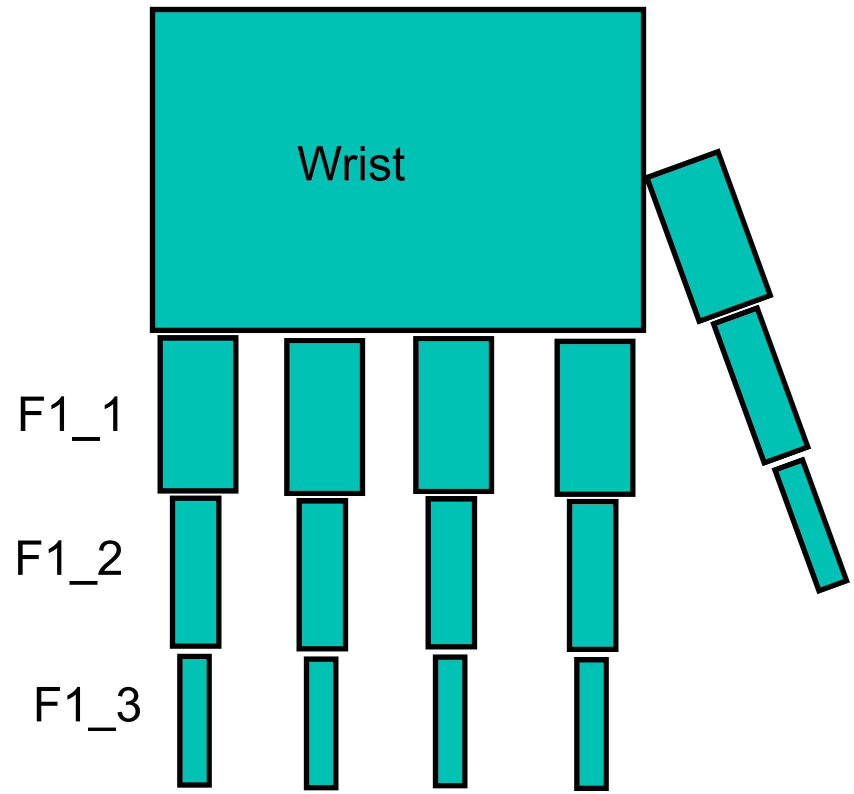
Hierarchical structures

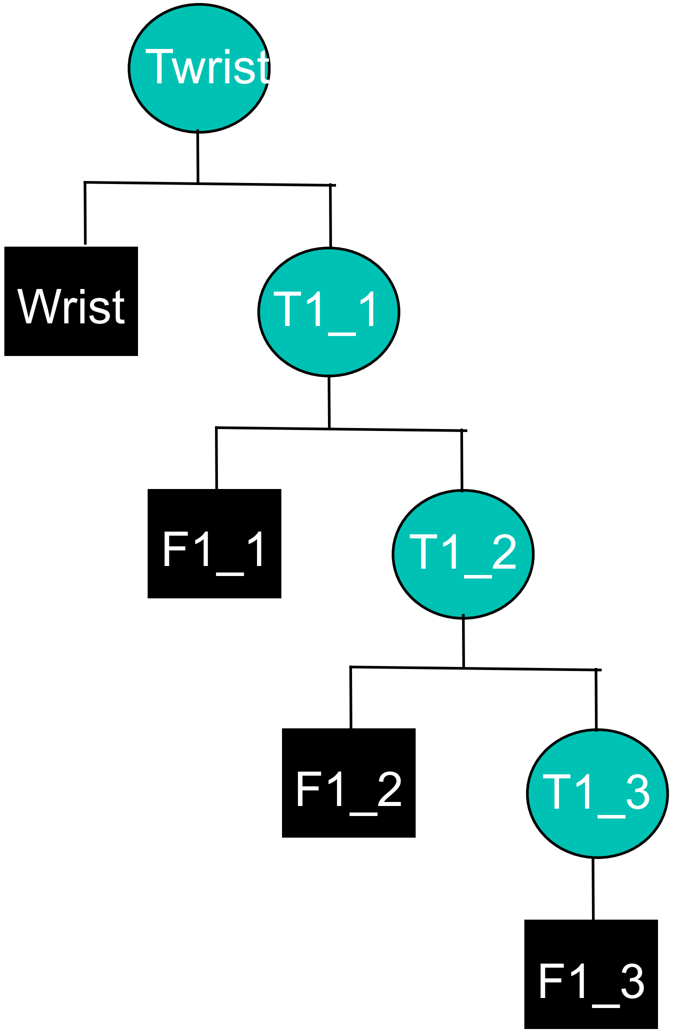
Wrist and 5 fingers

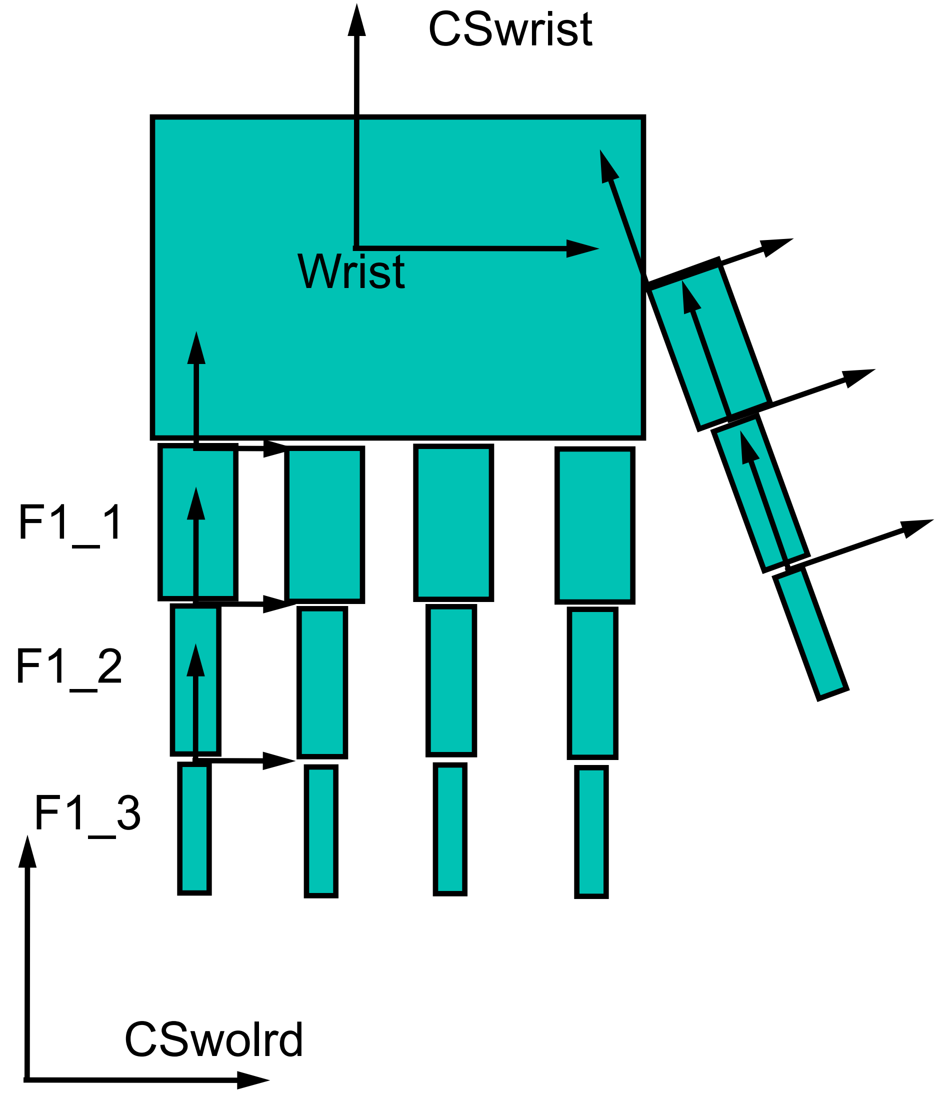
We want the fingers to stay attached to the wrist as the wrist moves.

Each segment is abstracted by a coordinate system and has its own transformation matrix with respect to its parent's system thus modelling relative motion.



