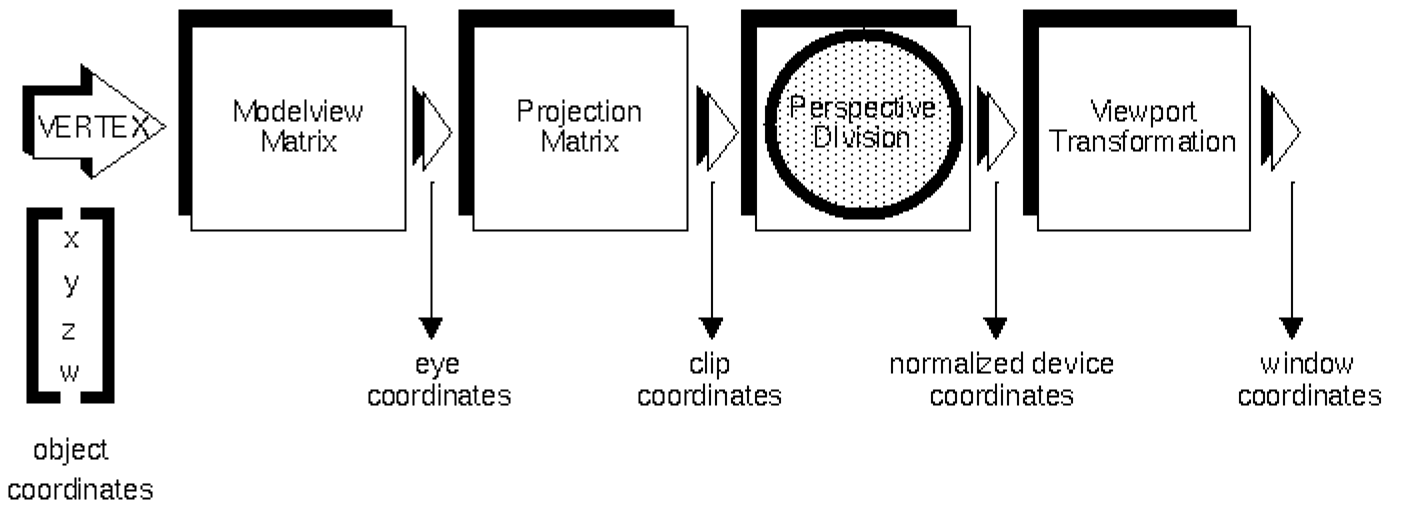
2D transformations, homogeneous coordinates, hierarchical transformations

Transformation pipeline



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

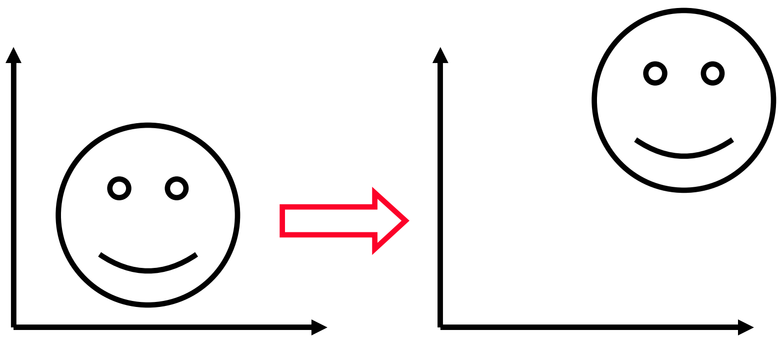
Projection: map viewing volume to a standard cube

Perspective division: project 3D to 2D

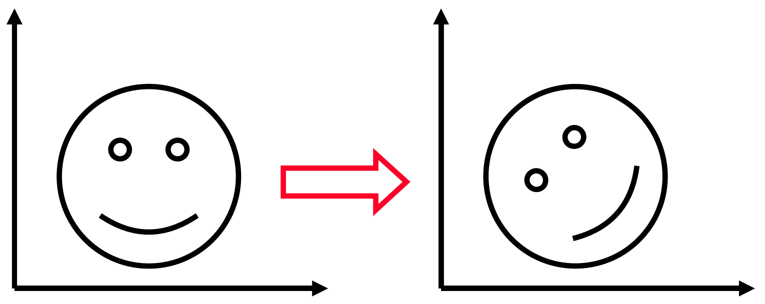
Viewport: map the square [-1,1]x[-1,1] in normalized device coordinates to the screen

Transformations

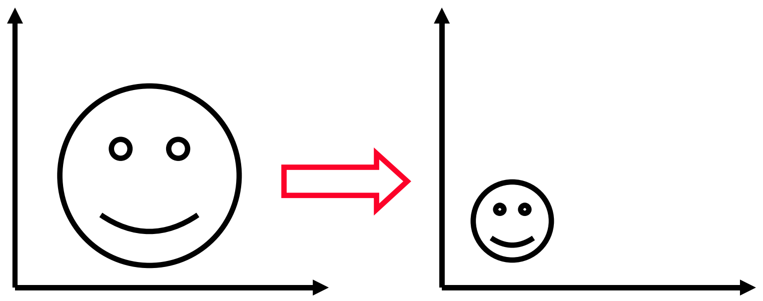
Examples of transformations:



