Homogeneous coordinates

regular 3D point to homogeneous:

$$\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \longrightarrow \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

homogeneous point to regular 3D:

$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} \longrightarrow \begin{pmatrix} p_x/p_w \\ p_y/p_w \\ p_z/p_w \end{pmatrix}$$

Translation and scaling

Similar to 2D; translation by a vector

$$t = [t_x, t_y, t_z]$$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Nonuniform scaling in three directions

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations around coord axes

angle θ, around X axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

around Y axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

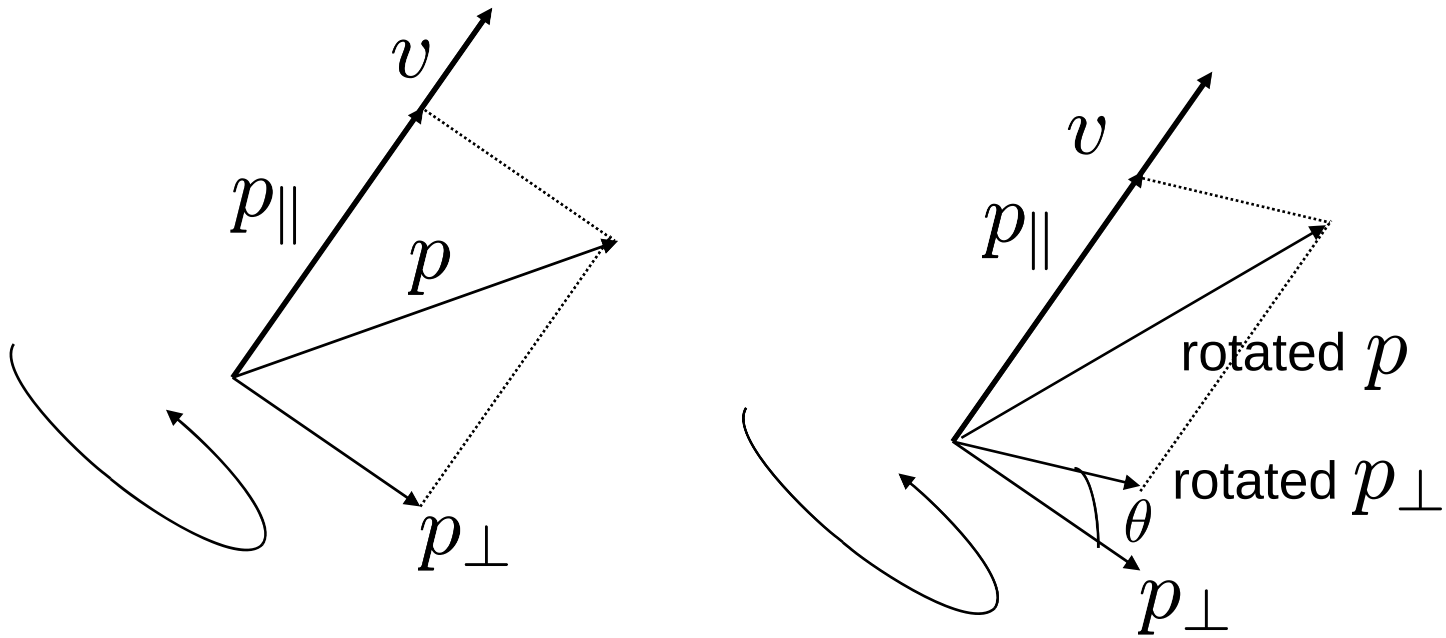
note where the minus is!

around Z axis:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General rotations

Given an axis (a unit vector) and an angle, find the matrix



Only the component perpendicular to axis changes

General rotations

(rotated vectors are denoted with $'$)

project p on v: $$p_{||} = (p, v)v$$

the rest of p is the other component: $$p_{\perp} = p - (p, v)v$$

rotate perp. component: $$p'_{\perp} = p_{\perp} \cos\theta + (v \times p_{\perp}) \sin\theta$$

add back two components: $$p' = p'_{\perp} + p_{||}$$

Combine everything, using $v \times p_{\perp} = v \times p$ to simplify:

$$\boxed{p' = \cos\theta \; p + (1 - \cos\theta)(p, v)v + \sin\theta(v \times p)}$$

General rotations

How do we write all this using matrices?

$$p' = \cos\theta\ p + (1-\cos\theta)(p,v)v + \sin\theta(v \times p)$$

$$(p,v)v = \begin{bmatrix} v_x v_x p_x + v_x v_y p_y + v_x v_z p_z \\ v_y v_x p_x + v_y v_y p_y + v_y v_z p_z \\ v_z v_x p_x + v_z v_y p_y + v_z v_z p_z \end{bmatrix} = \begin{bmatrix} v_x v_x & v_x v_y & v_x v_z \\ v_y v_x & v_y v_y & v_y v_z \\ v_z v_x & v_z v_y & v_z v_z \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

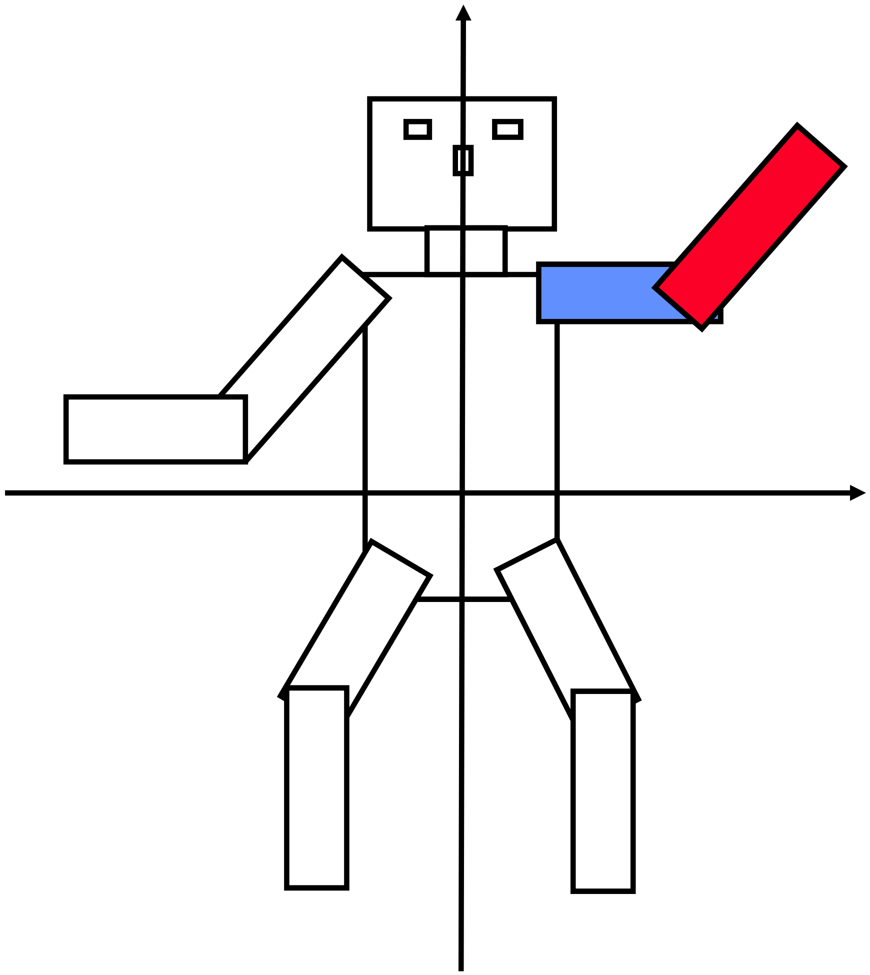
$$(v \times p) = \begin{bmatrix} -v_z p_y + v_y p_z \\ v_z p_x - v_x p_z \\ -v_y p_x + v_x p_y \end{bmatrix} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