Hierarchy of systems



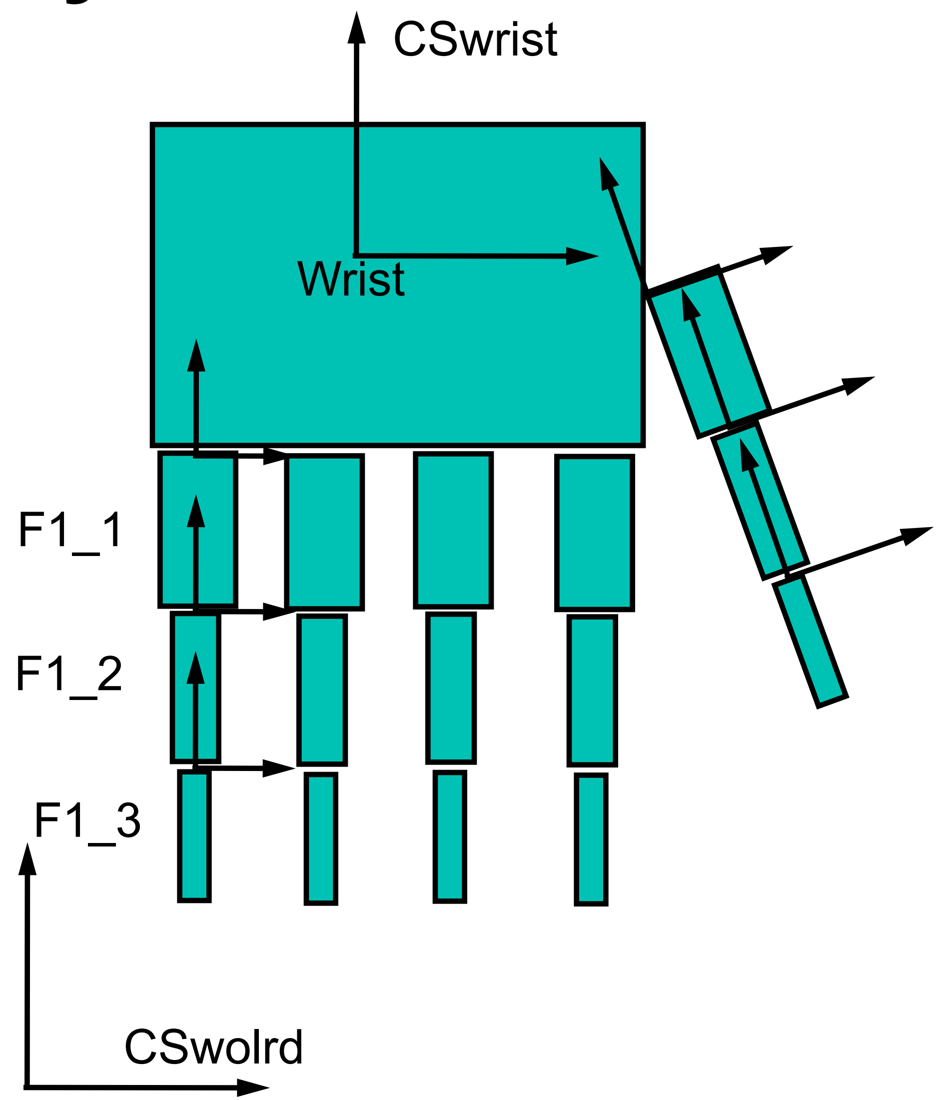


Hierarchy of systems

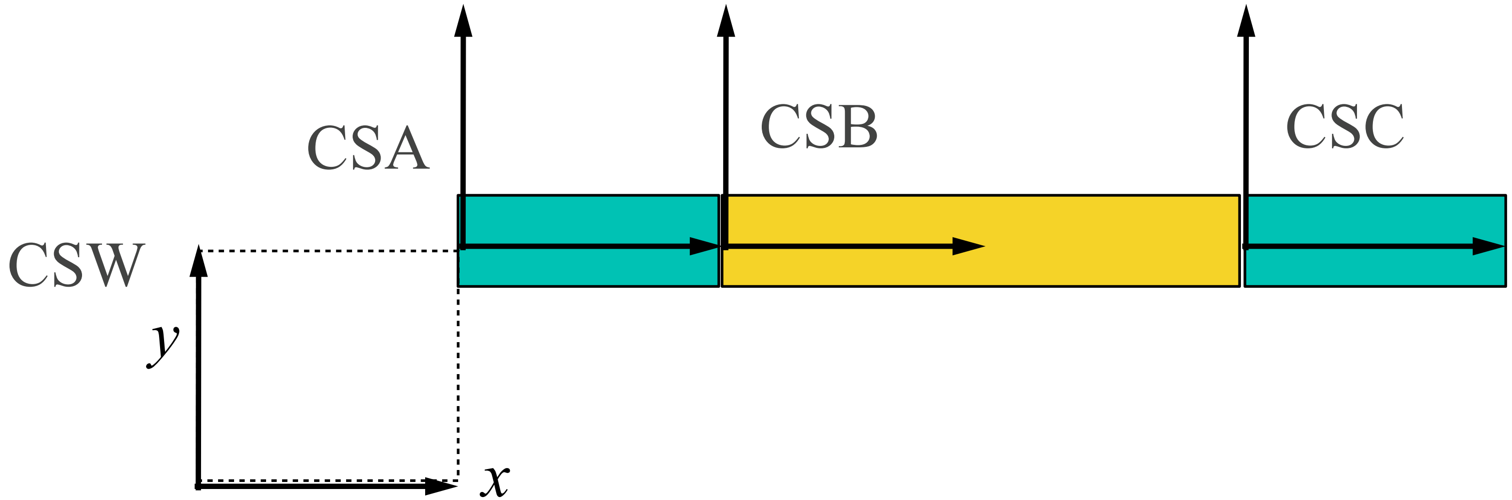
CSF1_1 = T1_1(CSwrist)

CSF1_2 = T1_2(CSF1_1)

CSF1_3 = T1_3(CSF1_2)



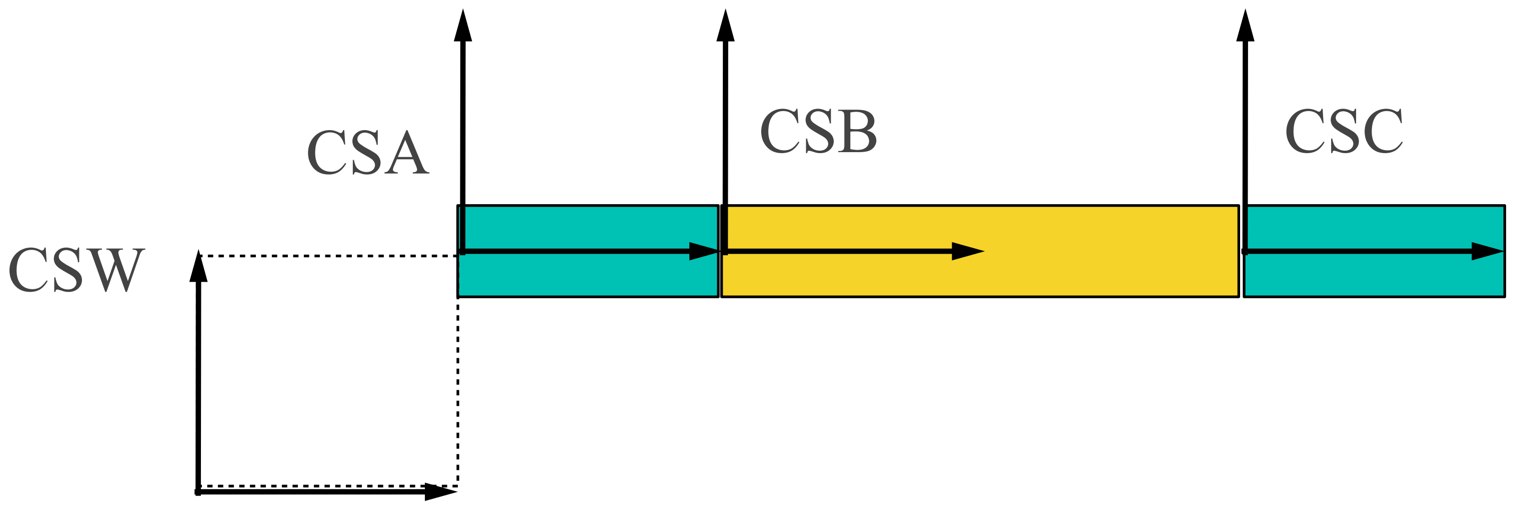
Example: Kinematic Chain



$P_w = {}_wM_{A\,A}M_{B\,B}M_C P_C$

What do these matrices look like?

