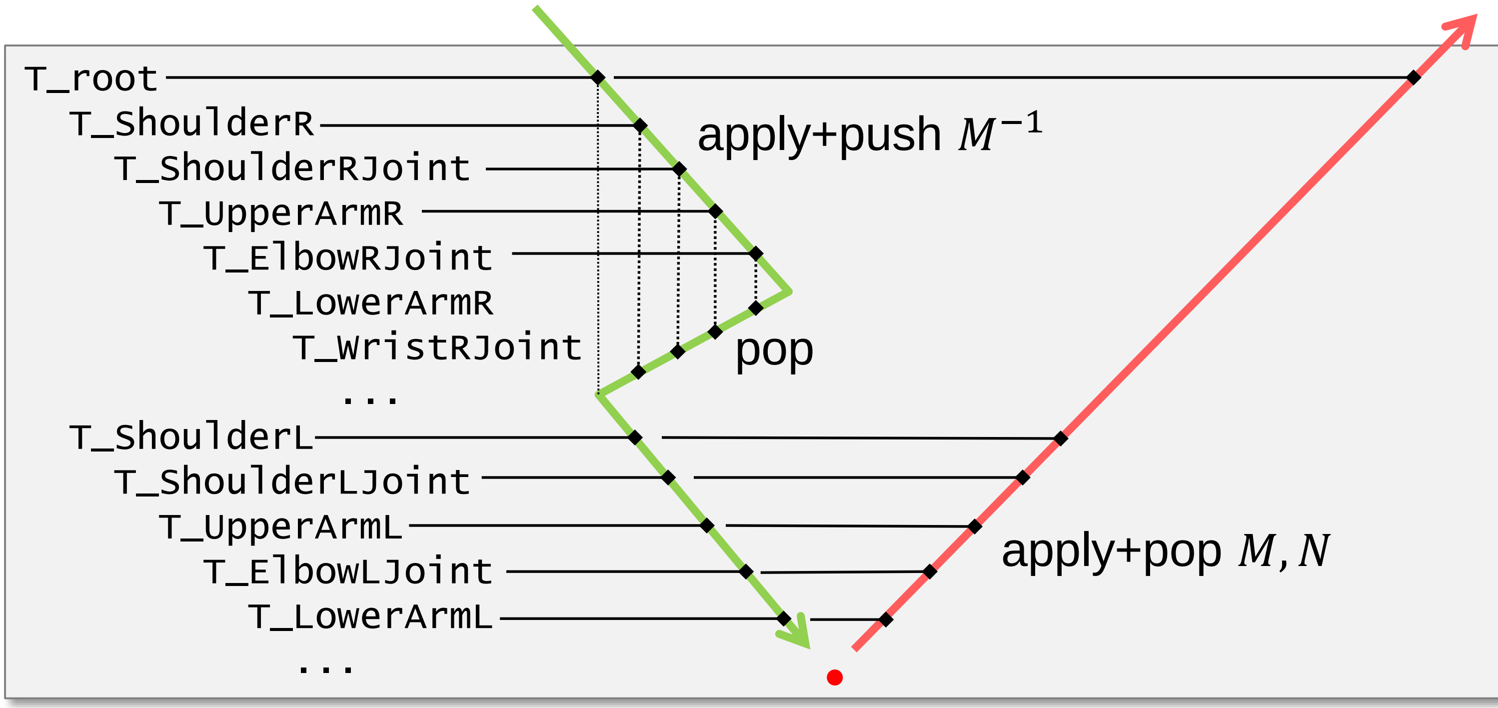
Ray-tracing Transformed Objects

- **Option 2: transform the ray**

```
def intersect(obj,ray)
    Minv = obj.M.inverse()
    N = obj.M.normalTransform()
    local_ray = transform(ray,Minv)
    hit = obj.intersect(local_ray)
    global_hit.point = transform(hit.point,M)
    global_hit.normal = transform(hit.normal,N)
    return global_hit
```

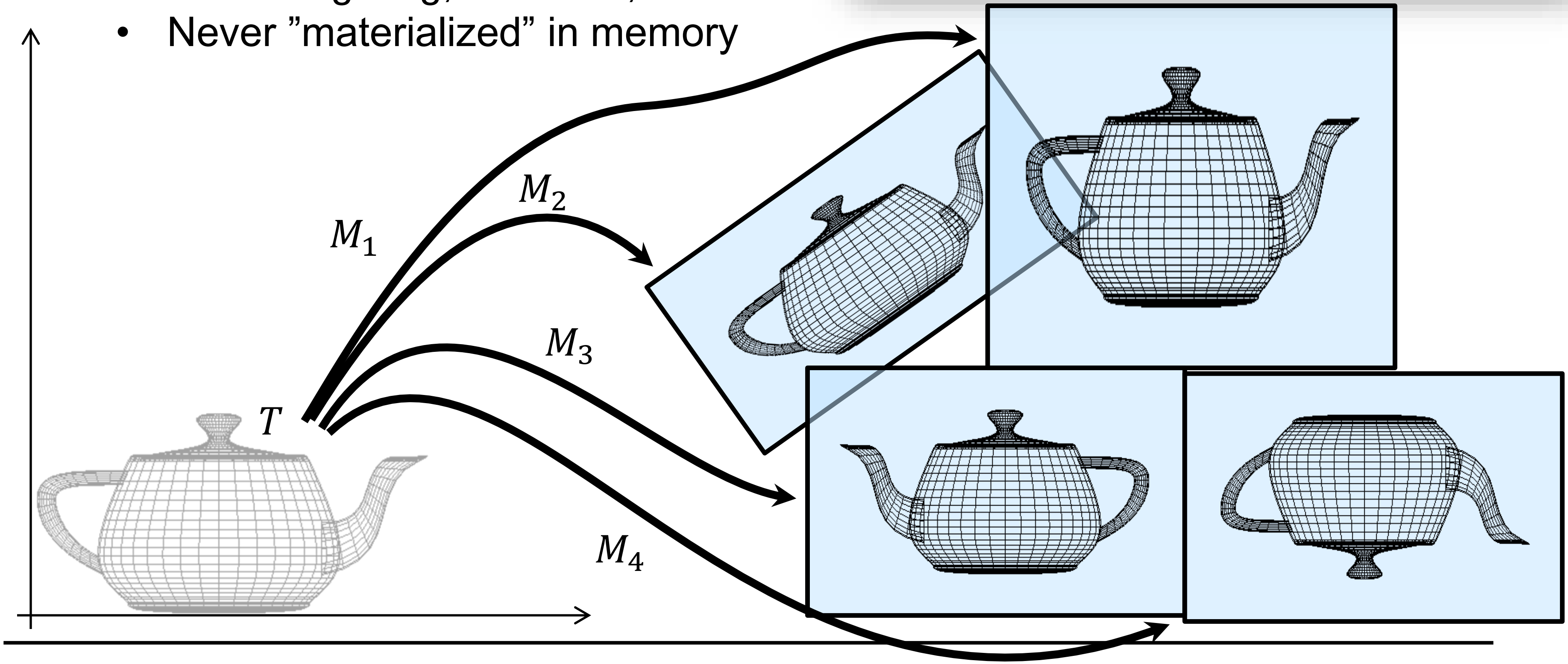


Ray-tracing through a Hierarchy



Instancing

- T – set of triangles
  - local coordinates
  - memory
- M_i – transformation matrices
  - local-to-world
- Multiple rendered objects
  - Correct lighting, shadows, etc...
  - Never ”materialized” in memory