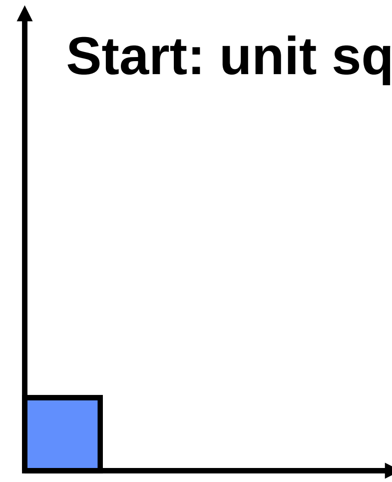
Final result, the matrix for a general rotation around a by angle θ:

$$\cos\theta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1-\cos\theta) \begin{bmatrix} v_x v_x & v_x v_y & v_x v_z \\ v_y v_x & v_y v_y & v_y v_z \\ v_z v_x & v_z v_y & v_z v_z \end{bmatrix} + \sin\theta \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

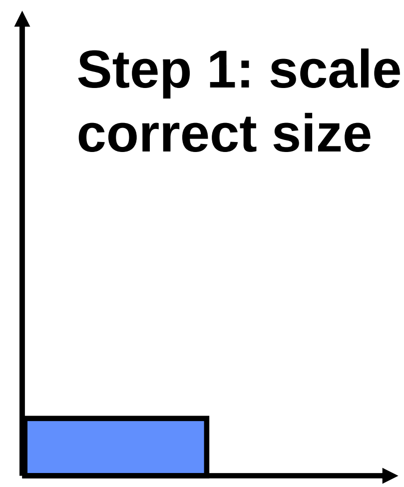
Hirerarchical transformations

Building the arm



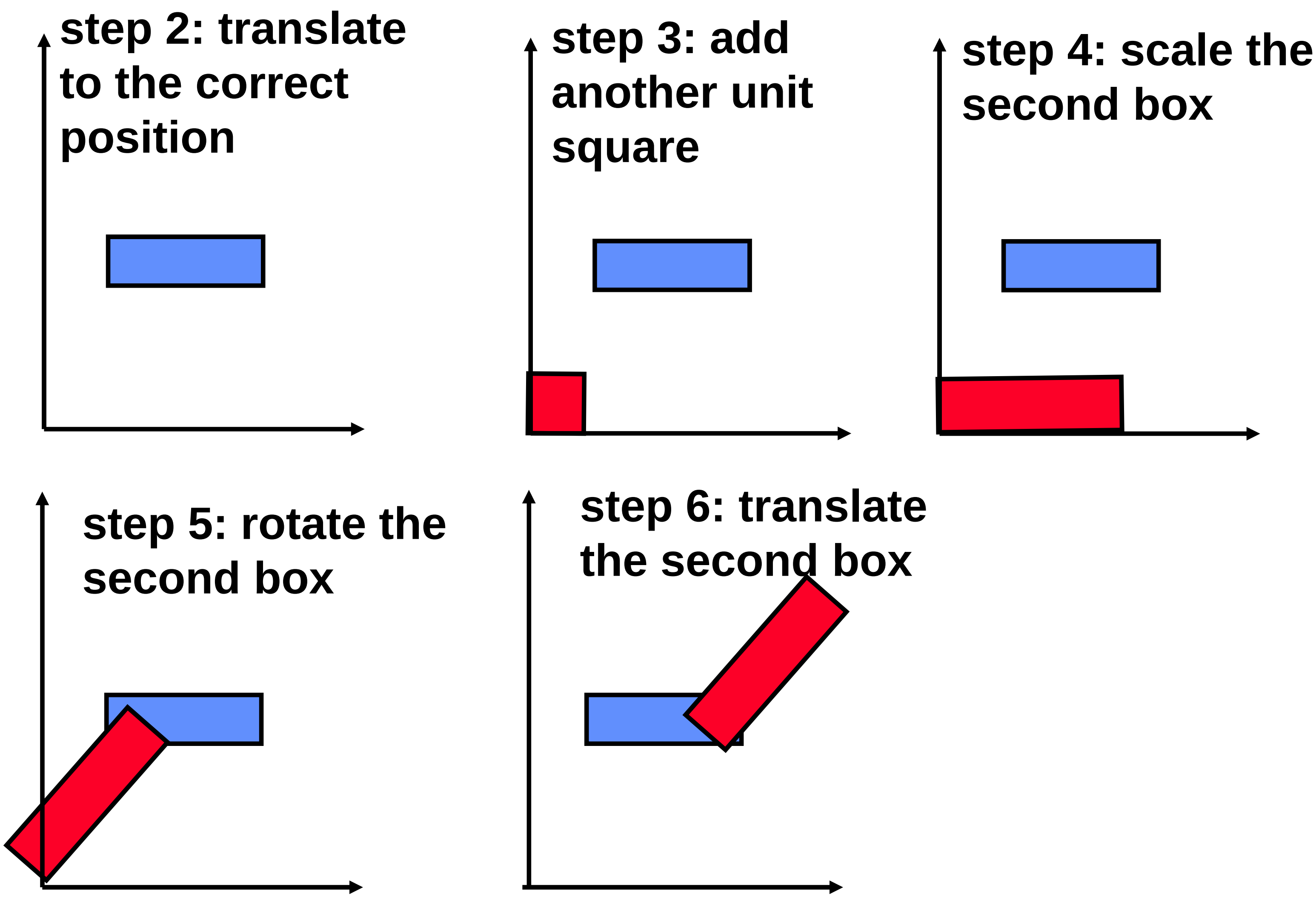


Start: unit square



Step 1: scale to the correct size

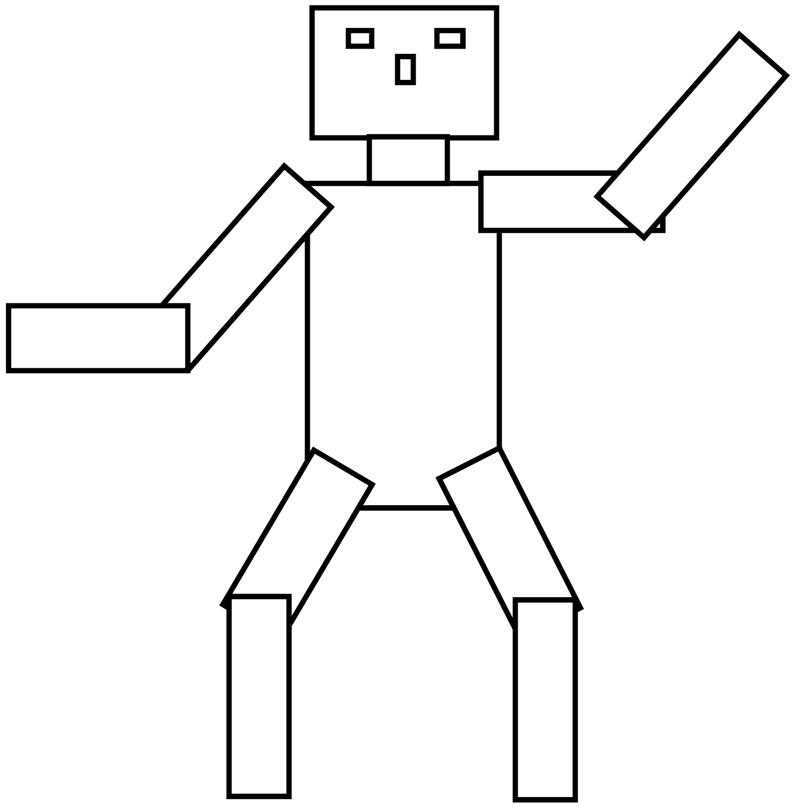