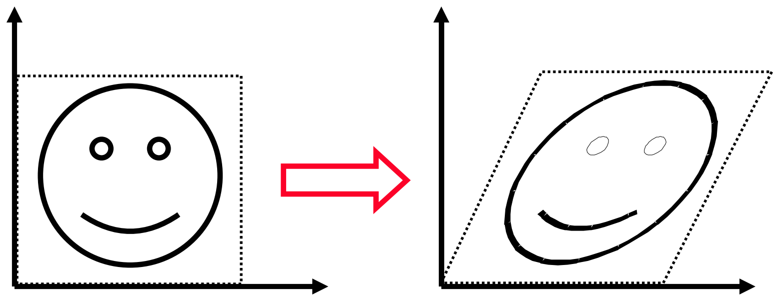
translation



rotation



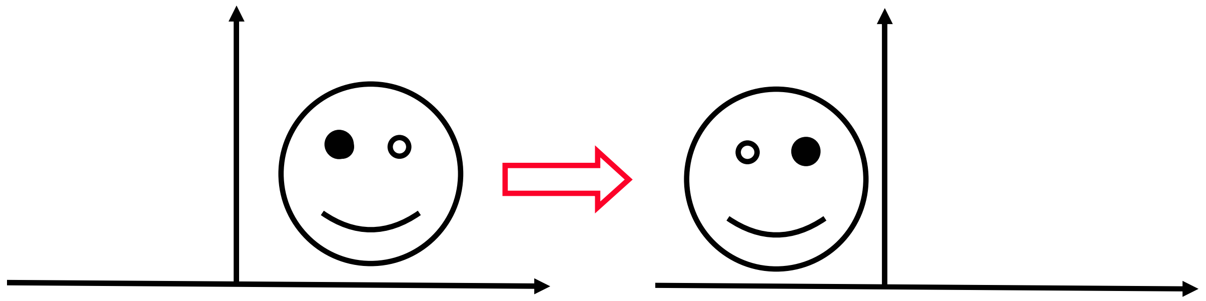
scaling



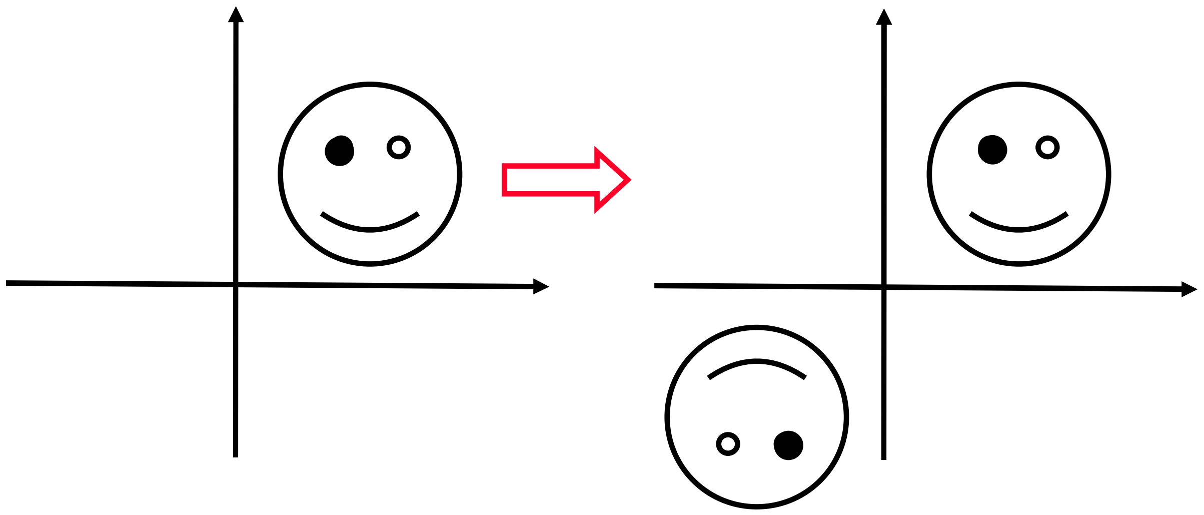
shear

Transformations

More examples:



reflection with respect to the *y* axis



reflection with respect to the origin

Transformations

Linear transformations: take straight lines to straight lines.

All of the examples are linear.

Affine transformations: take paralllel lines to parallel lines.

All of the examples are affine,

an example of linear non- affine is perspective projection.

Orthogonal transformations: preserve distances, move all objects as rigid bodies.

rotation, translation and reflections are affine.

Transformations and matrices

Any affine transformation can be written as

$$\begin{pmatrix} \mathbf{x'} \\ \mathbf{y'} \end{pmatrix} = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{a}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} + \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}$$

p'=Ap

Images of basis vectors under affine transformations:

$$\mathbf{e}_\mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

(column form of writing vectors)

$$\mathbf{Ae}_\mathbf{x} = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{a}_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{a}_{11} \\ \mathbf{a}_{21} \end{pmatrix} \qquad \mathbf{Ae}_\mathbf{y} = \begin{pmatrix} \mathbf{a}_{21} \\ \mathbf{a}_{22} \end{pmatrix}$$

$$\mathbf{e}_\mathbf{y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Transformations and matrices

Matrices of some transformations:

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$ **shear** $$\begin{pmatrix} \mathbf{s} & 0 \\ 0 & \mathbf{s} \end{pmatrix}$$ **scale by factor s**