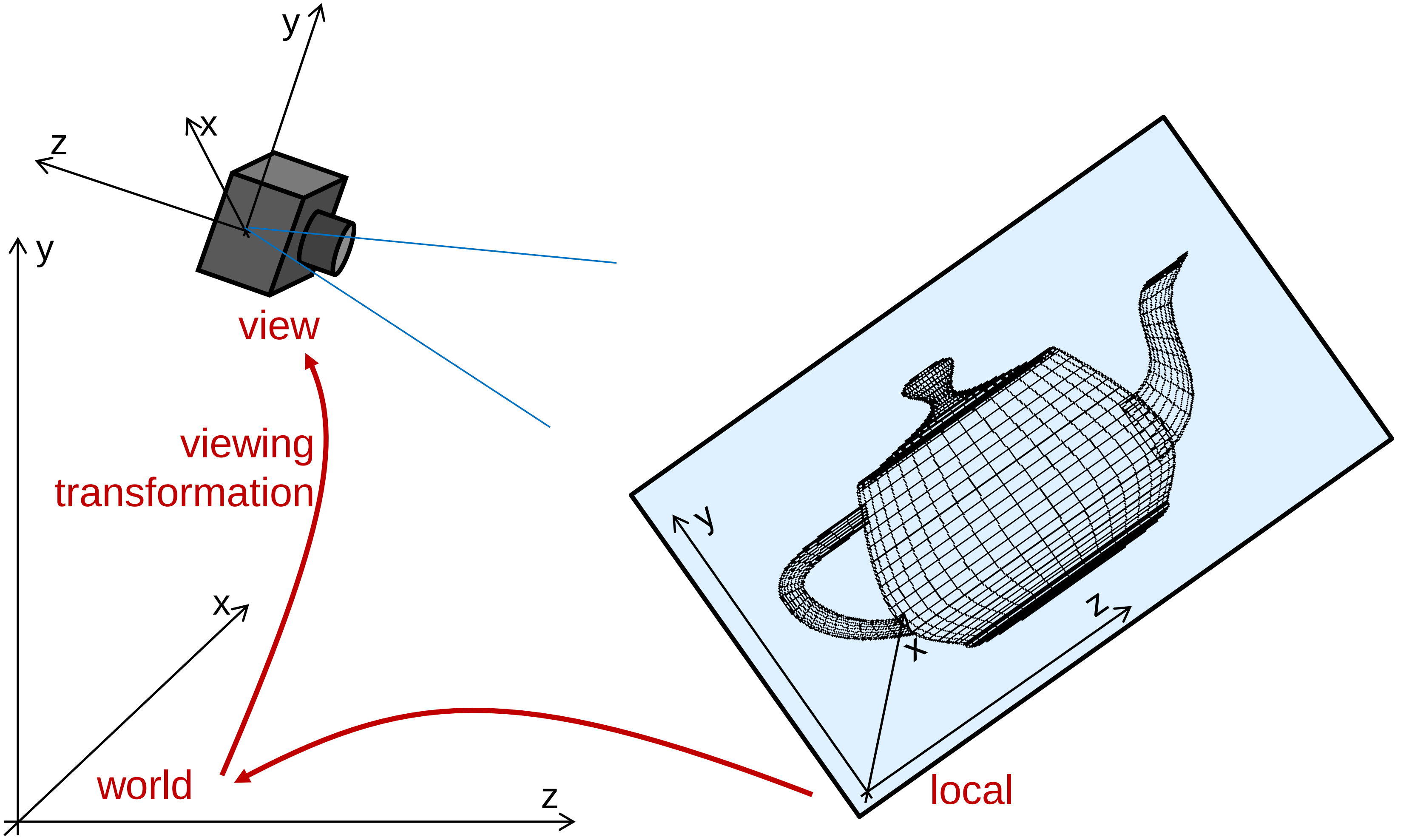
Coordinate Systems

- **Local (object) coordinate system (3D)**
- **World (global) coordinate system (3D)**
- **Camera/view/eye coordinate system (3D)**
  - Coordinates relative to camera position & direction
    - Camera itself specified relative to world space
  - Illumination can also be done in that space
- **Normalized device coordinate system (2.5D)**
  - After perspective transformation, rectilinear, in $[0,1]^3$
  - Normalization to view frustum, 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

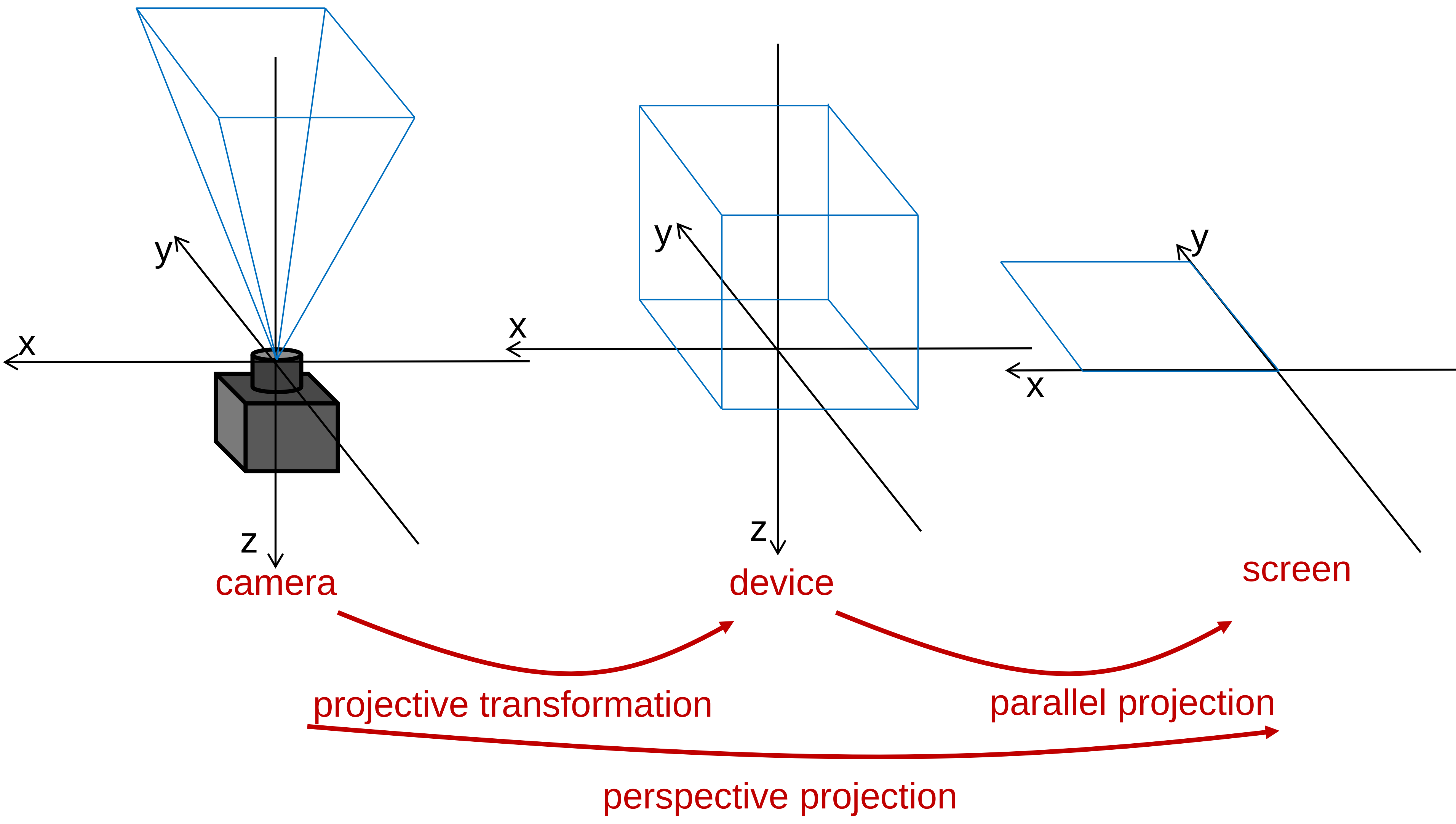
Goal: Transform objects from local to screen