Building the arm



Hierarchical transformations

- Positioning each part of a complex object separately is difficult
- If we want to move whole complex objects consisting of many parts or complex parts of an object (for example, the arm of a robot) then we would have to modify transformations for each part
- solution: build objects hierarchically

Hierarchical transformations



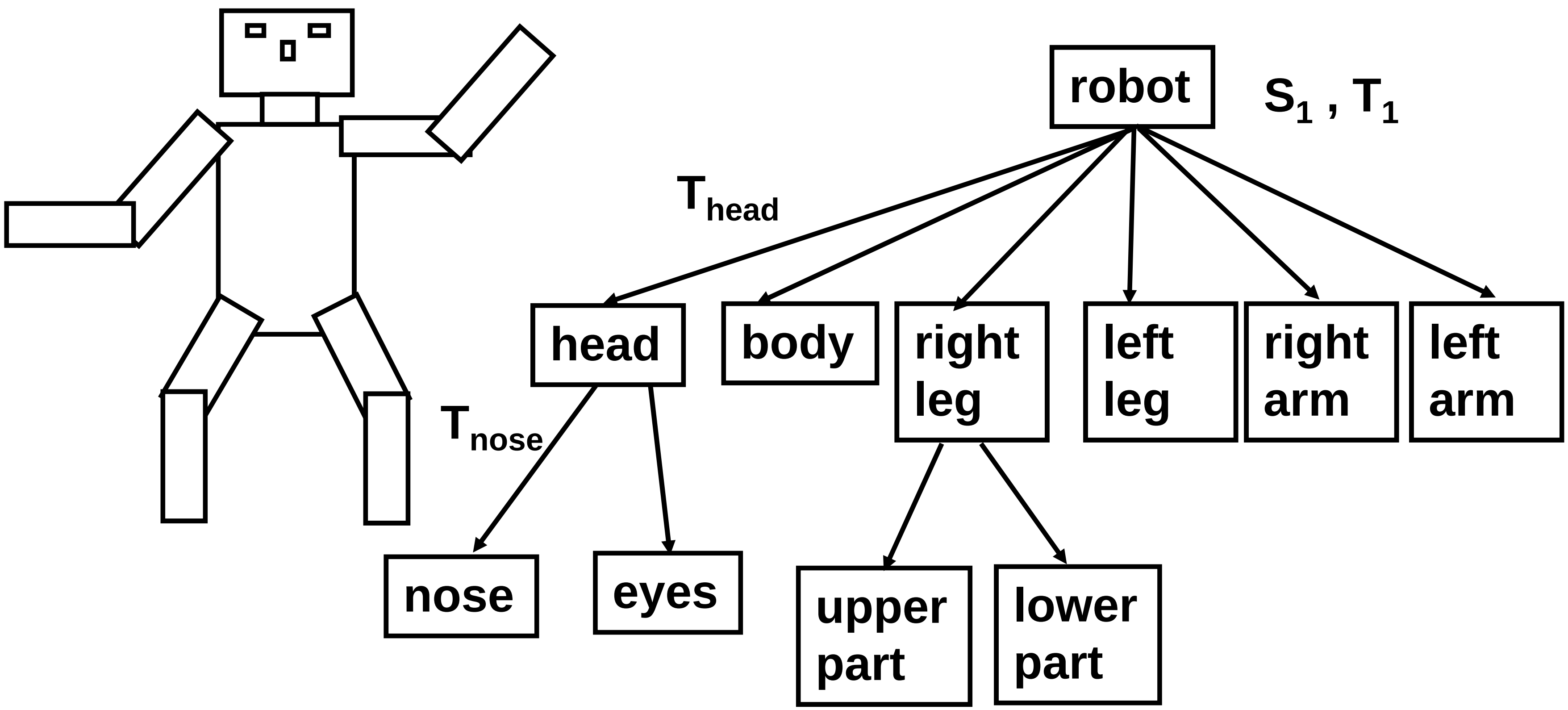
Idea: group parts hierarchically, associate transforms with each group.

whole robot = head + body + legs + arms
leg = upper part + lower part
head = neck + eyes + ...