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$ **rotation**

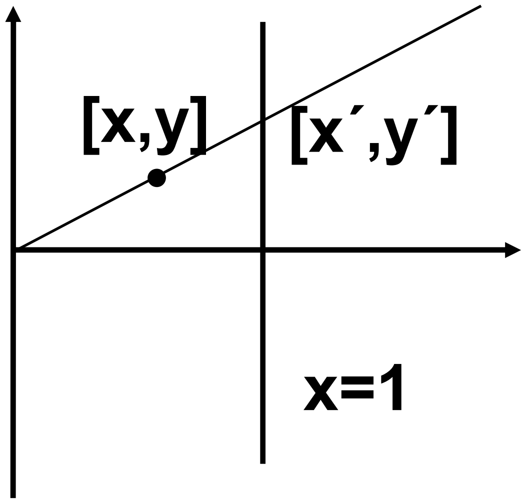
$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$ **reflection with respect to the origin**

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$ **reflection with respect to y axis**

Problem

Even for affine transformations we cannot write them as a single 2x2 matrix; we need an additional vector for translations.

We cannot write all linear transformations even in the form Ax +b where A is a 2x2 matrix and b is a 2d vector. Example: perspective projection



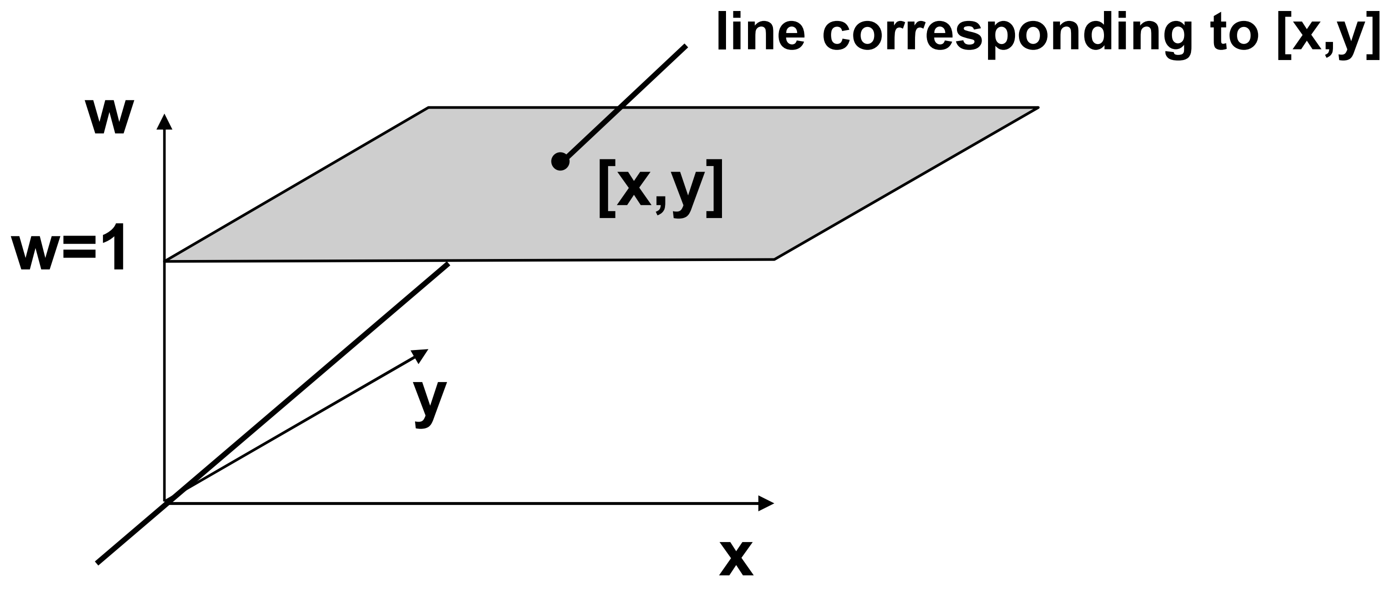
$x' = 1$
$y' = y/x$

equations not linear!

Homogeneous coordinates

- replace 2d points with 3d points, last coordinate 1
- for a 3d point (x,y,w) the corresponding 2d point is (x/w,y/w) if w is not zero
- each 2d point (x,y) corresponds to a line in 3d; all points on this line can be written as [kx,ky,k] for some k.
- (x,y,0) does not correspond to a 2d point, corresponds to a direction (will discuss later)
- Geometric construction: 3d points are mapped to 2d points by projection to the plane z =1 from the origin