  - typical for rasterization

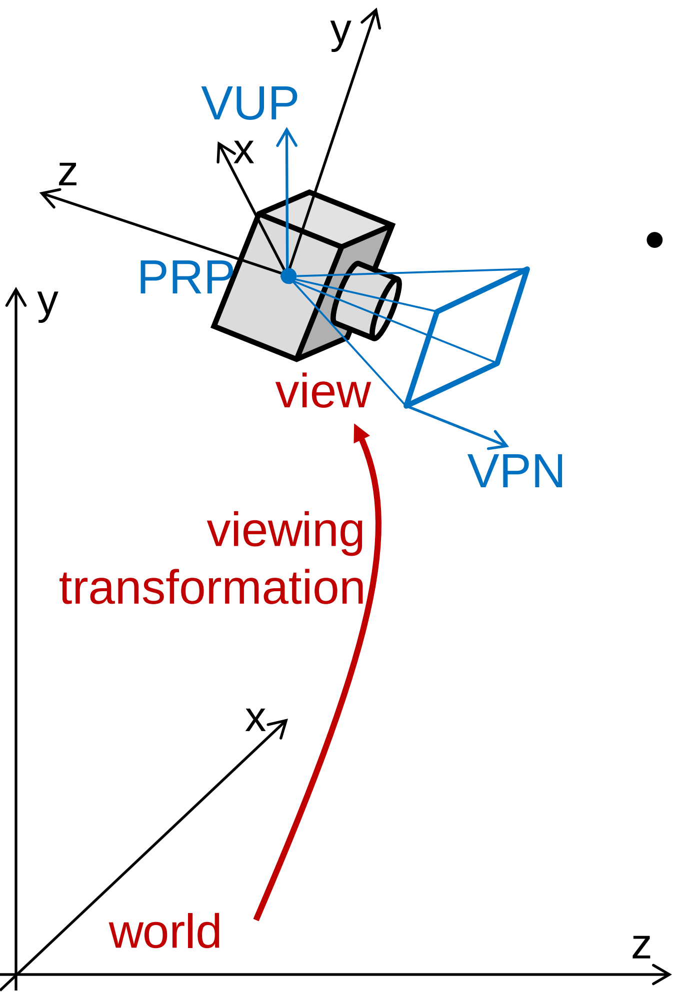
Coordinate Systems



Coordinate Systems

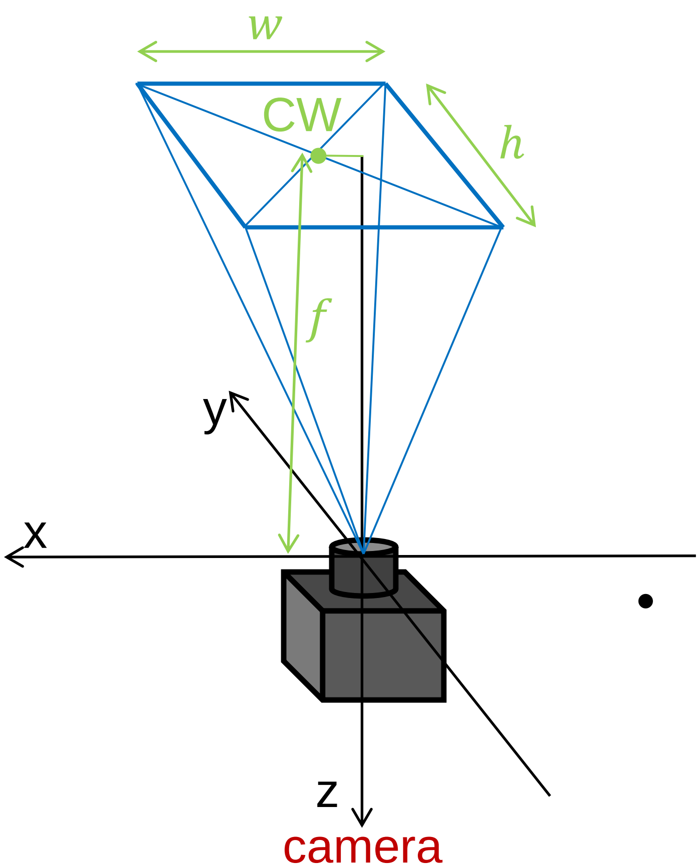


Viewing Transformation



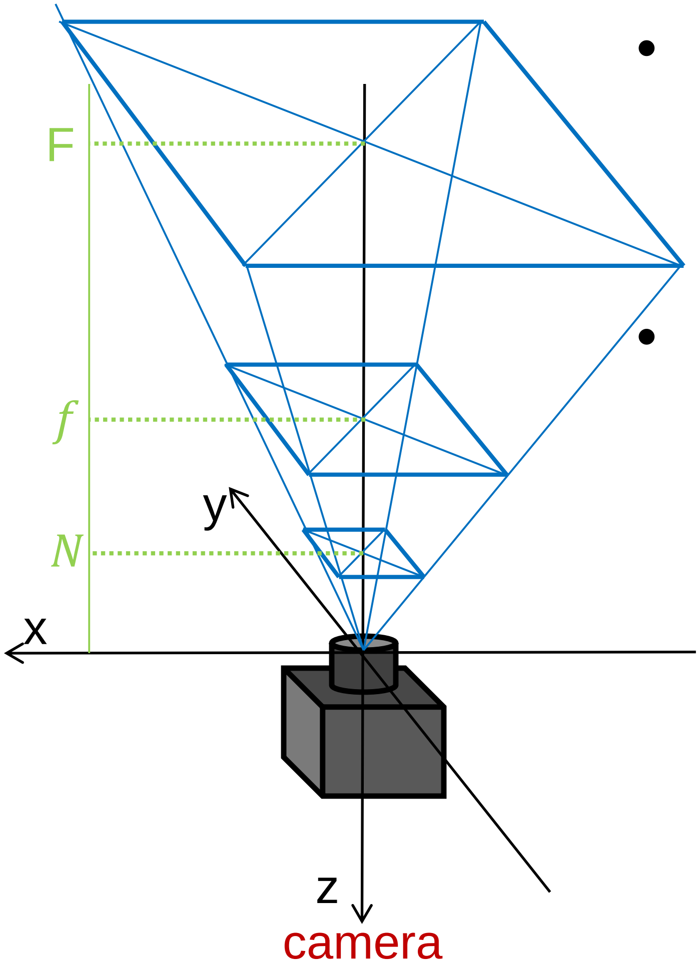
- **External (extrinsic) camera parameters**
  - Center of projection
    - projection reference point (PRP)
  - Optical axis: view-plane normal (VPN)
  - View up vector (VUP)
- **Needed Transformations**
  - Translation $T(-PRP)$
  - Rotation $R(\vec{u}, \phi)$:
    - $VPN \parallel -\vec{z}$
    - $VUP \in \mathrm{Span}(\vec{y}, \vec{z})$