Hierarchical transformations

- Hierarchical representation of an object is a tree.
- The non-leaf nodes are groups of objects.
- The leaf nodes are primitives (e.g. polygons)
- Transformations are assigned to each node, and represent the relative transform of the group or primitive with respect to the parent group
- As the tree is traversed, the transformations are combined into one

Hierarchical transformations



Transformation stack

To keep track of the current transformation,

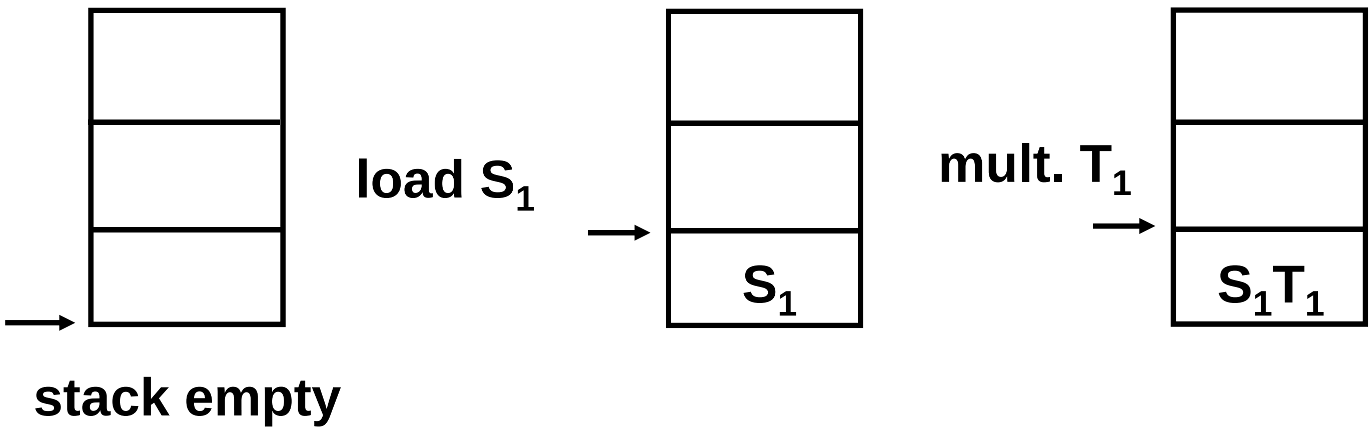
the transformation stack is maintained.

Basic operations on the stack:

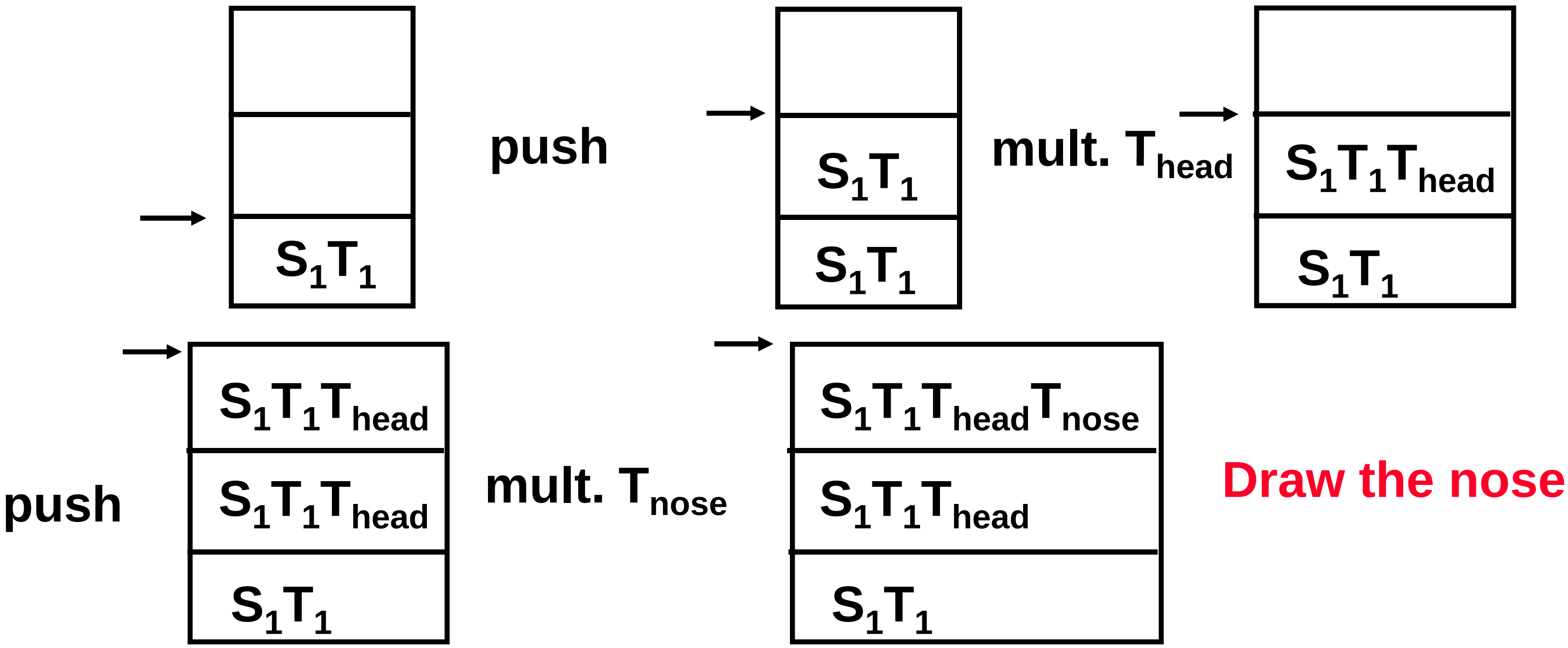
- push: create a copy of the matrix on the top and put it on the top
- pop: remove the matrix on the top
- multiply: multiply the top by the given matrix
- load: replace the top matrix with a given matrix