Homogeneous coordinates



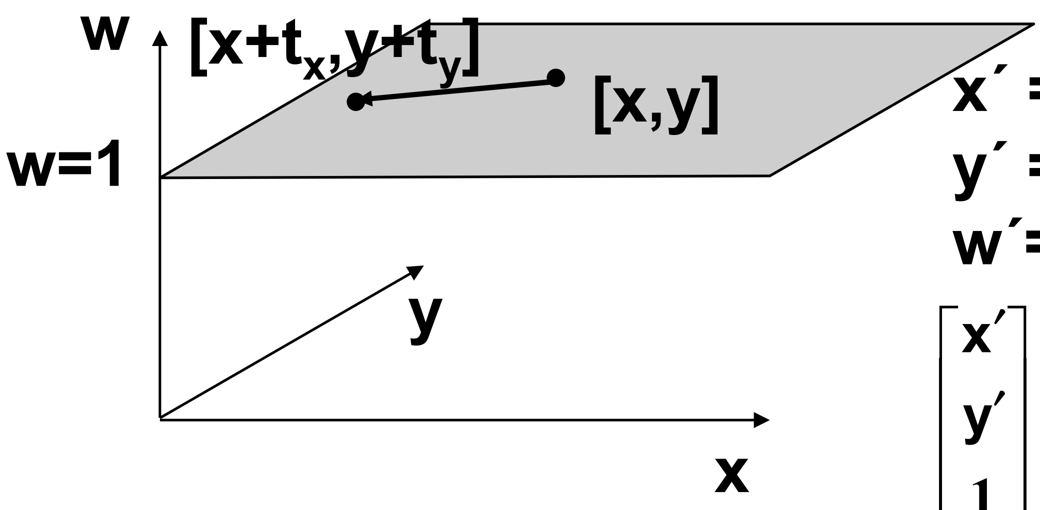
From homogeneous to 2d: [x,y,w] becomes [x/w,y/w]
From 2d to homogeneous: [x,y] becomes [kx,ky,k]
(can pick any nonzero k!)

Homogeneous transformations

Any linear transformation can be written in matrix form in homogeneous coordinates.

Example 1: translations

[x,y] in hom. coords is [x,y,1]



$x' = x+t_x = x + t_x \cdot 1$
$y' = y+t_y = y+t_y \cdot 1$
$w' = 1$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

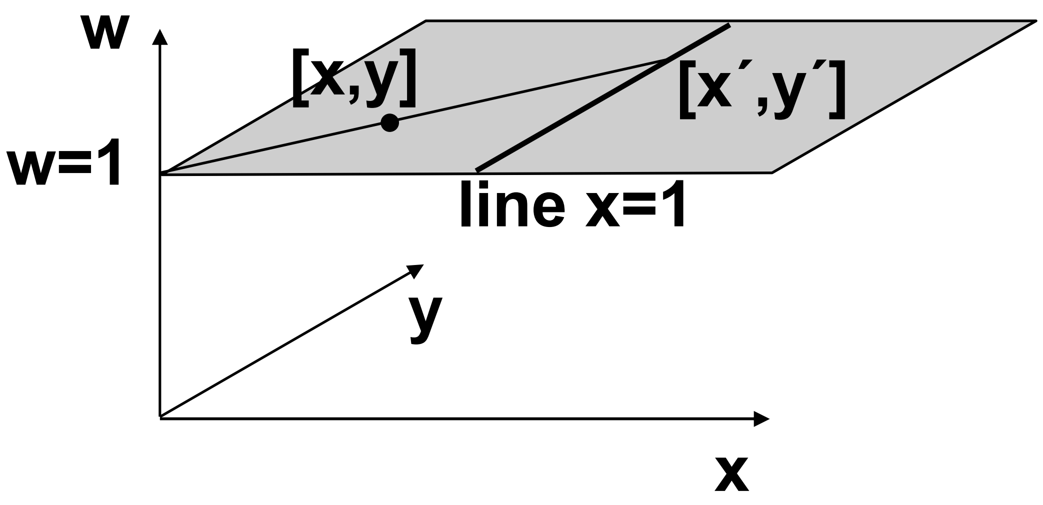
Homogeneous transformations

Example 2: perspective projection

$x' = 1$
$y' = y/x$
$w' = 1$

Can multiply all three components by the same number-- the 2D point won't change! Multiply by x.

$x' = x$
$y' = y$
$w' = x$



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Matrices of basic transformations

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

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

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

$$\begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ skew

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$$ general affine transform

Composition of transformations

- Order matters! (rotation * translation $\neq$ translation * rotation)
- Composition of transformations = matrix multiplication:
  if T is a rotation and S is a scaling, then applying scaling first and rotation second is the same as applying transformation given by the matrix TS (note the order).
- Reversing the order does not work in most cases