Viewing Transformation



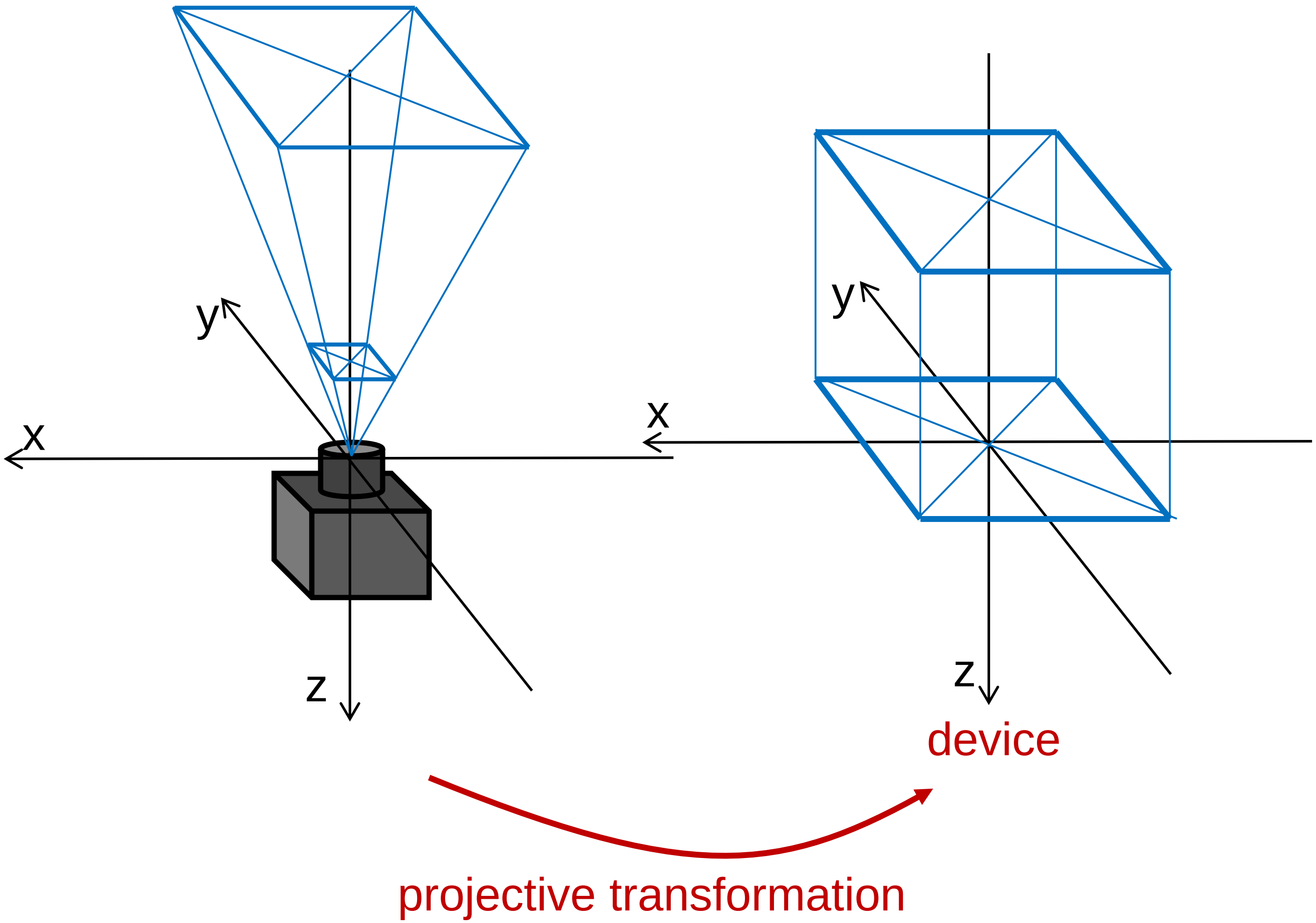
- **Internal (intrinsic) camera parameters**
  - Screen window
    - center of the window (CW)
    - width, height
  - Focal length f
    - projection plane distance along $-\vec{z}$
  - FOV
    - Instead of f
    - CW in the center
    - vertical/horizontal
    - aspect ratio
- **Needed Transformations**
  - Shear to move CW to center
  - $$\mathrm{H}_{xy}\left(-\frac{CW_x}{f}, -\frac{CW_y}{f}\right) = \begin{bmatrix} 1 & 0 & -\frac{CW_x}{f} & 0 \\ 0 & 1 & -\frac{CW_y}{f} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing Transformation