Transformation stack example

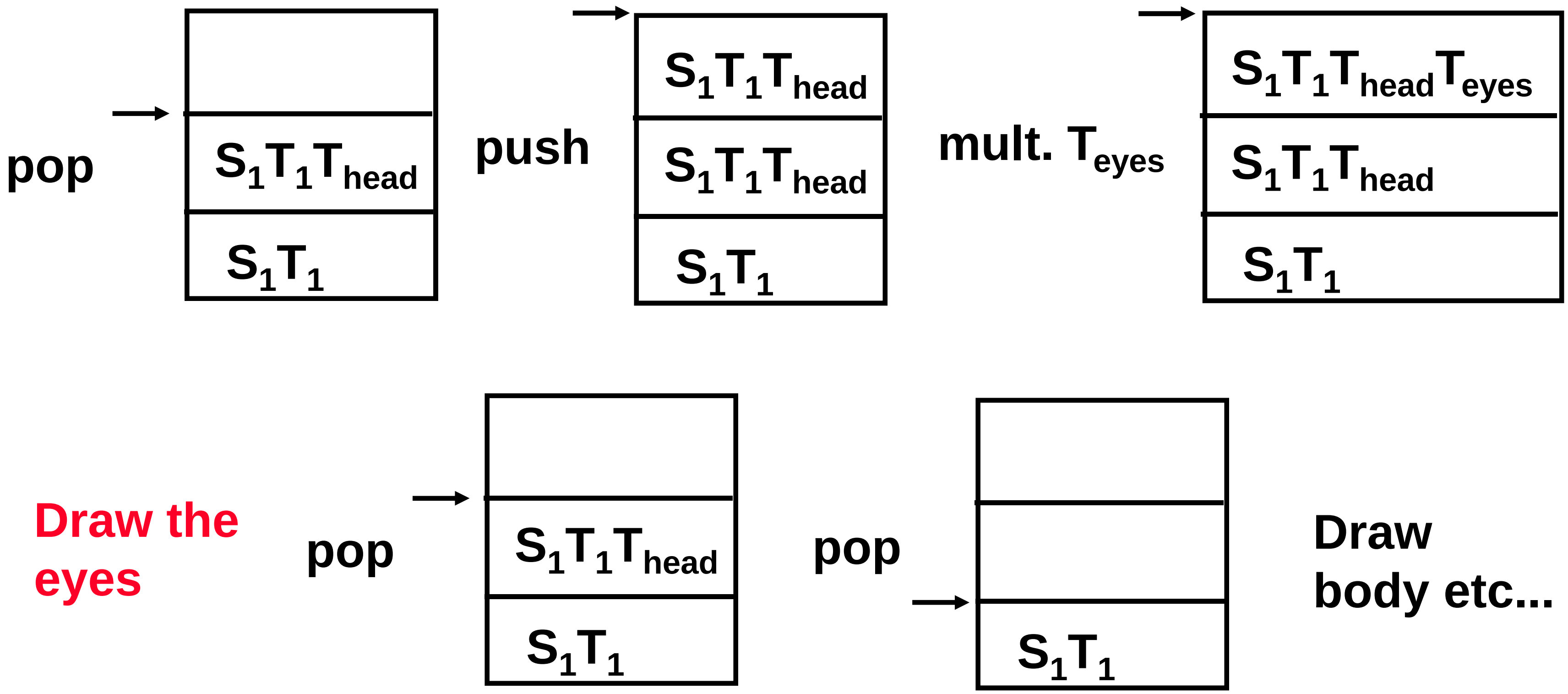
TO draw the robot, we use manipulations with the transform stack to get the correct transform for each part. For example, to draw the nose and the eyes:



Transformation stack example



Transformation stack example



Transformation stack example

Sequence of operations in the (pseudo)code:

```
load S1 ; mult T1;
push; mult. Thead;
  push;
     mult Tnose; draw nose;
  pop;
  push;
     mult. Teyes;  draw eyes;
  pop;
pop;
```

...

Animation

The advantage of hierarchical transformations is that everything can be animated with little effort.

General idea: before doing a mult. or load, compute

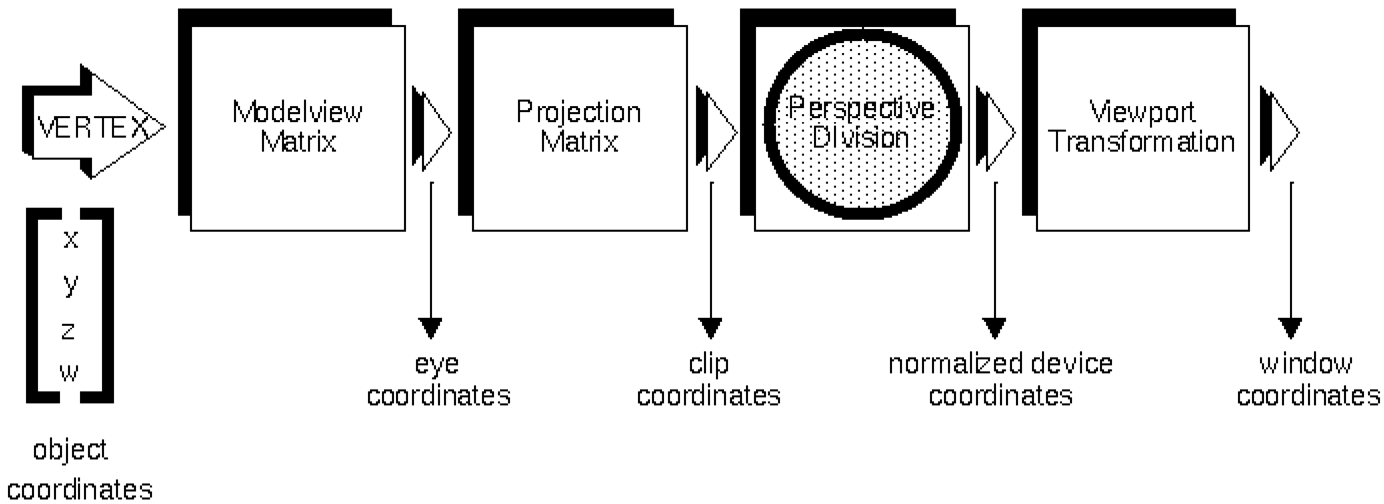
transform as a function of time.

```
time = 0;

main loop {

    draw(time);

    increment time;

}
```

```
draw( time ) {
...
compute R_arm(time)
mult. R_arm
...
}
```

Perspective transformations

Transformation pipeline



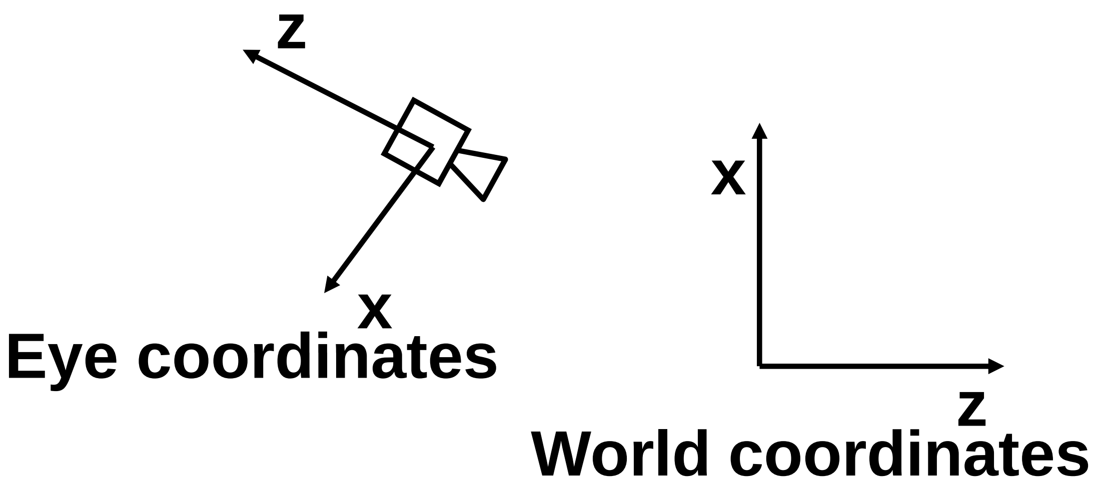
Modelview: model (position objects) + view (position the camera)