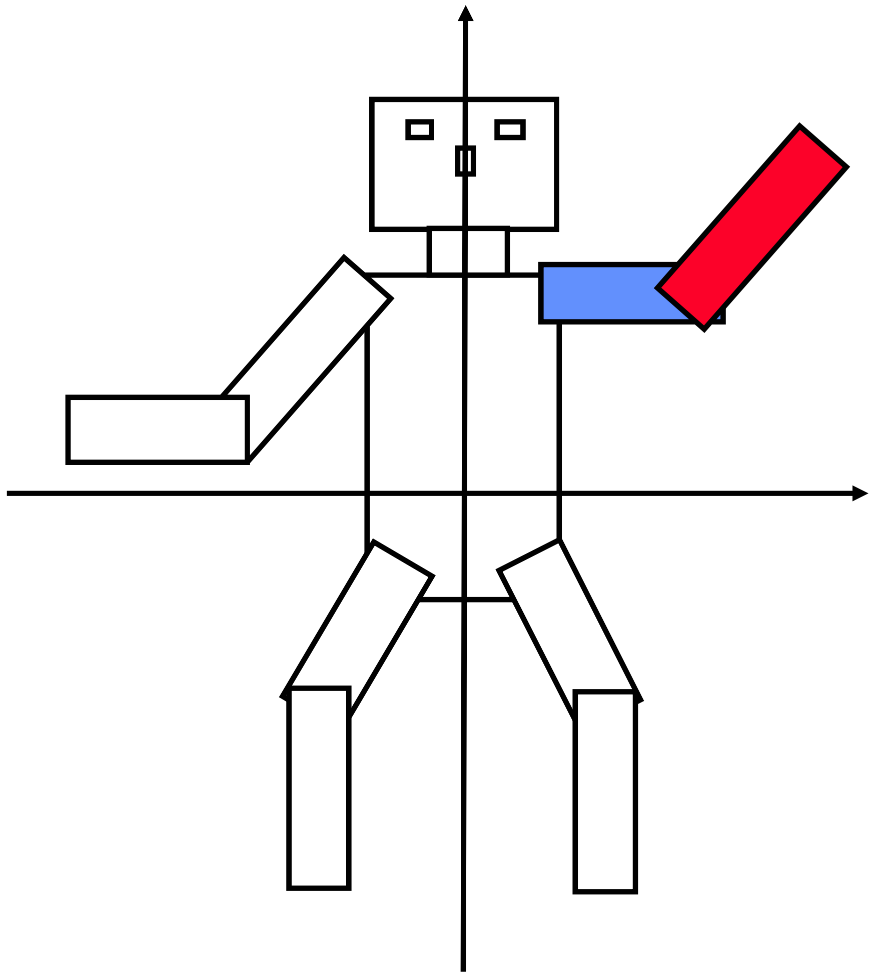
Transformation order

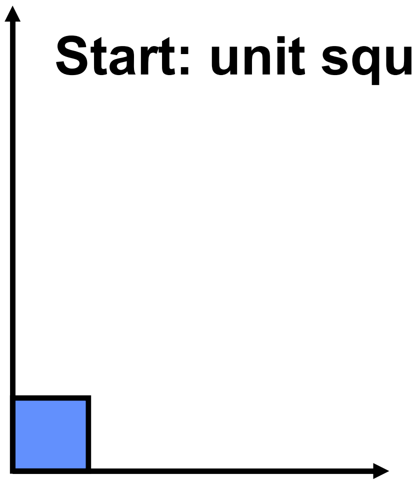
- When we write transformations using standard math notation, the closest transformation to the point is applied first:

$$T\ R\,S\,p = T(R(Sp))$$

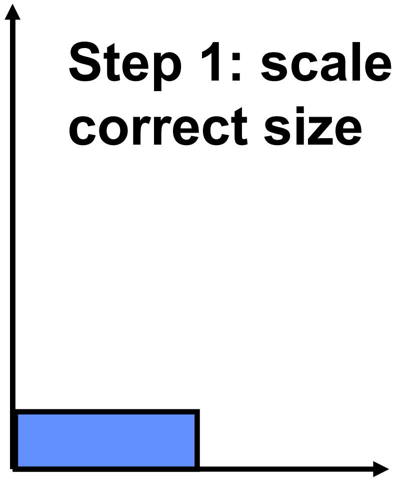
- first, the object is scaled,
  then rotated,
  then translated
- This is the most common transformation order for an object (scale- rotate- translate)