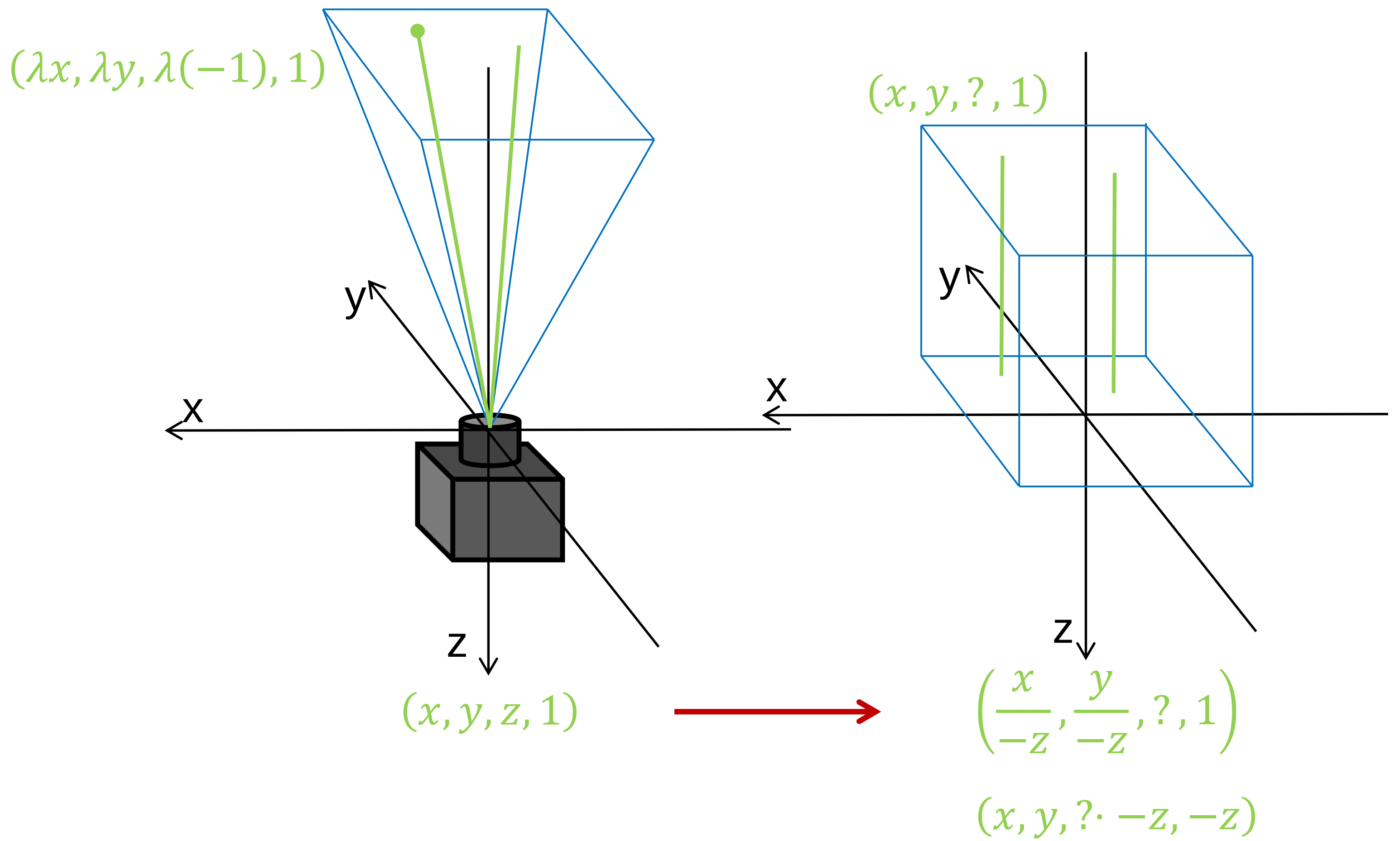


- **Internal (intrinsic) camera parameters**
  - Near/Far planes
    - N, F
    - Render only objects between Near and Far
- **Normalization Transformations**
  - Frustrum boundaries open at $45°$
    - $S\left(\frac{2f}{w}, \frac{2f}{h}, 1\right) = \begin{bmatrix} \frac{2f}{w} & 0 & 0 & 0 \\ 0 & \frac{2f}{h} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
  - Far plane at $z = -1$
    - $S\left(\frac{1}{F}, \frac{1}{F}, \frac{1}{F}\right) = \begin{bmatrix} \frac{1}{F} & 0 & 0 & 0 \\ 0 & \frac{1}{F} & 0 & 0 \\ 0 & 0 & \frac{1}{F} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Projective Transformation

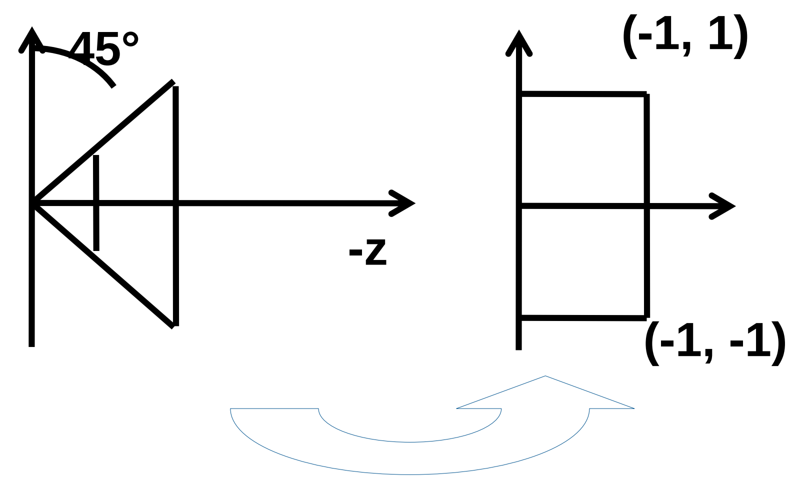


Perspective Transformation



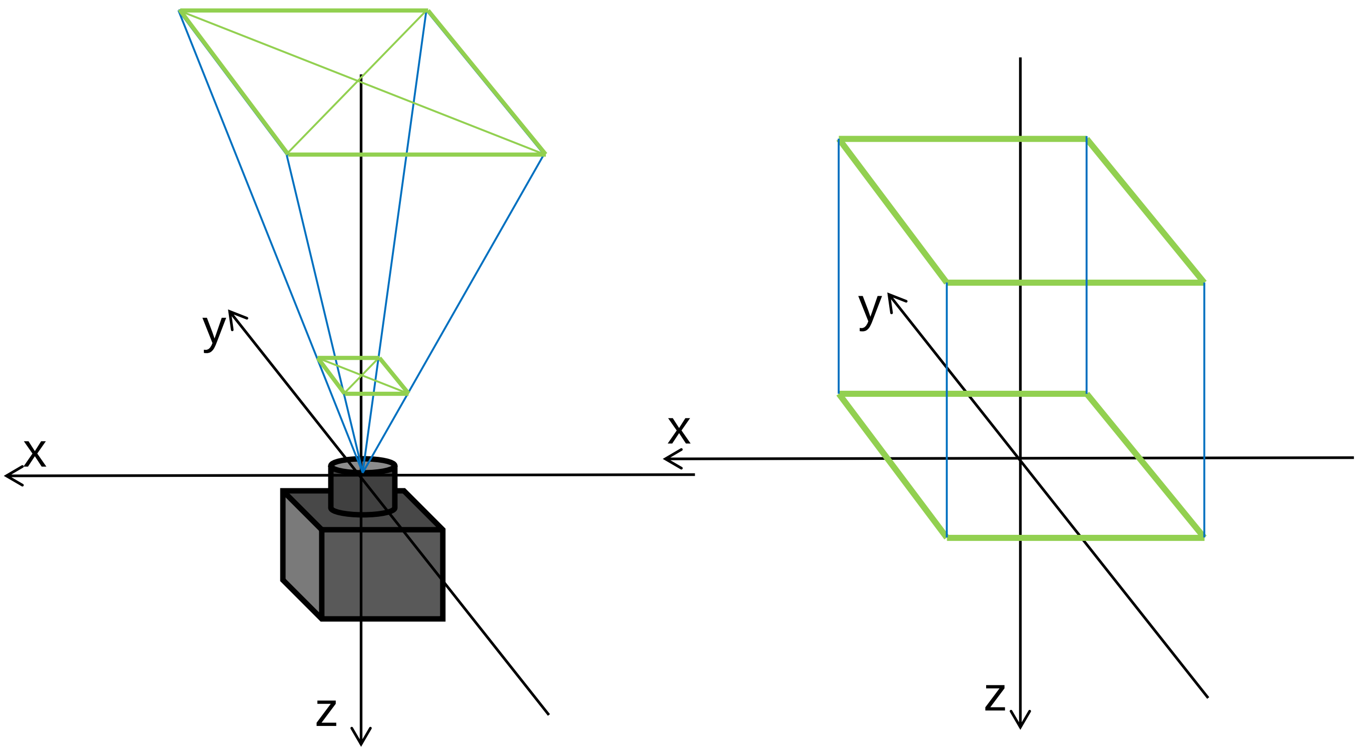
Perspective Transformation

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



  - Note: Perspective projection = perspective transformation + parallel projection