Projection: map viewing volume to a standard cube

Perspective division: project 3D to 2D

Viewport: map the square [-1,1]x[-1,1]
in normalized device coordinates to the screen

Coordinate systems



World coordinates - fixed initial coord system; everything is defined with respect to it

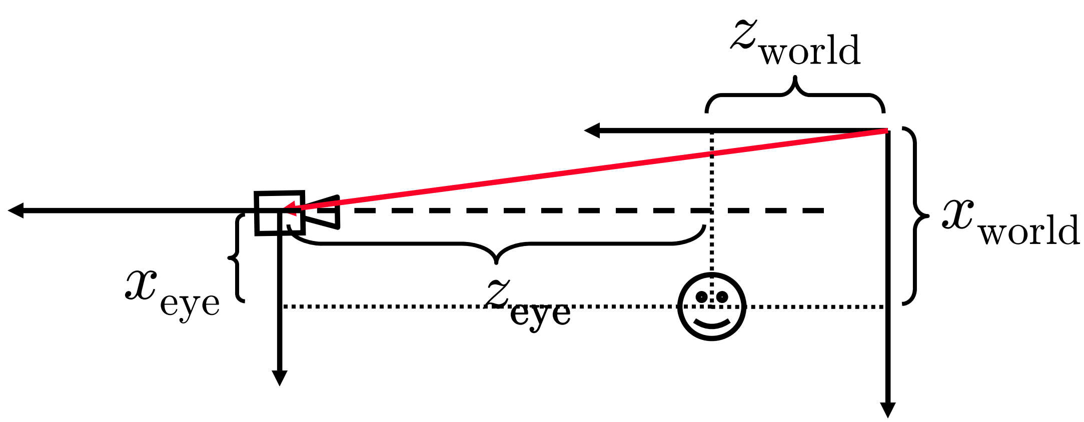
Eye coordinates - coordinate system attached to the camera; in this system camera looks down negative Z-axis

Positioning the camera

- Modeling transformation: reshape the object, orient the object, position the object with respect to the world coordinate system
- Viewing transformation: transform world coordinates to eye coordinates
- Viewing transformation is the *inverse* of the camera positioning transformation
- Viewing transformation should be rigid: rotation + translation
- Steps to get the right transform: first, orient the camera correctly, then translate it

Positioning the camera

Viewing transformation is the *inverse* of the camera positioning transformation:



Camera positioning: translate by (t_x, t_z)

Viewing transformation (world to eye):

$$x_{\text{eye}} = x_{\text{world}} - t_z$$

$$z_{\text{eye}} = x_{\text{world}} - t_x$$

Look-at positioning

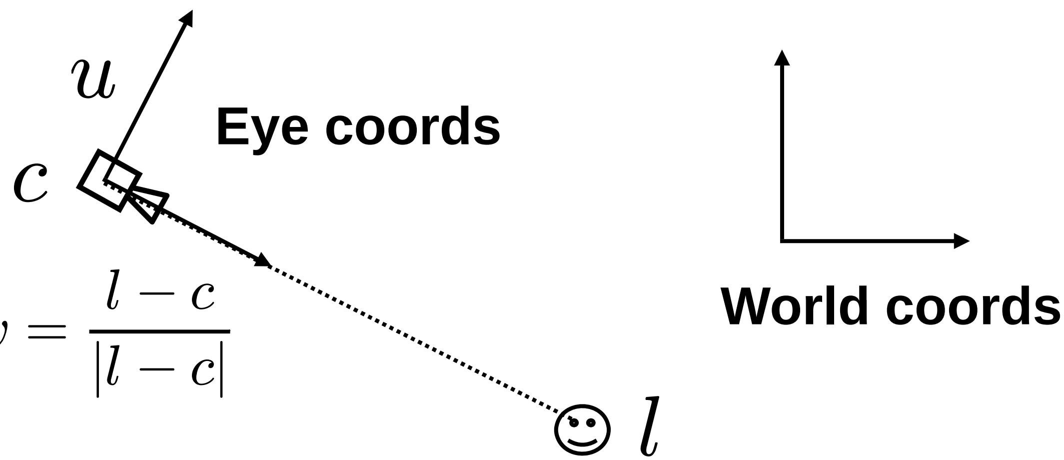
Find the viewing transform given the eye (camera) position, point to look at, and the up vector

- Need to specify two transforms: rotation and translation.
- translation is easy
- natural rotation: define implicitly using a point at which we want to look and a vector indicating the vertical in the image (*up vector*)

can easily convert the eye point to the direction vector of the camera axis; can assume up vector perpendicular to view vector

Look-at positioning

Problem: given two pairs of perpendicular unit vectors, find the transformation mapping the first pair into the second



$$v = \frac{l - c}{|l - c|}$$

Look-at positioning

Determine rotation first,
looking how coord vectors change:

$$R\begin{bmatrix}0\\0\\-1\end{bmatrix}=v \qquad R\begin{bmatrix}1\\0\\0\end{bmatrix}=v\times u \qquad R\begin{bmatrix}0\\1\\0\end{bmatrix}=u$$

$$R\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}=R=[v\times u, u, -v]$$

Look-at positioning

Recall the matrix for translation:

$$T = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now we have the camera positioning matrix, TR

To get the viewing transform, invert: $(TR)^{-1} = R^{-1}T^{-1}$

For rotation the inverse is the transpose!

$$R^{-1} = [v \times u \; u \; -v]^T = \begin{bmatrix} (v \times u)^T \\ u^T \\ -v^T \end{bmatrix}$$

Look-at viewing transformation

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = [e_x \; e_y \; e_z \; -c]$$

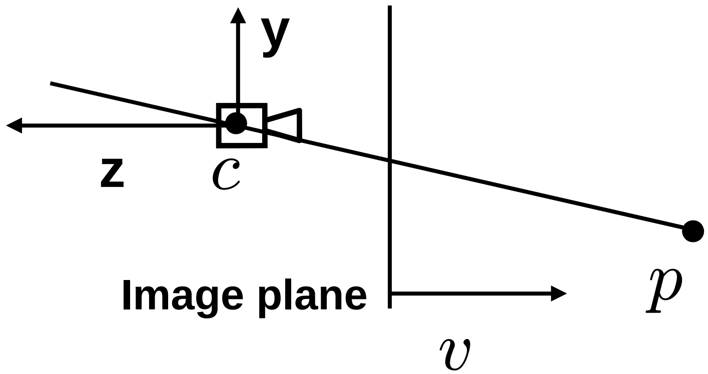
$$\boxed{V = R^{-1}T^{-1} = \begin{bmatrix} (v \times u)^T & -(v \times u \cdot c) \\ u^T & -(u \cdot c) \\ -v^T & (v \cdot c) \\ [0,0,0] & 1 \end{bmatrix}}$$