Example: Kinematic Chain

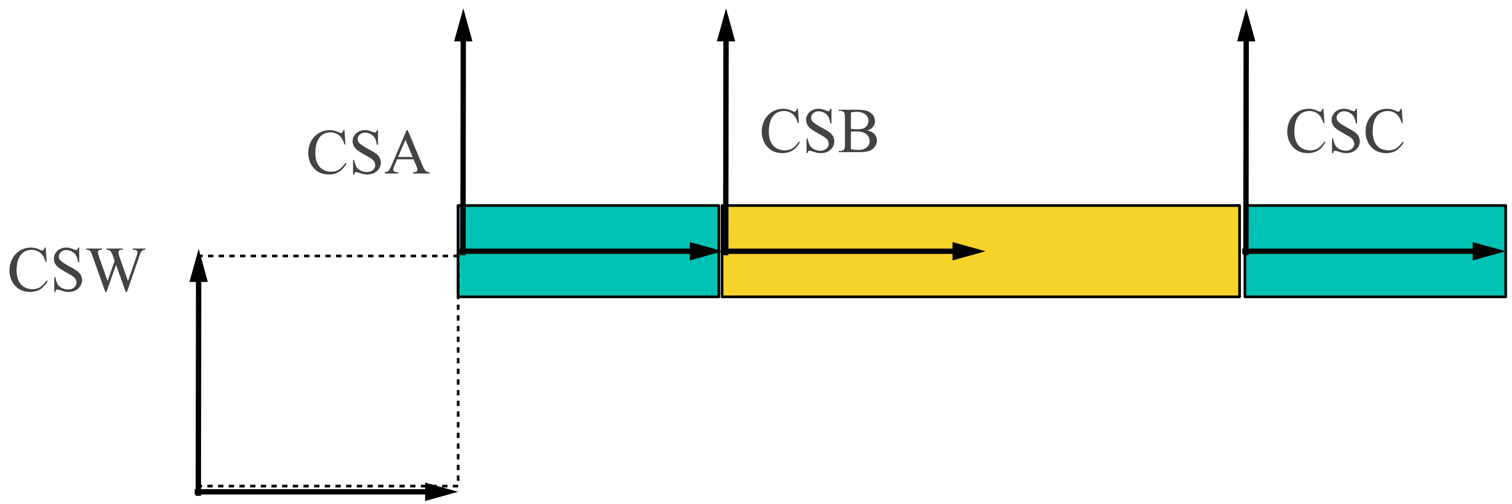


$P_W = {}_W M_A {}_A M_B {}_B M_C P_C$

What do these matrices look like?

Depends on the structure and the degrees of freedom we want

Example: Kinematic Chain

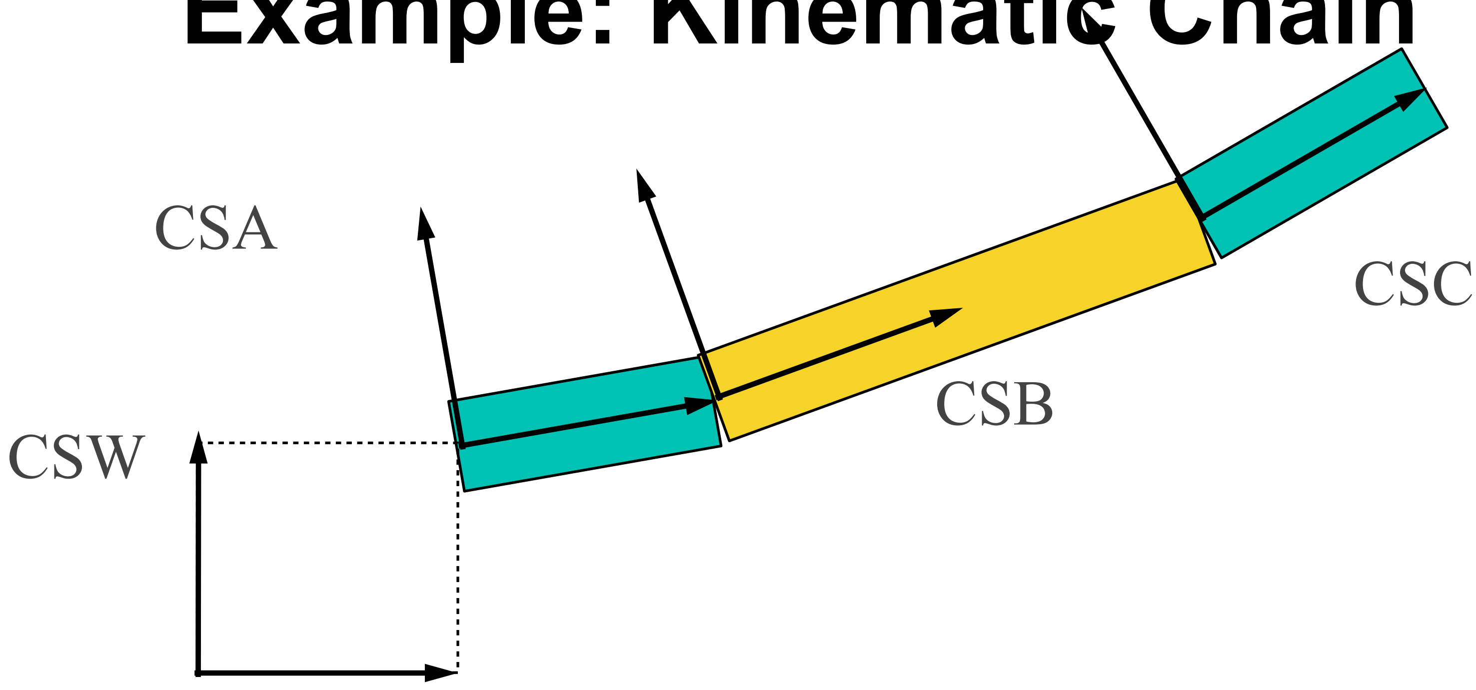


$P_w = {}_wM_A\,{}_AM_B\,{}_BM_C\,P_C$

One possibility *(three degrees of freedom)*

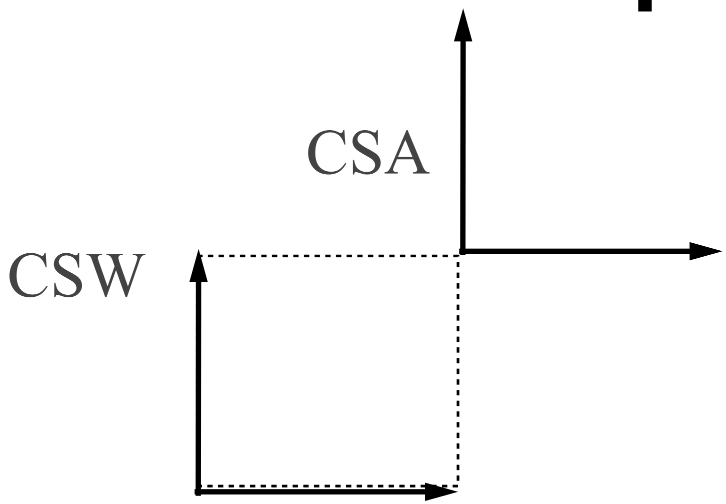
$$P_w(\theta_1, \theta_2, \theta_3) = (T(1,1)R(z,\theta_1))(T(1,0)R(z,\theta_2))(T(2,0)R(z,\theta_3))P_c$$

Example: Kinematic Chain



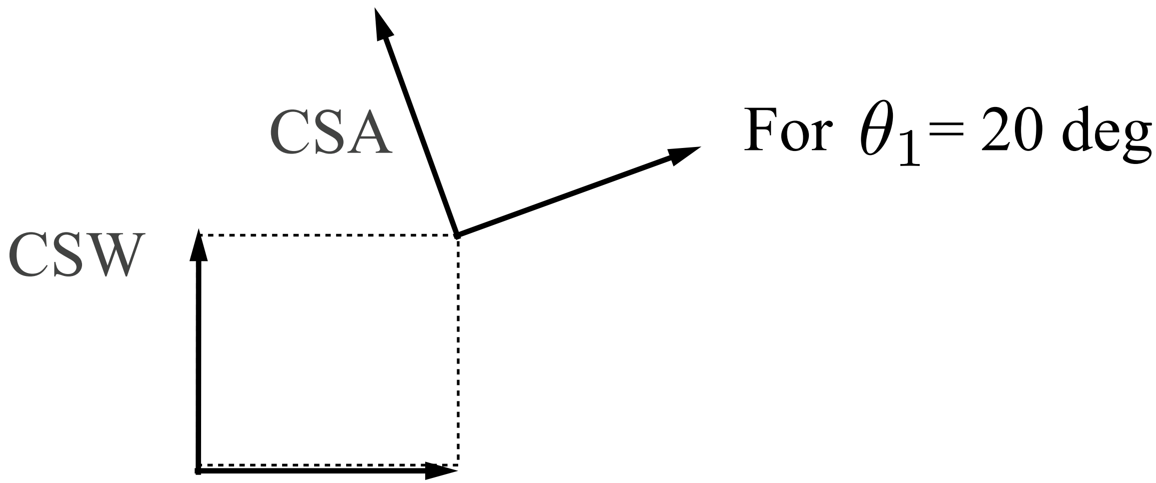
$$P_w(10,10,10) = (T(1,1)R(z,10))(T(1,0)R(z,10))(T(2,0)R(z,10))P_c$$

How is each part drawn in code?