Perspective Transformation

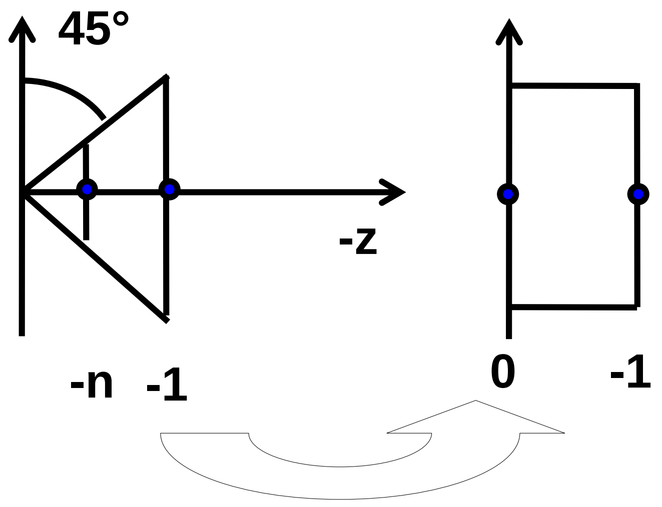


Far: -1 $(?,?,-1,1)$ → $(?,?,-1,1)$

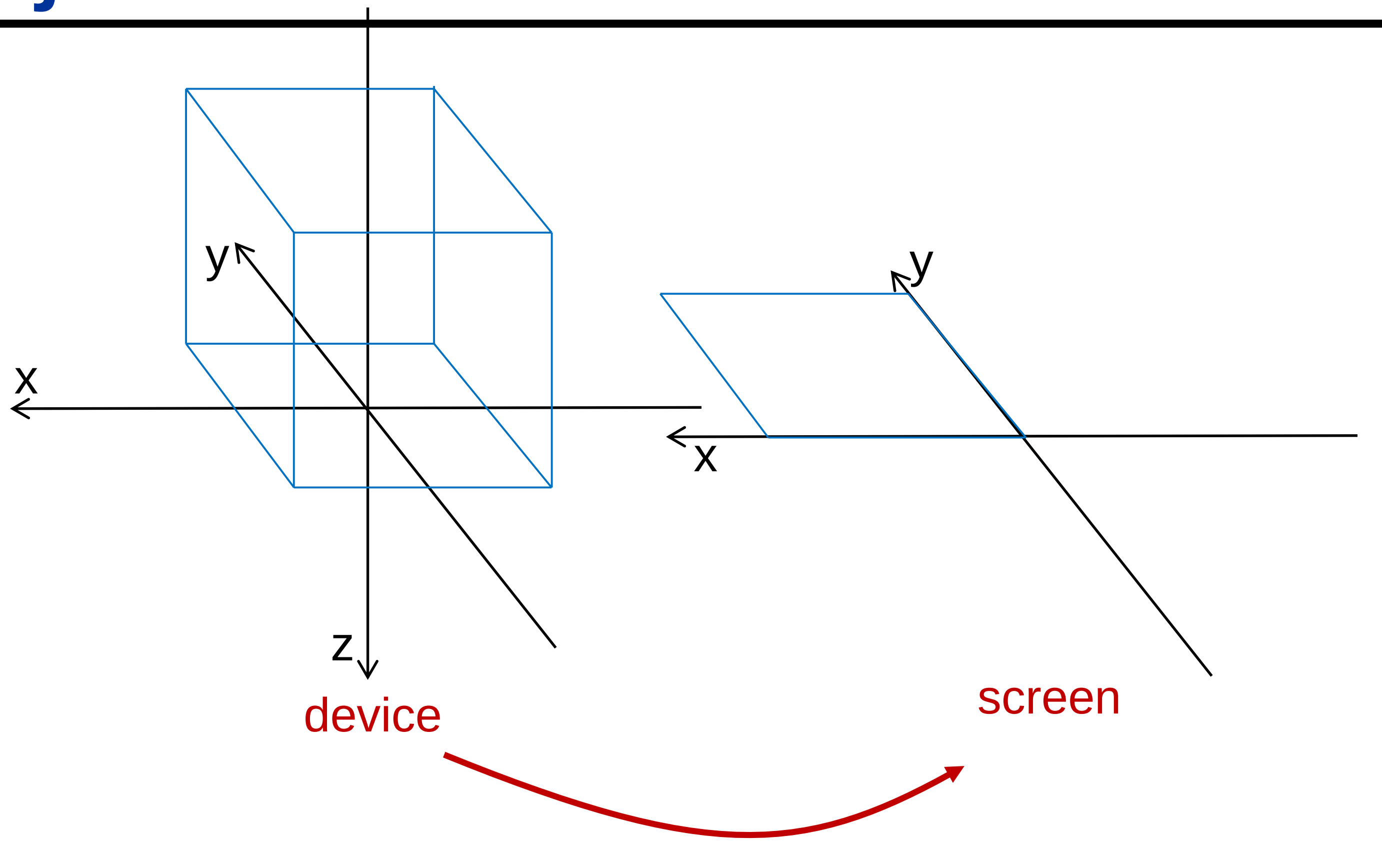
Near: $-n = -\frac{N}{F}$ $(?,?,-n,1)$ → $(0,0,0,1)$

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 = near / far (due to previous scaling by 1/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**
  - $P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1-n} & \frac{n}{1-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$
  - Transform Z non-linearly (in 3D)
    - $z' = -\frac{z+n}{z(1-n)}$



Projection to Screen



$$P_{parallel} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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 x [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
    - $T_{camera} = T_{raster}\, S_{raster}\, P_{parallel}\, P_{persp}\, S_{far}\, S_{xy}\, H_{xy}\, R\, T$
  - Orthographic Projection
    - $T_{camera} = T_{raster}\, S_{raster}\, P_{parallel}\, S_{xyz}\, T_{near} H_{xy} R\, T$
- **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 !!!

Perspective vs. Orthographic

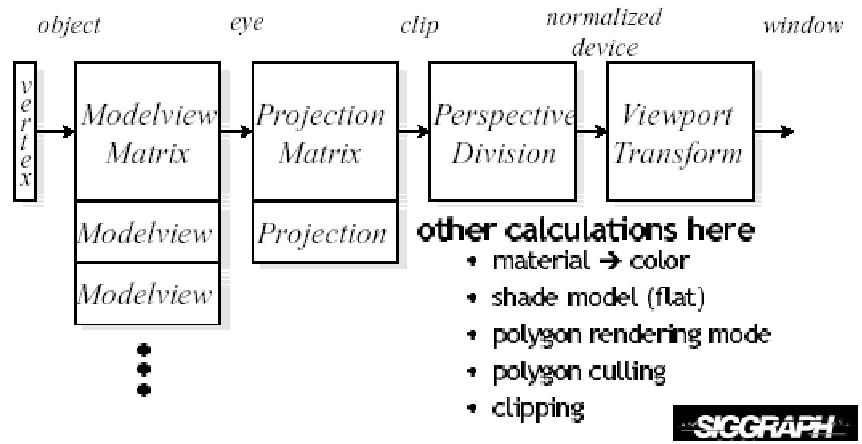
- **Parallel lines remain parallel**
- **Useful for modeling => feature alignment**

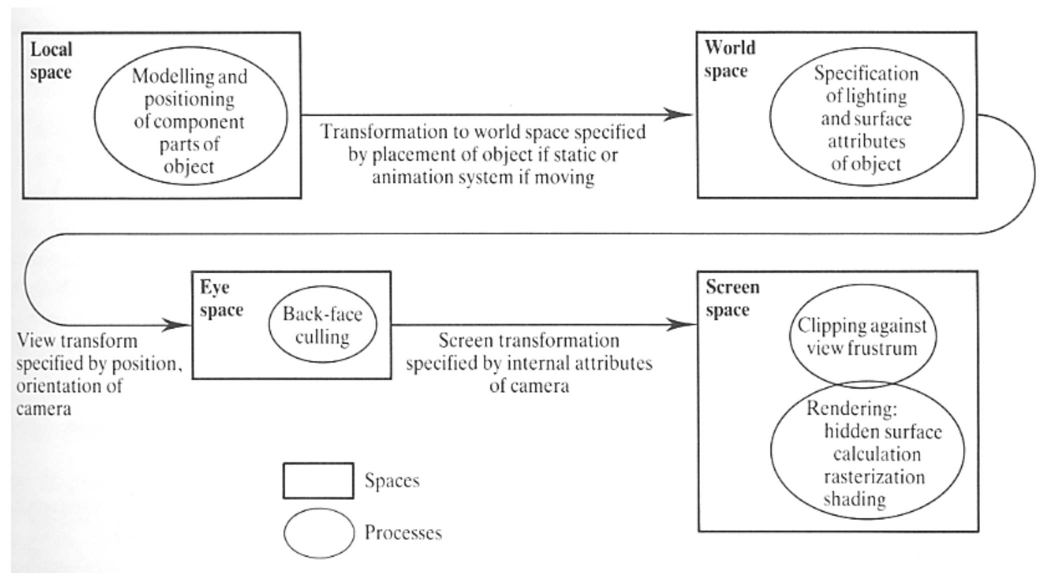
Coordinate Systems

- **Normalized (projection) coordinates**
  - 3D: normalized $[-1 .. 1]^3$ or $[-1 .. 1]^2$ x $[0 .. -1]$
  - Clipping
  - Parallel projection
- **Normalized 2D device coordinates $[-1 .. 1]^2$**
  - Translation and scaling
- **Normalized 2D device coordinates $[0 .. 1]^2$**
  - Where is the origin?
    - RenderMan, X11: upper left
    - OpenGL: lower left
  - Viewport transformation
    - Adjustment of aspect ratio
    - Position in raster coordinates
- **Raster coordinates**
  - 2D: units in pixels [0 .. xres-1, 0 .. yres-1]