Positioning the camera in OpenGL

- imagine that the camera is an object and write a sequence of rotations and translations positioning it
- change each transformation in the sequence to the opposite
- reverse the sequence
- Camera positioning is done in the code *before* modeling transformations
- OpenGL does not distinguish between viewing and modeling transformation and joins them into the modelview matrix

Space to plane projection

In eye coordinate system



$$c = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad v = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

projecting to the plane
z = -1

$$\mathrm{Proj}(p) = \begin{bmatrix} -p_x/p_z \\ -p_y/p_z \\ -1 \end{bmatrix}$$

$$\mathrm{Proj}(p) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Visibility

Objects that are closer to the camera occlude the objects that are further away

- All objects are made of planar polygons
- A polygon typically projects 1 to 1
- idea: project polygons in turn ; for each pixel, record distance to the projected polygon
- when writing pixels, replace the old color with the new one only if the new distance to camera for this pixel is less then the recorded one

Z-buffering idea

- *Problem:* need to compare distances for each projected point
- *Solution:* convert all points to a coordinate system in which (x,y) are image plane coords and the distance to the image plane increases when the z coordinate increases
- In OpenGL, this is done by the projection matrix

Z buffer

Assumptions:

- **each pixel has storage for a z-value, in addition to RGB**
- **all objects are "scanconvertible" (typically are polygons, lines or points)**

Algorithm:

```
initilize zbuf to maximal value

for each  object
   for each pixel (i,j) obtained by scan conversion
        if znew(i,j) < zbuf(i,j)
                 zbuf(i,j) = znew(i,j) ;
                 write pixel(i,j)
```

Z buffer

What are z values?

Z values are obtained by applying the projection transform, that is, mapping the viewing frustum to the standard cube.

Z value increases with th distance to the camera.

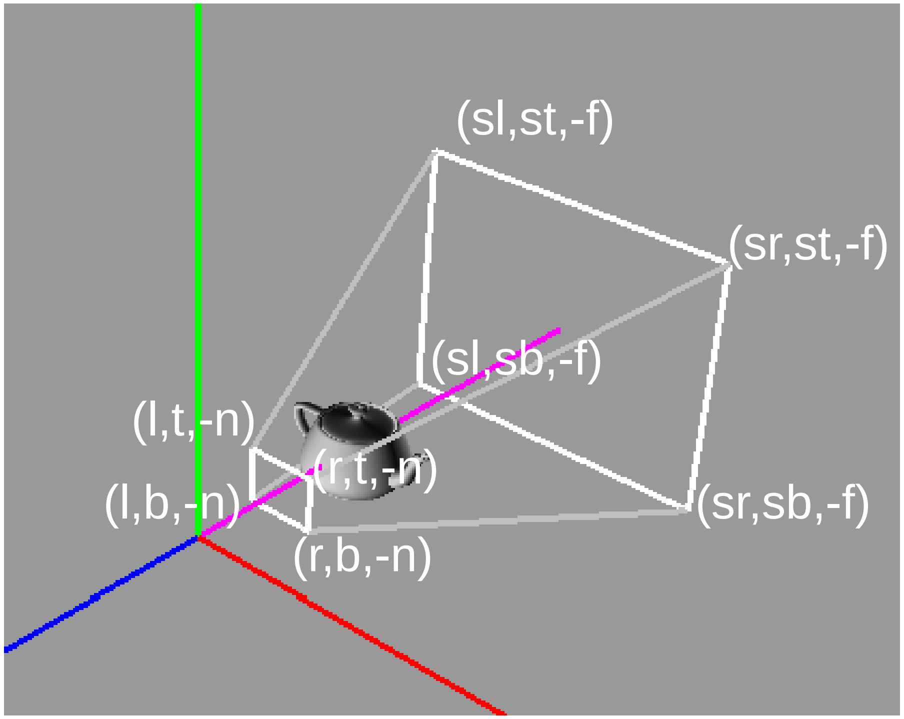
Z values for each pixel are computed for each pixel covered by a polygon using linear interpolation of z values at vertices.

Typical Z buffer size: 24 bits (same as RGB combined).

Camera specification

Define the dimensions of the viewing volume (frustum)

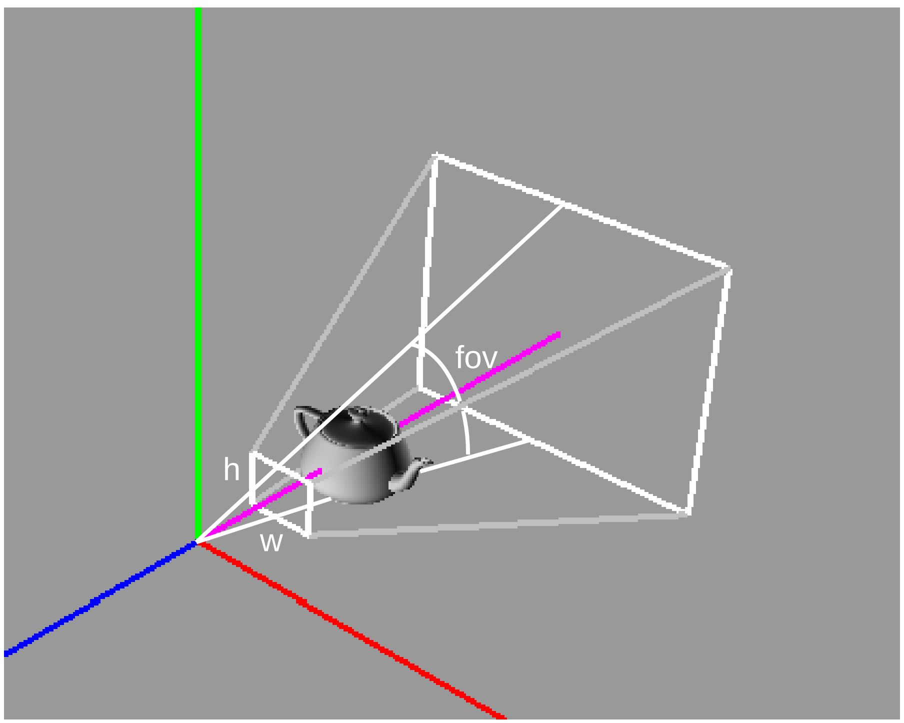
- **most general glFrustum(left,right,bottom,top,near,far)**



In the picture:
l = left
r = right
b = bottom
t = top
n = near
f = far
s = far/near