Building the arm



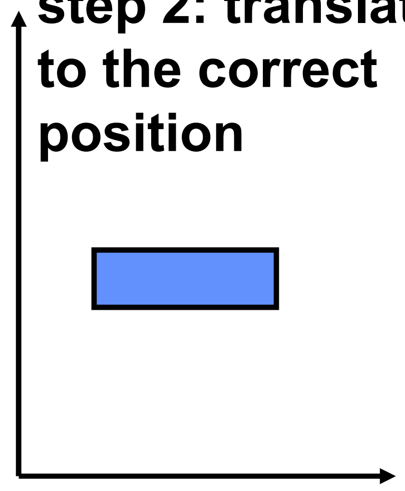


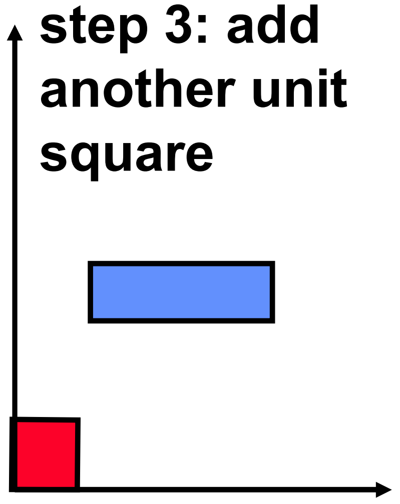
Start: unit square

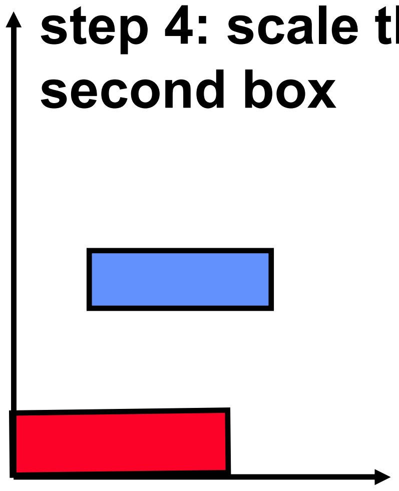


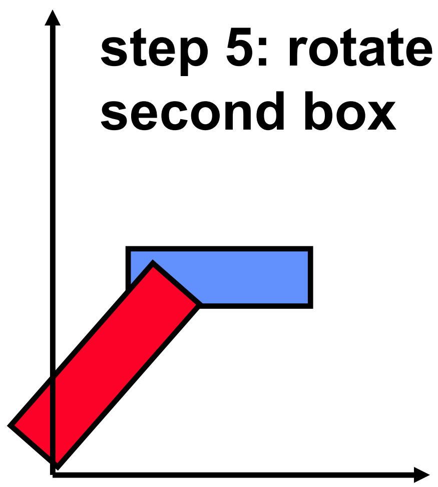
Step 1: scale to the correct size

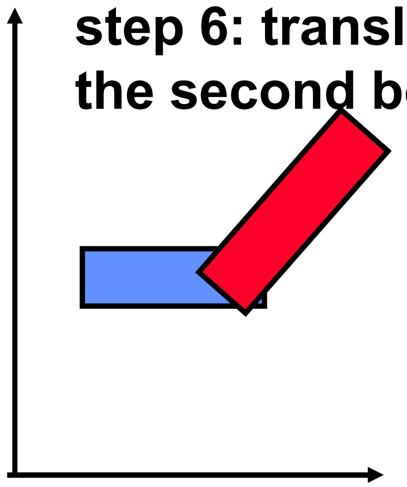
Building the arm


step 2: translate to the correct position


step 3: add another unit square


step 4: scale the second box

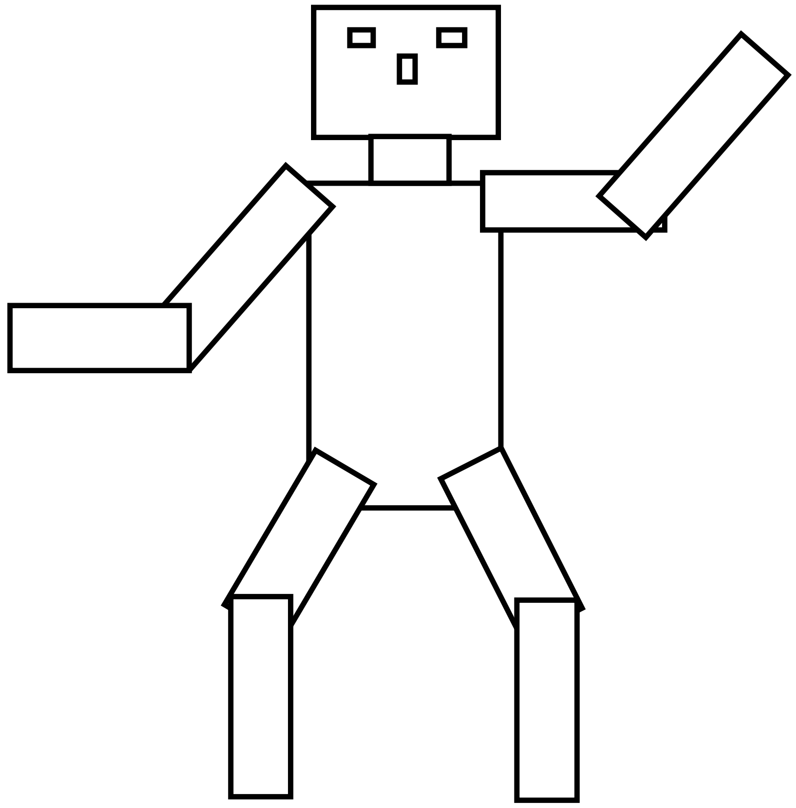

step 5: rotate the second box


step 6: translate the second box

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



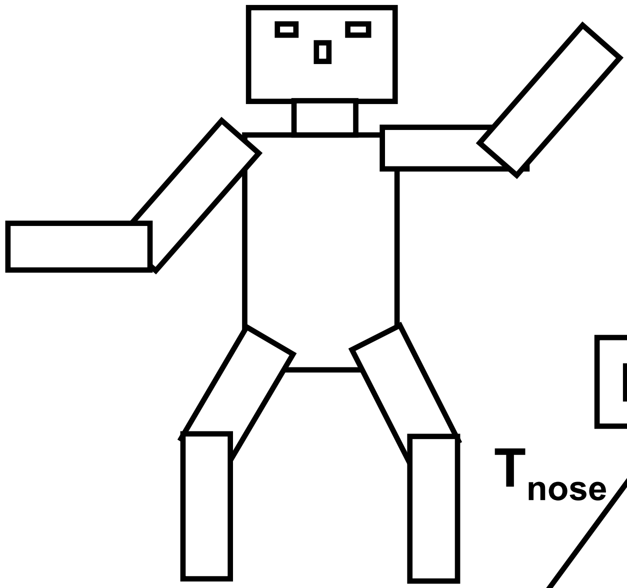
Idea: group parts hierarchically, associate transforms with each group.