OpenGL

- **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

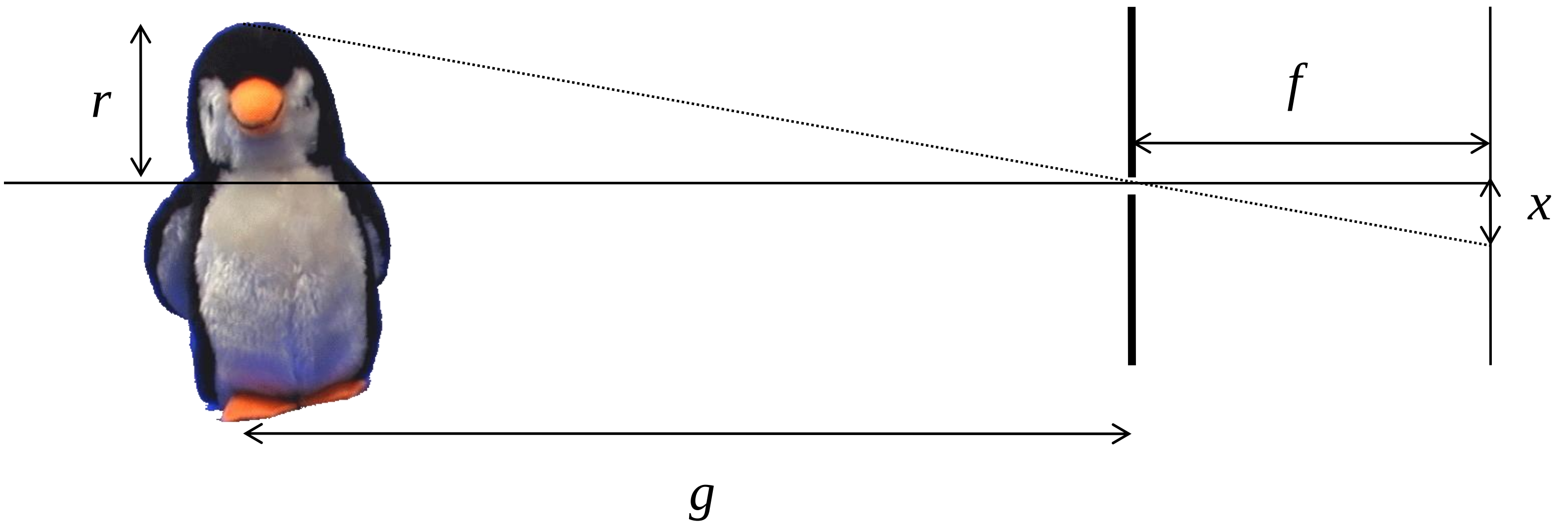




OpenGL

- **Traditional ModelView matrix**
  - Modeling transformations AND viewing transformation
  - No explicit world coordinates
- **Traditional Perspective transformation**
  - Simple specification
    - glFrustum(left, right, bottom, top, near, far)
    - glOrtho(left, right, bottom, top, near, far)
- **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)

Pinhole Camera Model



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

Infinitesimally small pinhole

$\Rightarrow$ Theoretical (non-physical) model

$\Rightarrow$ Sharp image everywhere

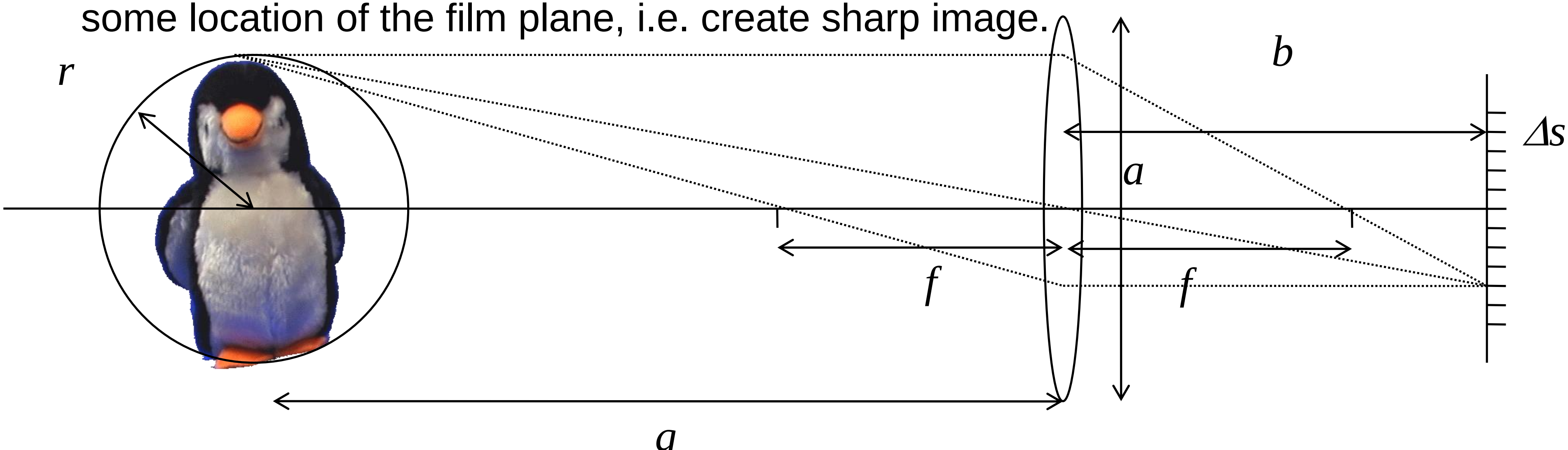
$\Rightarrow$ Infinite depth of field

$\Rightarrow$ Infinitely dark image in reality

$\Rightarrow$ 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)

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

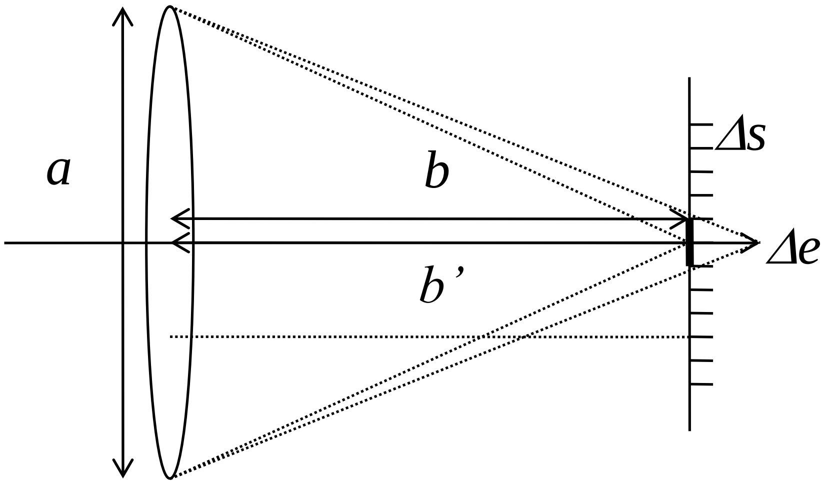
Object center at distance *g* is in focus at