Camera specification

Less general but more convenient:
gluPerspective(field_of_view,aspect_ratio, near,far)



In the picture:
fov = field of view,
h/w = a=aspect ratio

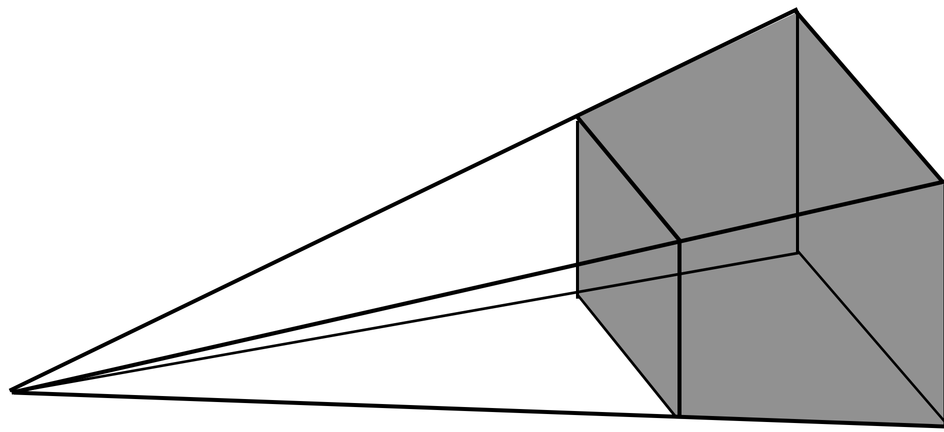
Relationship to frustum:

left	= -a*near*tan(fov/2)
right	= a*near*tan(fov/2)
bottom	= -a*near*tan(fov/2)
top	= a*near*tan(fov/2)

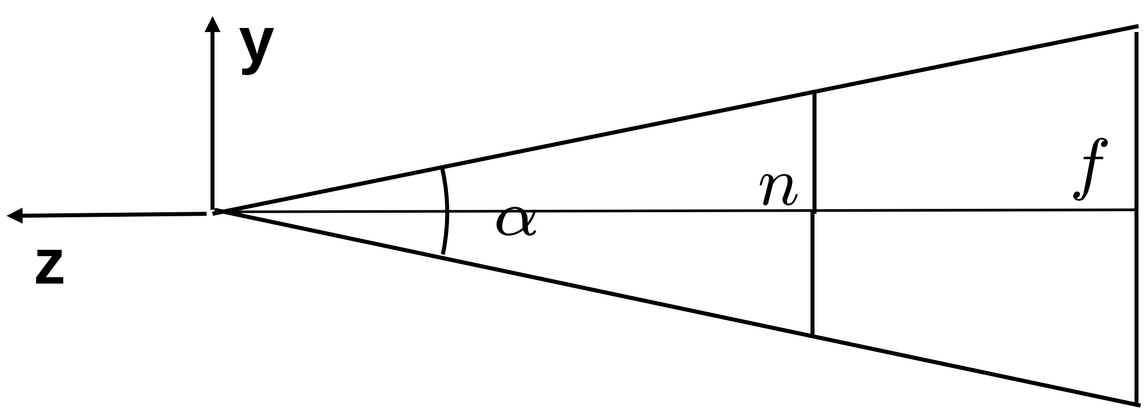
gluPerspective requires fov in degrees, not radians!

Viewing frustum

Volume in space that will be visible in the image



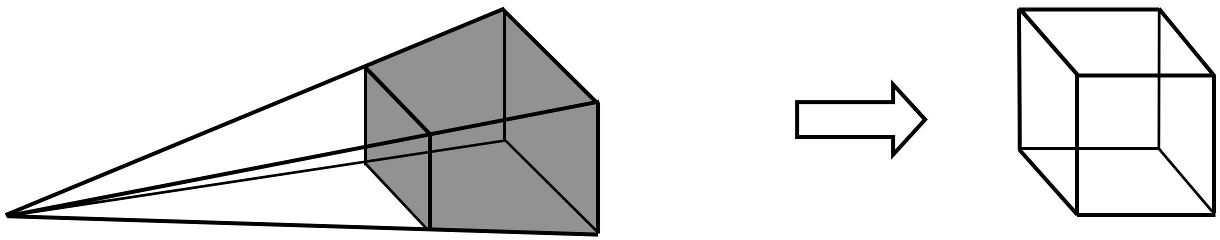
r **is the aspect ratio of the image width/height**



Projection transformation

Maps the viewing frustum into a standard cube extending from -1 to 1 in each coordinate

(normalized device coordinates)



3 steps:

- **change the matrix of projection to keep z:**
  - **result is a parallelepiped**
- **translate:**
  - **parallelepiped centered at 0**
- **scale in all directions:**
  - **cube of of size 2 centered at 0**

Projection transformation

change $\mathrm{Proj}(p) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$ **so that we keep z:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ 1 \\ -p_z \end{bmatrix}$$

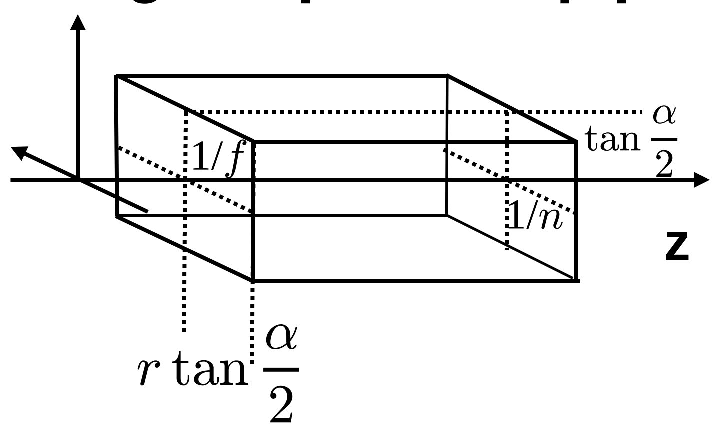
the homogeneous result corresponds to $\begin{bmatrix} -p_x/p_z \\ -p_y/p_z \\ -1/p_z \end{bmatrix}$

the last component increases monotonically with z!

Projection transformation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

maps the frustum to an axis-aligned parallelepiped



already centered in (x,y), center in z-direction and scale:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{2}\left(\frac{1}{f} + \frac{1}{n}\right) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} \frac{1}{r\tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{2}{\left(\frac{1}{n} - \frac{1}{f}\right)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection transformation

Combined matrix, mapping frustum to a cube:

$$P = ST \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{r \tan \frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & 2\frac{fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

To get normalized image plane coordinates (valid range [-1,1] both) , just drop z in the result and convert from homogeneous to regular.

To get pixel coordinates, translate by 1, and scale x and y (Viewport transformation)