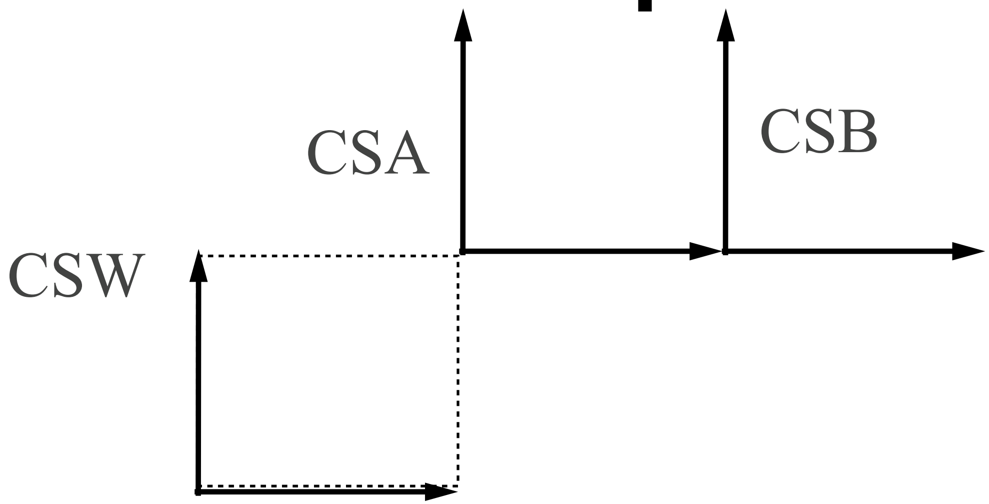
$T(1,1)$	$ModelMat = T(1,1)$
$R(z,\theta_1)$	$ModelMat = T(1,1)R(z,\theta_1)$
$drawCube()$	Cube size 1 centered at the origin

How is each part drawn in code?



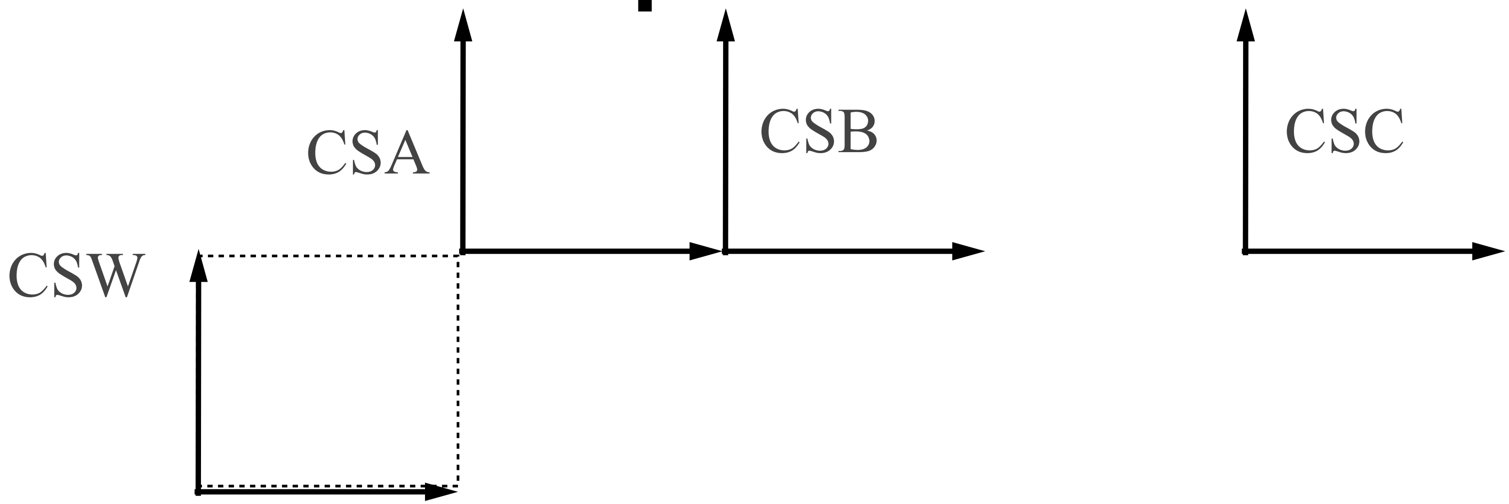
$T(1,1)$	$ModelMat = T(1,1)$
$R(z,\theta_1)$	$ModelMat = T(1,1)R(z,\theta_1)$
$drawCube()$	Cube size 1 centered at the origin

How is each part drawn in code?



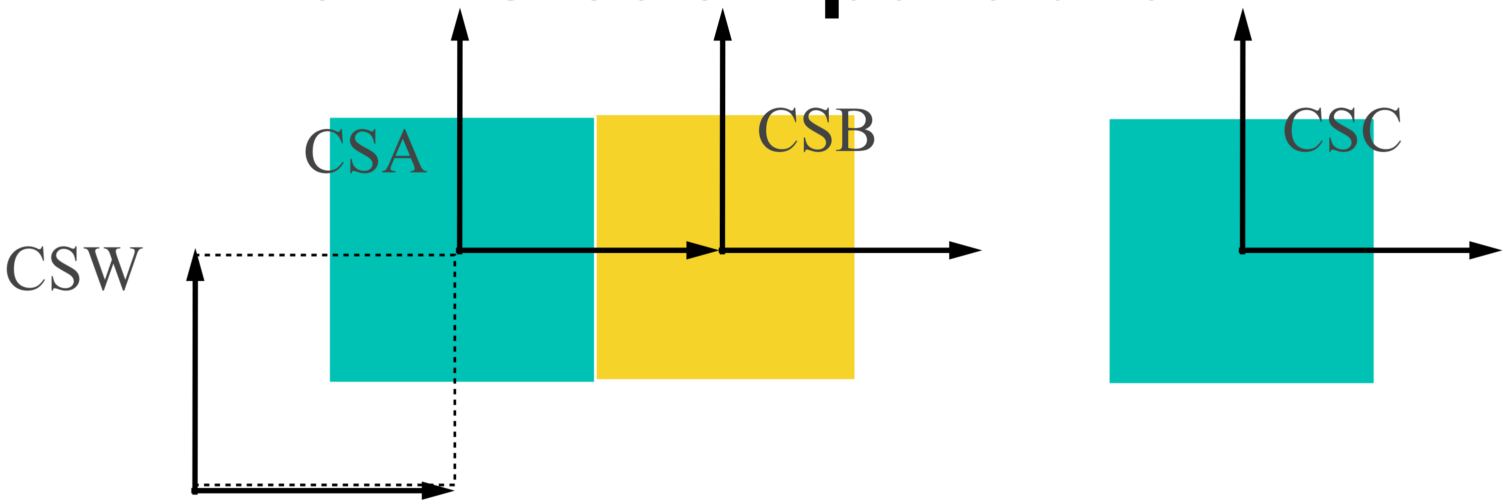
$T(1,1)$	$ModelMat = T(1,1)$
$R(z,\theta_1)$	$ModelMat = T(1,1)R(z,\theta_1)$
$drawCube()$	Cube size 1 centered at the origin
$T(1,0)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)$
$R(z,\theta_2)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)$
$drawCube()$	

How is each part drawn in code?



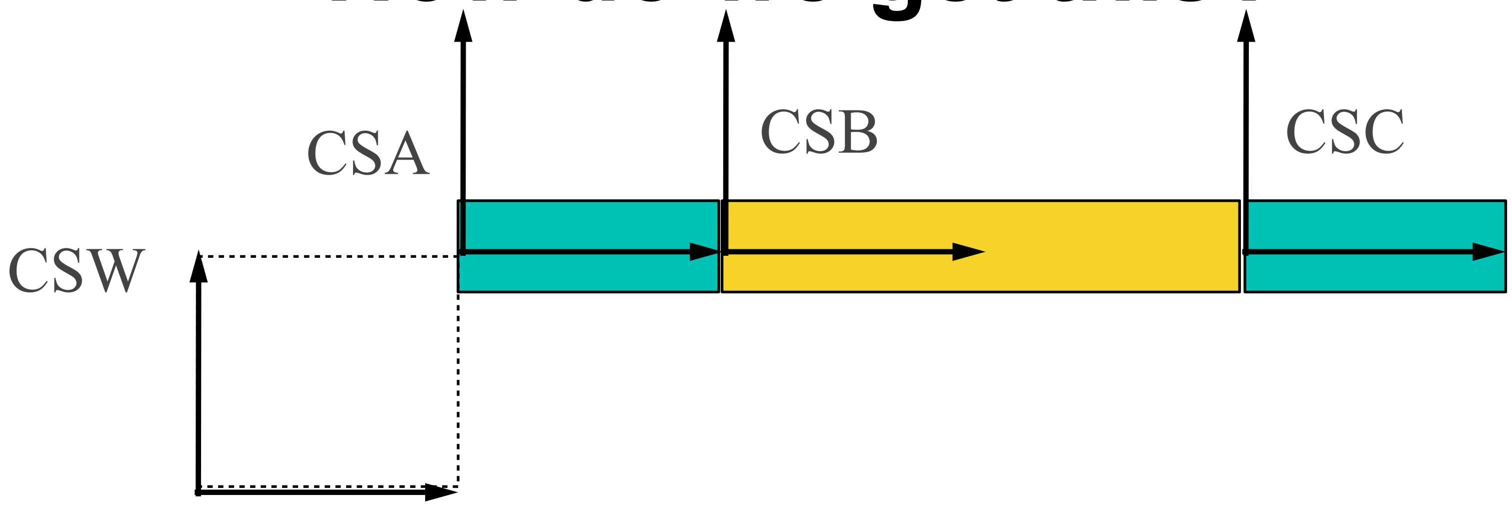
$T(1,1)$	$ModelMat = T(1,1)$
$R(z,\theta_1)$	$ModelMat = T(1,1)R(z,\theta_1)$
$drawCube()$	Cube size 1 centered at the origin
$T(1,0)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)$
$R(z,\theta_2)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)$
$drawCube()$	
$T(2,0)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)T(2,0)$
$R(z,\theta_3)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)T(2,0)Rz(\theta_3)$
$dawCube()$	