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

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 Δ*s*

Depth of field (DOF)

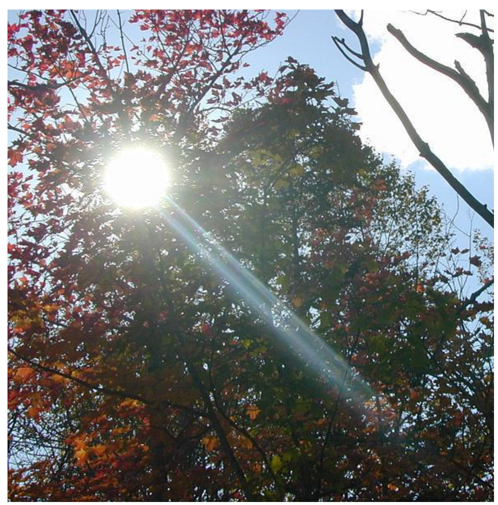
$$r < \frac{g \Delta s (g - f)}{af + \Delta s (g - f)} \Rightarrow r \sim \frac{1}{a}$$

The smaller the aperture, the larger the depth of field

Ignored Effects

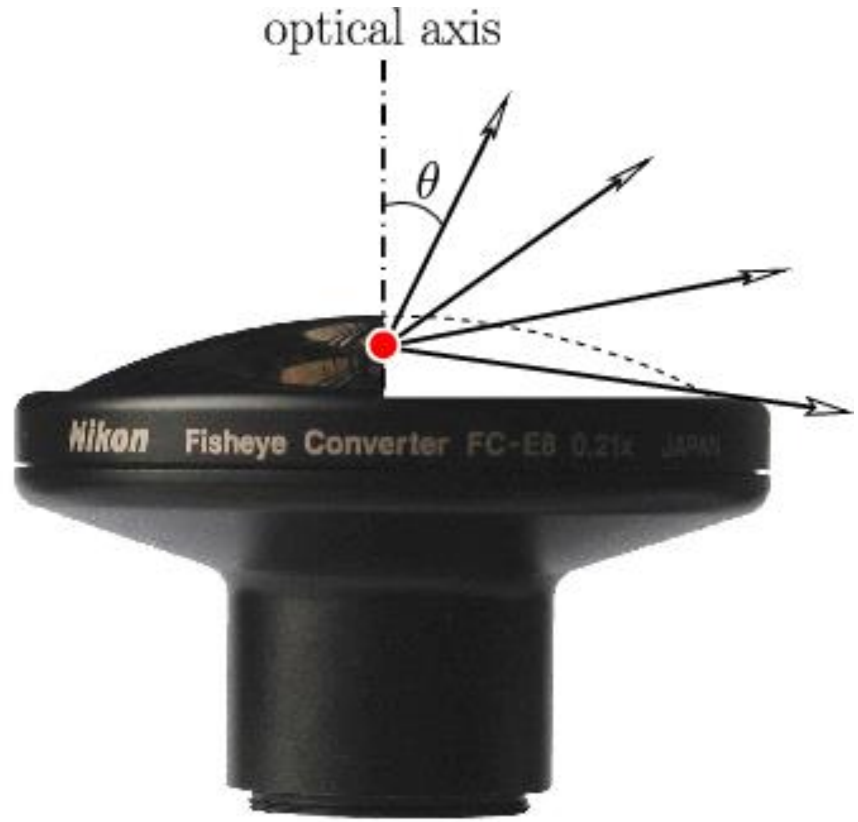
A lot of things that we ignored with our pinhole camera model

- Depth-of-field
- Lens distortion
- Aberrations
- Vignetting
- Flare
- …



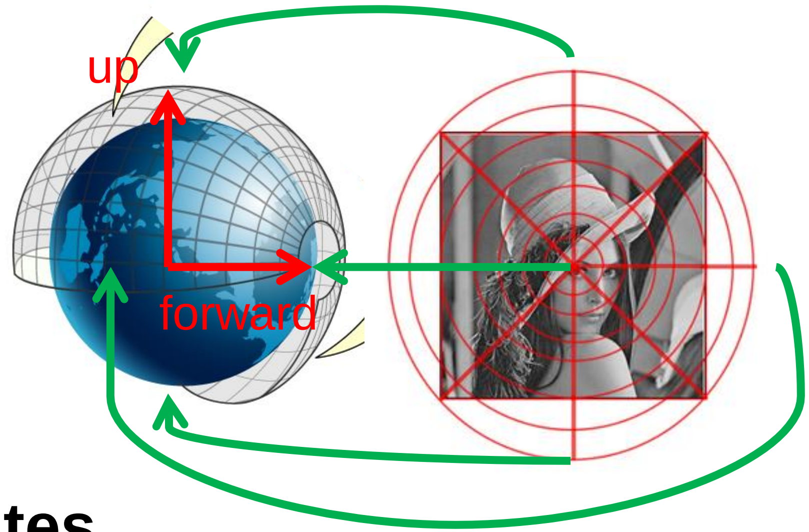
Fish-Eye Camera

- **Physical limitations of mapping function**



Fish-Eye Camera

- **Go beyond physical limitations**
- **Use polar parameterization**
  - r = sqrt(sscx^2 + sscy^2)
  - φ = atan2(sscy, sscx)
- **Wrap onto a sphere**
  - Equi-angular mapping
  - θ = r * fov / 2 (inclination angle)
  - φ = φ
- **Convert to Cartesian coordinates**
  - x = sin θ cos φ
  - y = sin θ sin φ
  - z = cos θ



Fish-Eye Camera

- Distortion: straight lines become curved

Fish-Eye Camera

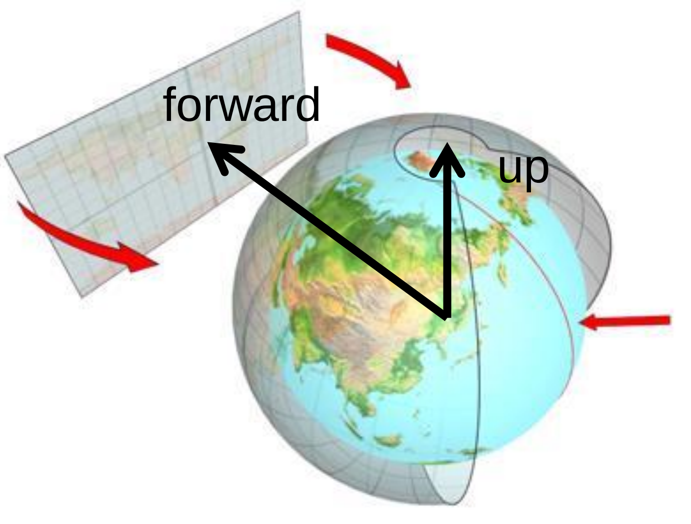
- **Capture Environment**

Fish-Eye Camera

- **Little Planet**

Environment Camera

- **Go way beyond physical limitations**
- **Use spherical parameterization**
  - Equi-angular mapping
  - θ = sscy * fovy / 2 (elevation angle)
  - φ = sscx * fovx / 2
- **Convert to Cartesian coordinates**
  - $x = \cos\theta \cos\varphi$
  - $y = \cos\theta \sin\varphi$
  - $z = \sin\theta$



Environment Camera

- Vertical straight lines remain straight
- Horizontal straight lines become curved