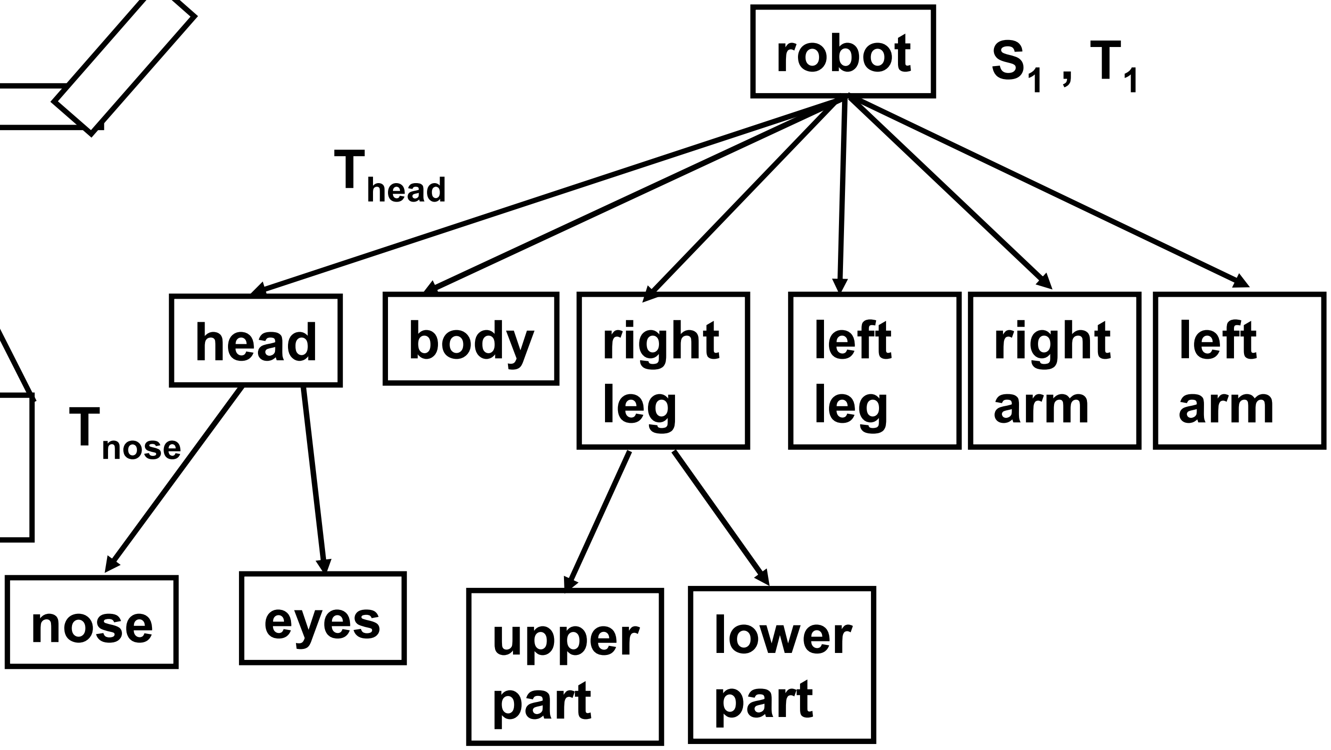
whole robot = head + body + legs + arms
leg = upper part + lower part
head = neck + eyes + ...

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,

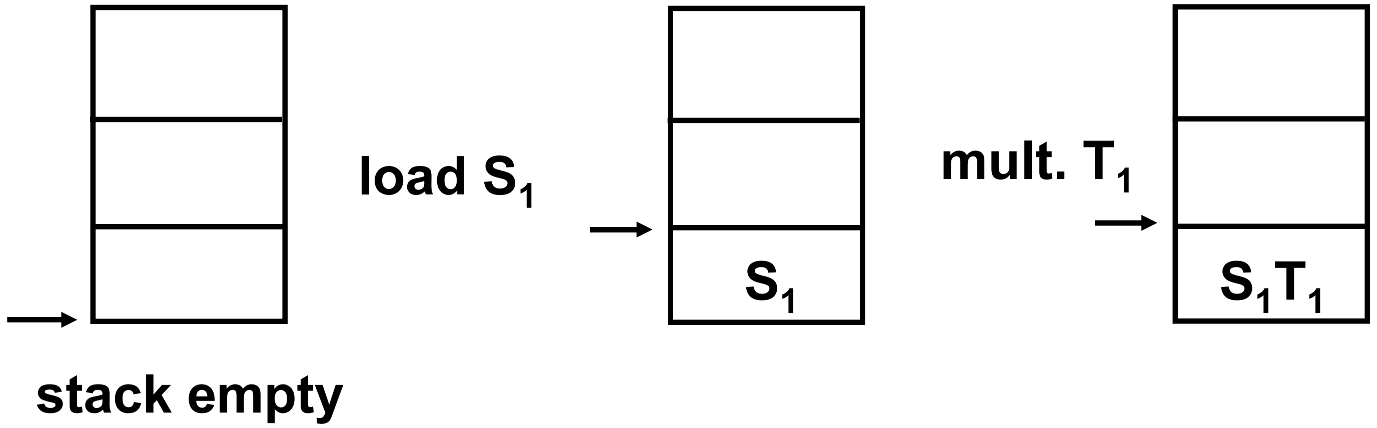
the transformation stack is maintained.