How is each part drawn?

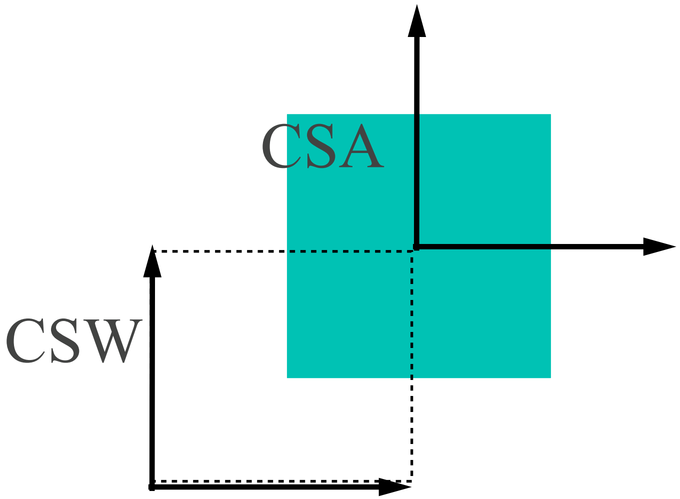


$T(1,1)$	$ModelMat = T(1,1)$
$R(z,\theta_1)$	$ModelMat = T(1,1)R(z,\theta_1)$
$drawCube()$	Cube size 1 centered at the origin
$T(1,0)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)$
$R(z,\theta_2)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)$
$drawCube()$	
$T(2,0)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)T(2,0)$
$R(z,\theta_3)$	$ModelMat = T(1,1)R(z,\theta_1)T(1,0)R(z,\theta_2)T(2,0)Rz(\theta_3)$
$dawCube()$	

How do we get this?



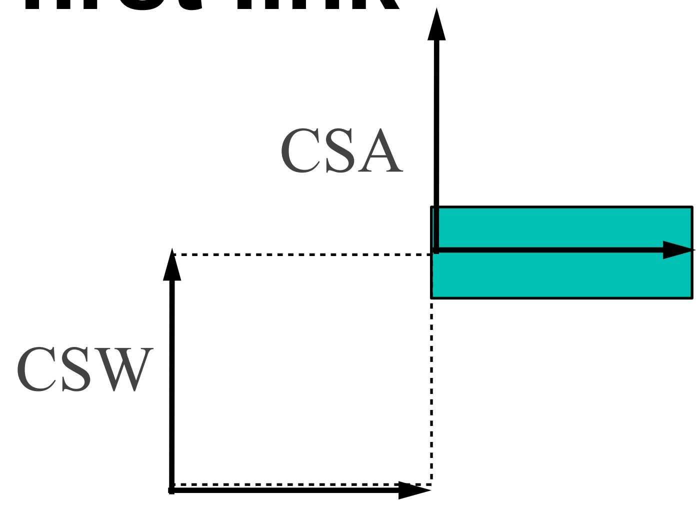
Let's look at the first link



$T(1,1) \qquad M = T(1,1)$

$R(z,\theta_1) \qquad M = T(1,1)R(z,\theta_1)$

$drawCube()$ Cube size 1 centered at the origin



$T(1,1) \qquad M = T(1,1)$

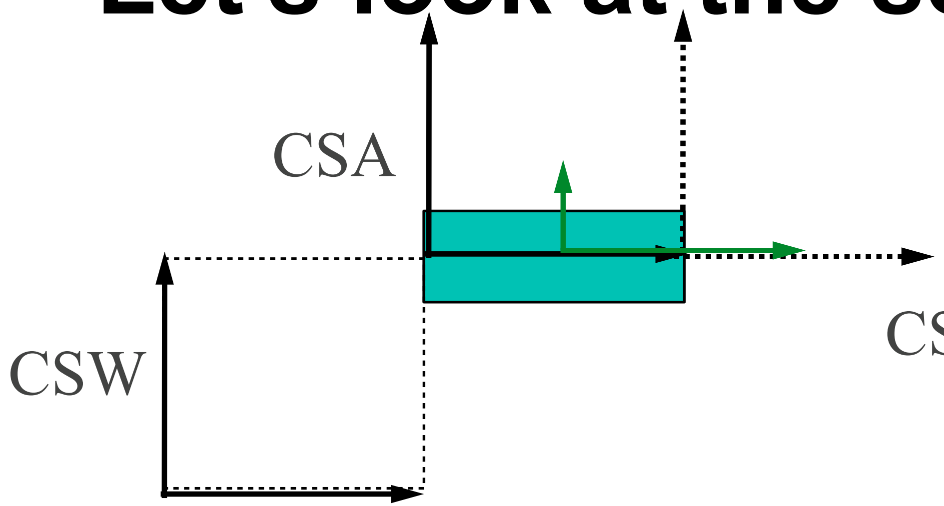
$R(z,\theta_1) \qquad M = T(1,1)R(z,\theta_1)$

$T(0.5,0) \qquad M = T(1,1)R(z,\theta_1)T(0.5,0)$

$S(1,0.3) \qquad M = T(1,1)R(z,\theta_1)T(0.5,0)S(1,0.3)$

$drawCube()$ Cube size 1 centered at the origin

Let's look at the second link



This is a problem now!
It should only apply to the first cube,
not the second system. I could undo the transformations.
However,

$T(1,1)$ $\quad M = T(1,1)$

$R(z,\theta_1)$ $\quad M = T(1,1)R(z,\theta_1)$

$T(0.5,0)$ $\quad M = T(1,1)R(z,\theta_1)T(0.5,0)$

$S(1,0.3)$ $\quad M = T(1,1)R(z,\theta_1)T(0.5,0)S(1,0.3)$

$drawCube()$ Cube size 1 centered at the origin

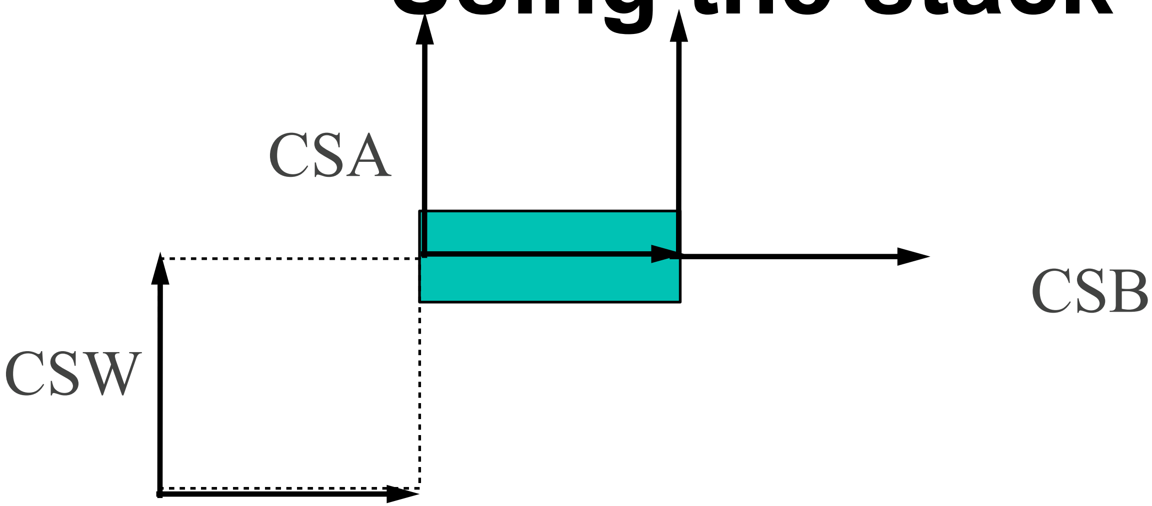
Matrix Stack

When we have more than one branch or multiple objects, it is often convenient to use a matrix stack to load and unload matrices

- Using the push(*) and pop() operations of a stack

- stack.push(M);   // in the code gPush()
- draw();
- stack.pop();   // in the code gPop()

Using the stack



$stack.init()$ $[\,]$
$T(1,1)$ $[M = T(1,1)]$
$R(z,\theta_1)$ $[M = T(1,1)R(z,\theta_1)]$
$stack.push()$ $[M, M]$
$T(0.5,0)$ $[M = T(1,1)R(z,\theta_1), M' = MT(0.5,0)]$
$S(1,0.3)$ $[M = T(1,1)R(z,\theta_1), M' = MT(0.5,0)S(1,0.3)]$
$drawCube()$ Cube size 1 centered at the origin # also sets the modeling matrix # in the shaders to the top of the # stack
$stack.pop()$ $[M = T(1,1)R(z,\theta_1)]$

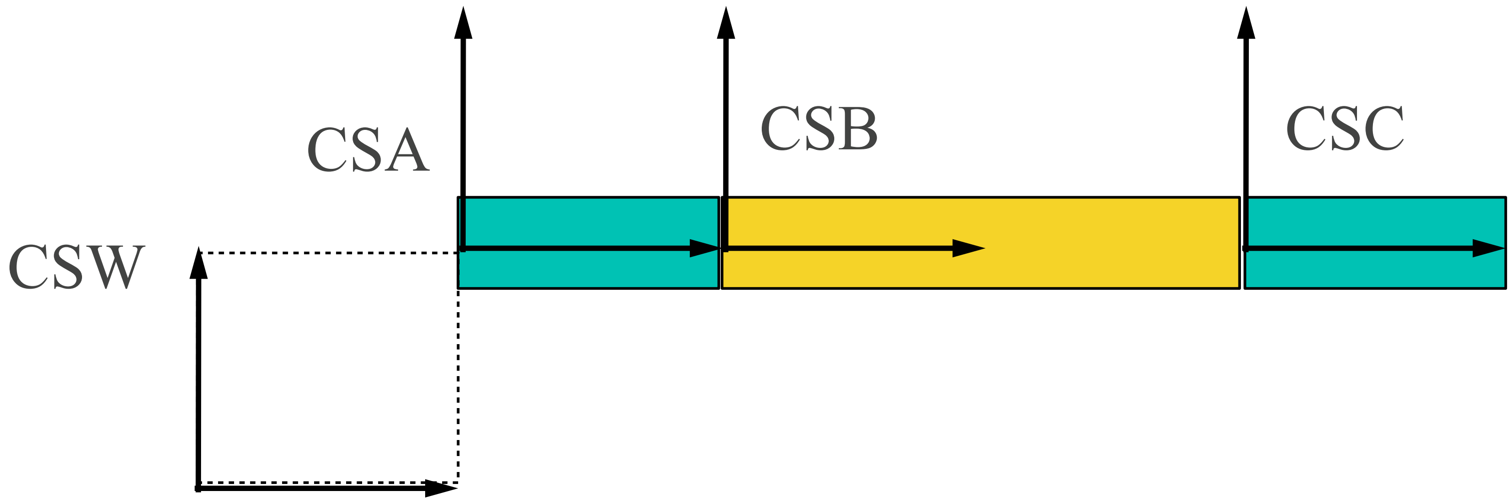
Hybrid way of thinking

Use TOP to BOTTOM to position a coordinate system

Then use BOTTOM to TOP to position the objects within that system

Often it is easier to do it in the opposite order

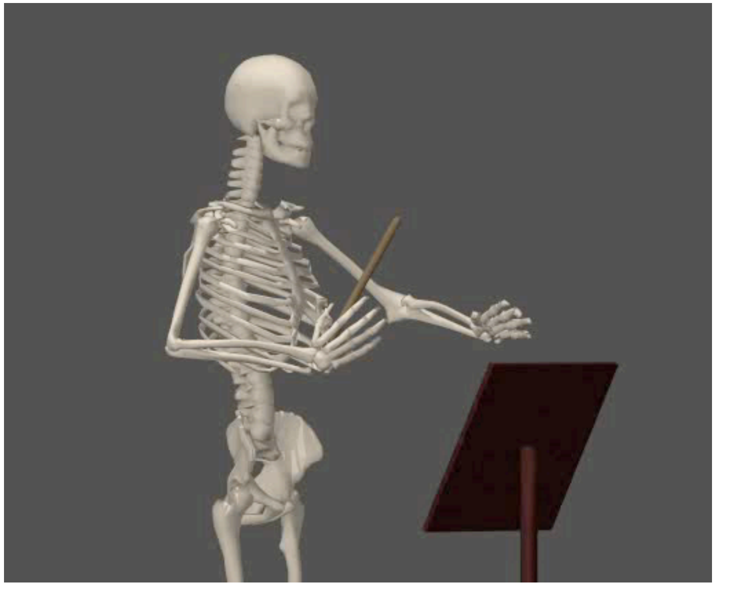
Exercise



Write the complete pseudo code using the stack

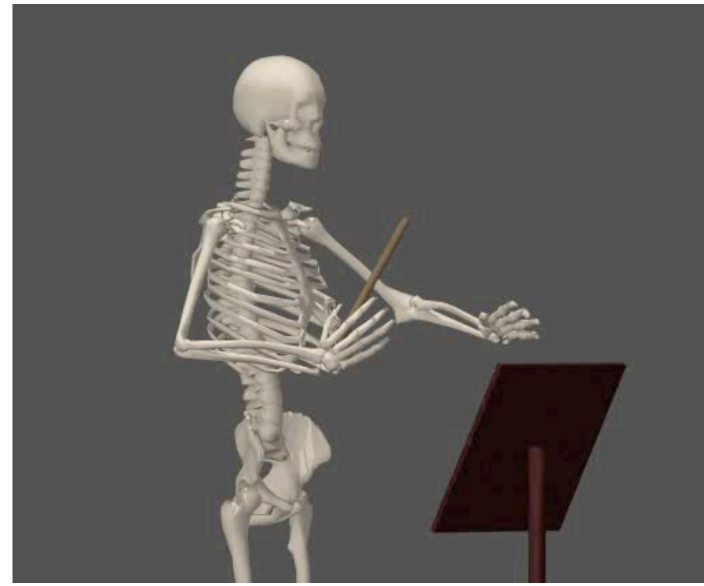
Complex Articulated structures

```
Class Link : Object {
   mat4 _LocalToParent;
   mat4 _ParentToLocal;
   mat4 _WorldToLocal;
   mat4 _LocalToWorld;

   void ComputeLW();
   void ComputeWL();
   void Draw() ;

   Geometry *_geom;
   int _numChildren;
   Link* _children[];
   Link* _parent;
   ….
}
```