Basic operations on the stack:

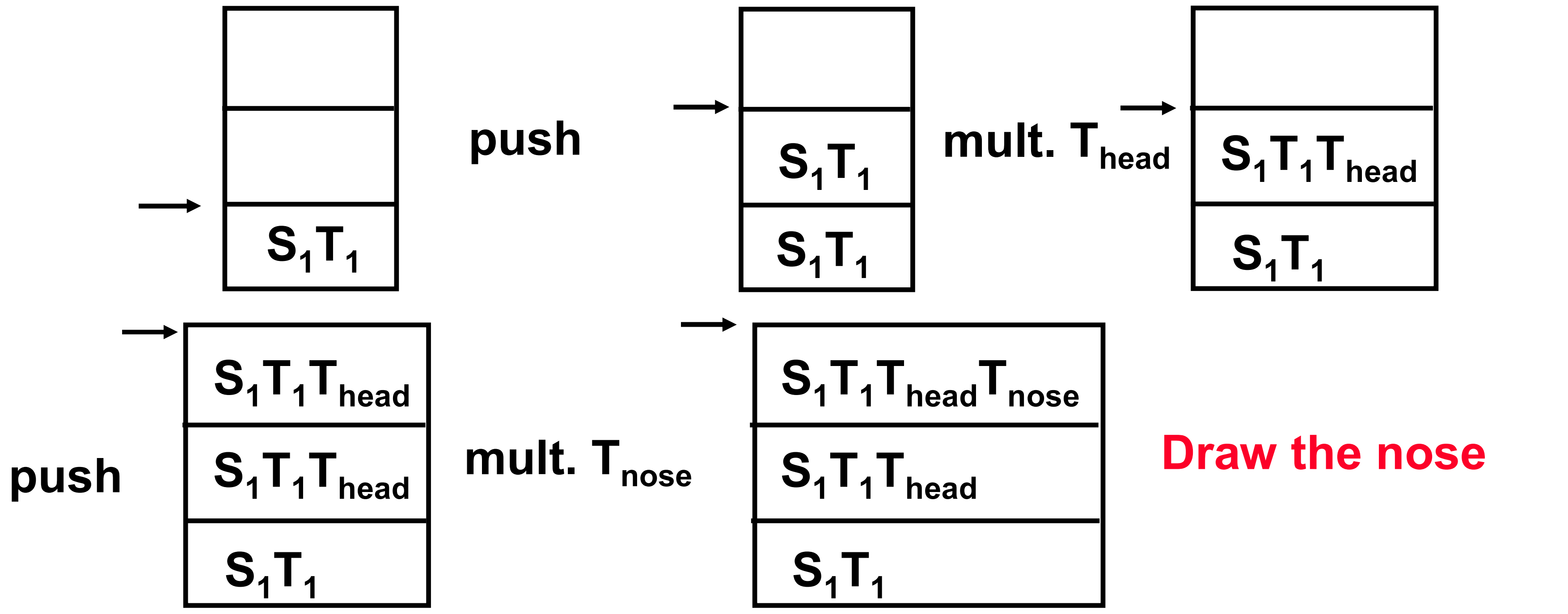
- push: create a copy of the matrix on the top and put it on the top; glPushMatrix
- pop: remove the matrix on the top; glPopMatrix
- multiply: multiply the top by the given matrix; glMultMatrixf, also glTransalatef,glRotatef,glScalef etc.
- load: replace the top matrix with a given matrix glLoadMatrixf, glLoadIdentity

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:

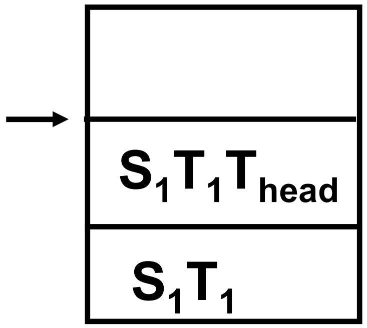


Transformation stack example

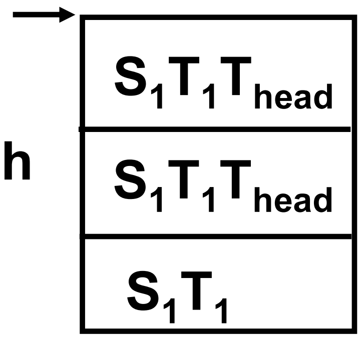


Transformation stack example

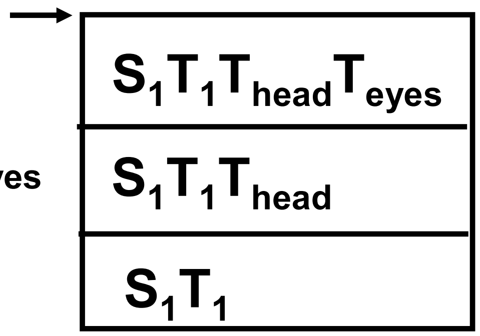
pop



push

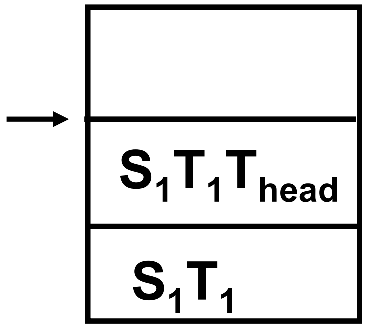


mult. T_{eyes}

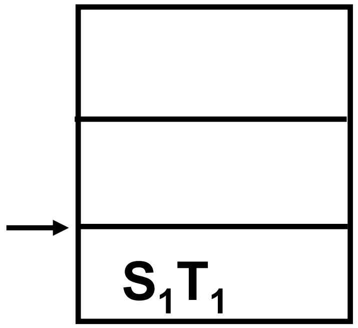


Draw the eyes

pop



pop



Draw body etc...

Transformation stack example

Sequence of operations in the (pseudo)code:

```
load $S_1$ ; mult $T_1$;
push; mult. $T_{head}$;
  push;
     mult $T_{nose}$; draw nose;
  pop;
  push;
     mult. $T_{eyes}$;  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. $R_{arm}$
...
}
```