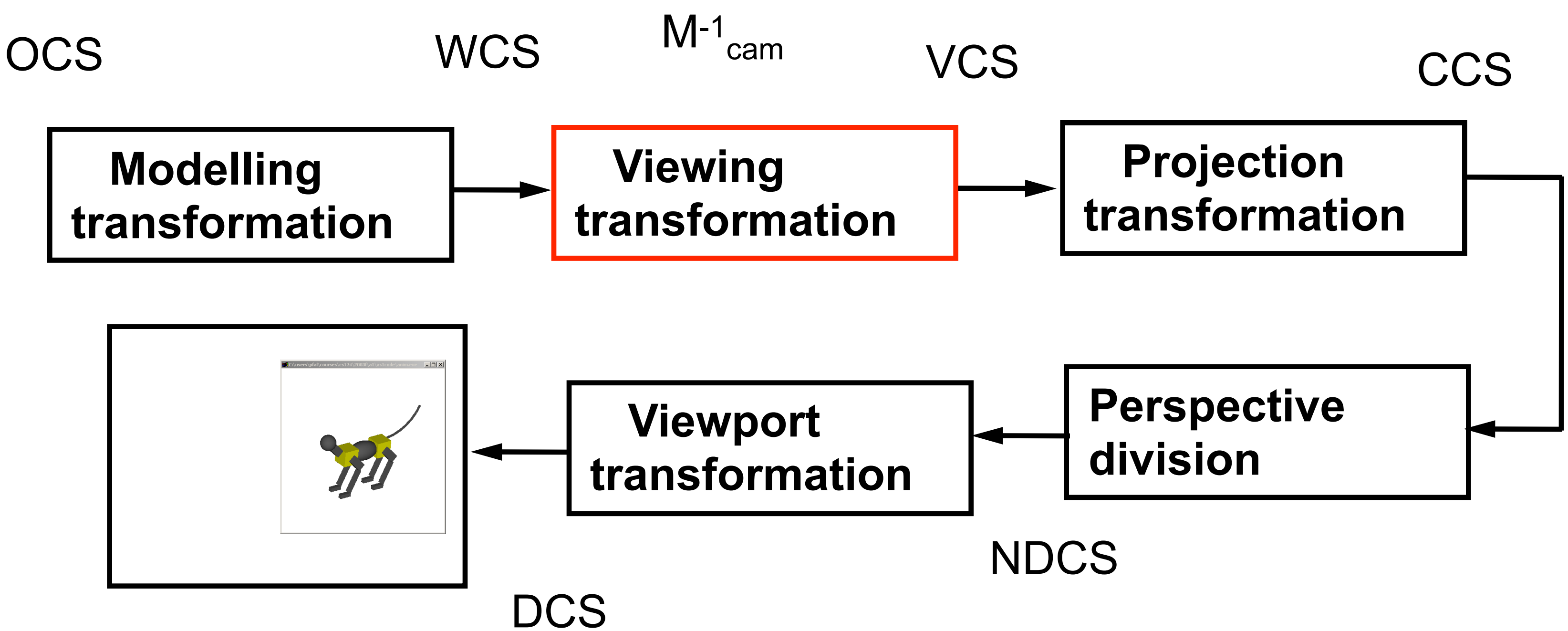


Complex Articulated structures

```
Class Geometry {
   mat4 m;
   Scale(float x, float y, float z);
   Translate(float x, float y, float z);
   Rotate(float theta,vec3 axis) ;
   virtual void Draw() ;
   … (setColor etc)
}

Class Cube(): Geometry {
   void Draw() ;
}
Class Sphere(int n, int m): Geometry {
   void Draw() ;
}
```



Graphics Pipeline



Taking a snapshot of a 3D Scene

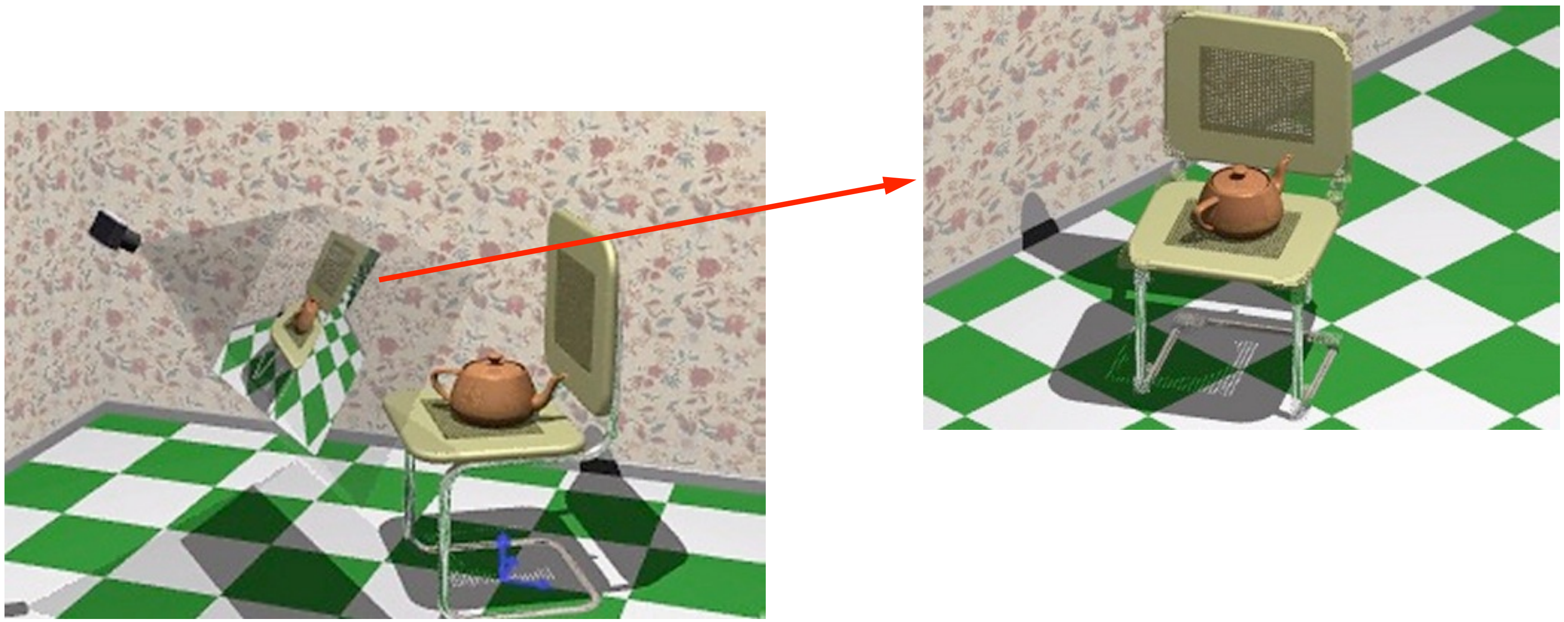
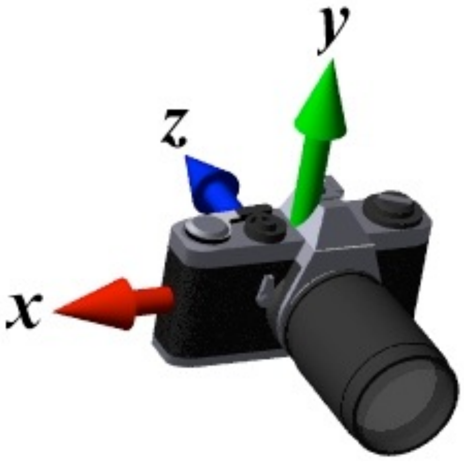


Image copyright E. Angel

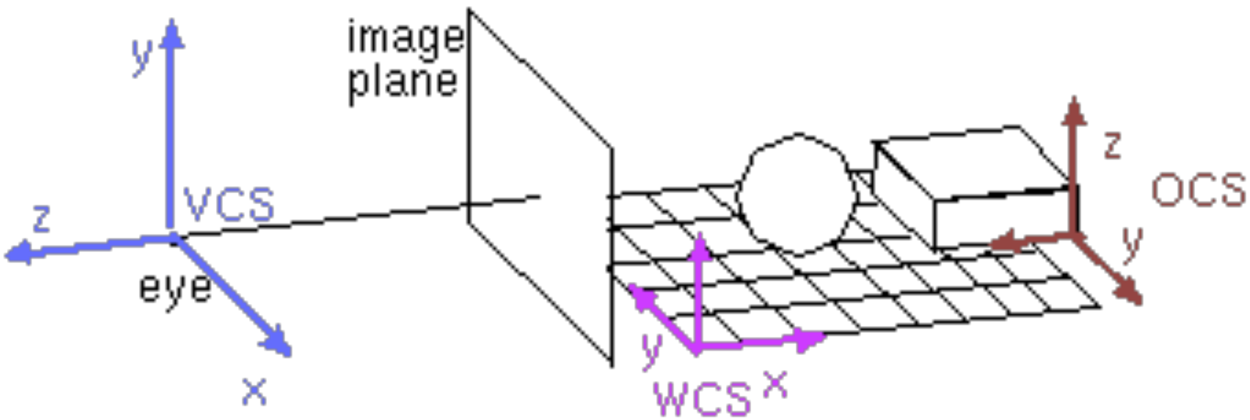
OpenGL Assumption

The camera system is:



Camera transformation

Transforms objects to camera coordinates



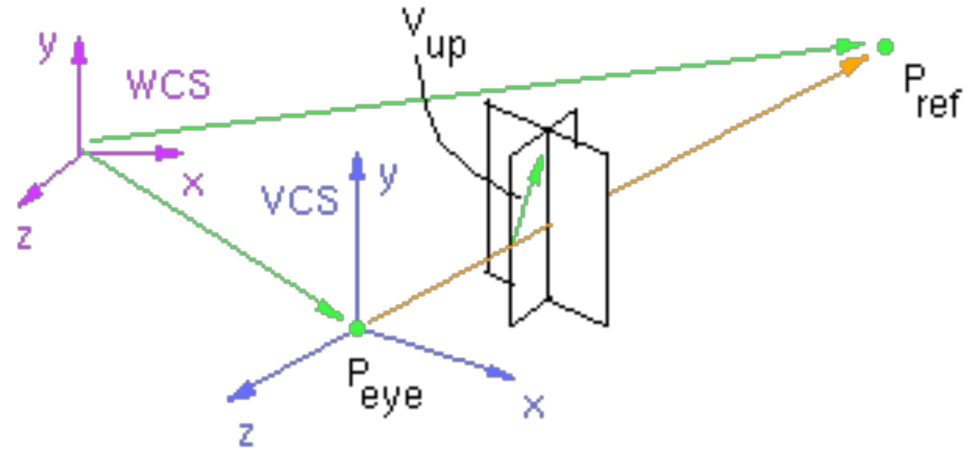
$$\left.\begin{array}{r} P_{wcs} = M_{cam}P_{vcs} \rightarrow P_{vcs} = M_{cam}^{-1}P_{wcs} \\ P_{wcs} = M_{mod}P_{obj} \end{array}\right\} \rightarrow$$

$$P_{vcs} = M_{cam}^{-1}M_{mod}P_{obj}$$

Defining Mcam

Common way

Eye point
Reference point
Upvector



To build Mcam we need to define a camera coordinate system (origin, **i**, **j**, **k**)

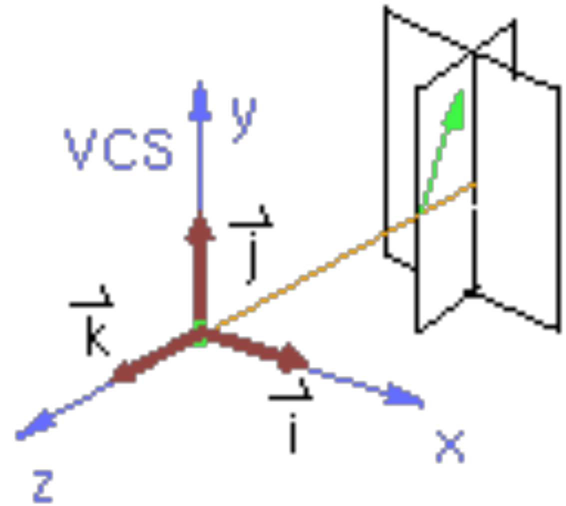
Camera Coordinate system

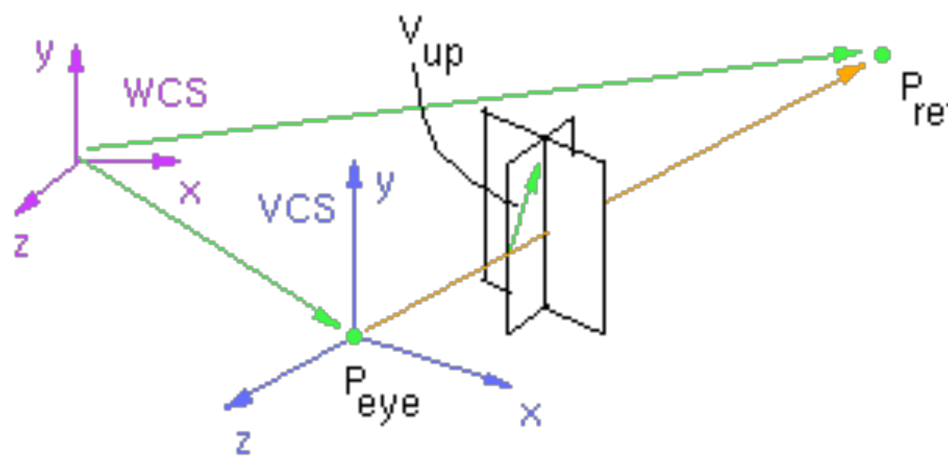
$$\mathbf{k} = \frac{P_{eye} - P_{ref}}{|P_{eye} - P_{ref}|}$$

$$\mathbf{I} = \mathbf{v}_{up} \times \mathbf{k}$$

$$\mathbf{i} = \frac{\mathbf{I}}{|\mathbf{I}|}$$

$$\mathbf{j} = \mathbf{k} \times \mathbf{i}$$

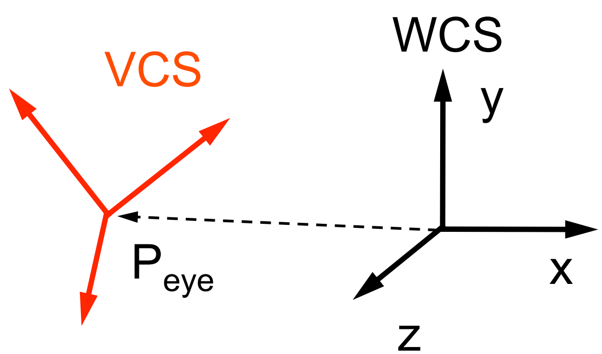




Building Mcam

Change of basis

Our reference system is WCS, we know the camera parameters with respect to the world



Align WCS with VCS

$$M_{cam} = \begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

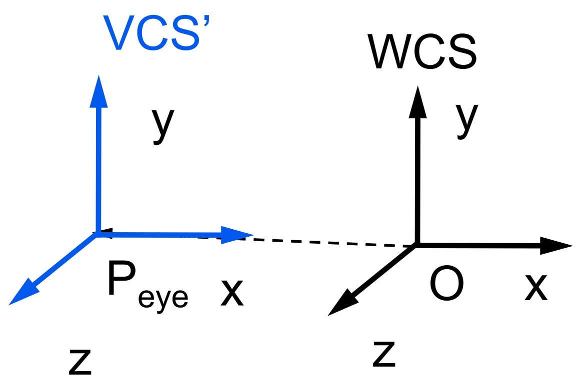
$$P_{wcs} = M_{cam} P_{vcs}$$

In steps…

Building Mcam

First step

Translate WCS to the Peye locations

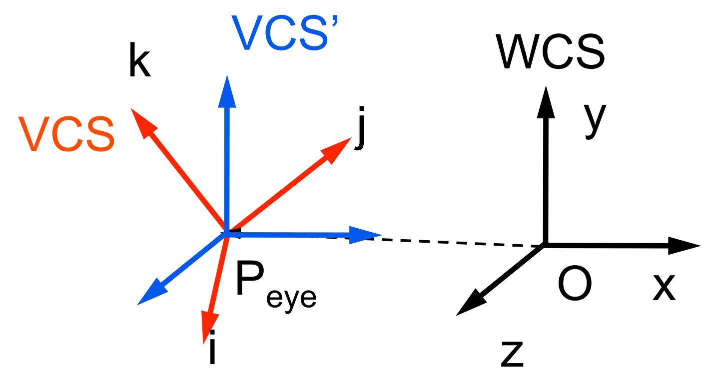


$$M_{cam} = \begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Building Mcam

Second step

Rotate VCS' with respect to itself to create VCS



$$M_{cam} = \begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P_{wcs} = M_{cam} P_{vcs}$$

Building Mcam inverse

Invert smart

$$
\begin{aligned}
M_{cam}^{-1} &= \left(\begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \\
&= \left(\begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \left(\begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1}
\end{aligned}
$$

Building Mcam inverse

Invert smart

$$M_{cam}^{-1} = \left(\begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\right)^{-1} \left(\begin{bmatrix} 1 & 0 & 0 & P_{eye,x} \\ 0 & 1 & 0 & P_{eye,y} \\ 0 & 0 & 1 & P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}\right)^{-1}$$

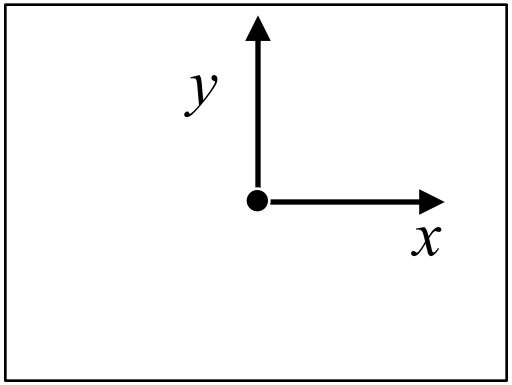
$$= \begin{bmatrix} \overset{\text{Transpose}}{\boxed{\begin{matrix} i_x & i_y & i_z \\ j_x & j_y & j_z \\ k_x & k_y & k_z \end{matrix}}} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & \quad 0 & \quad 0 \end{matrix} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{eye,x} \\ 0 & 1 & 0 & -P_{eye,y} \\ 0 & 0 & 1 & -P_{eye,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P_{vcs} = M_{cam}^{-1} P_{wcs}$$

Camera Transform

Summary

- The camera transformation is really another affine transformation
- It transforms the scene so that the camera is at zero looking down the -z axis



End of Modelling transformations

1. *Preservation of affine combinations of points.*
2. *Preservation of lines and planes.*
3. *Preservation of parallelism of lines and planes.*
4. *Relative ratios on a line are preserved*
5. *Affine transformations are composed of elementary ones.*

Camera transformation as a change of basis.

Graphics Pipeline