COMP3421

Scene Trees, Homogenous coordinates, Transformations

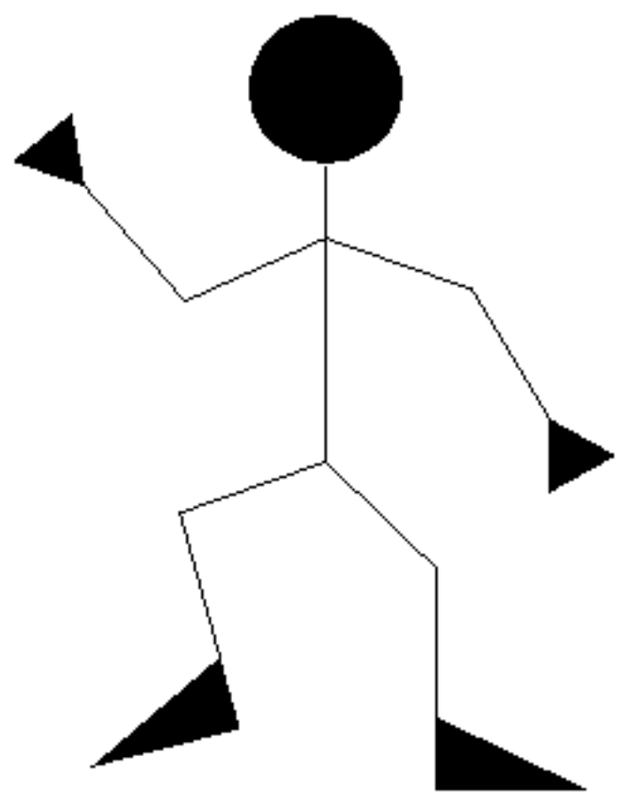
Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

Scene Trees

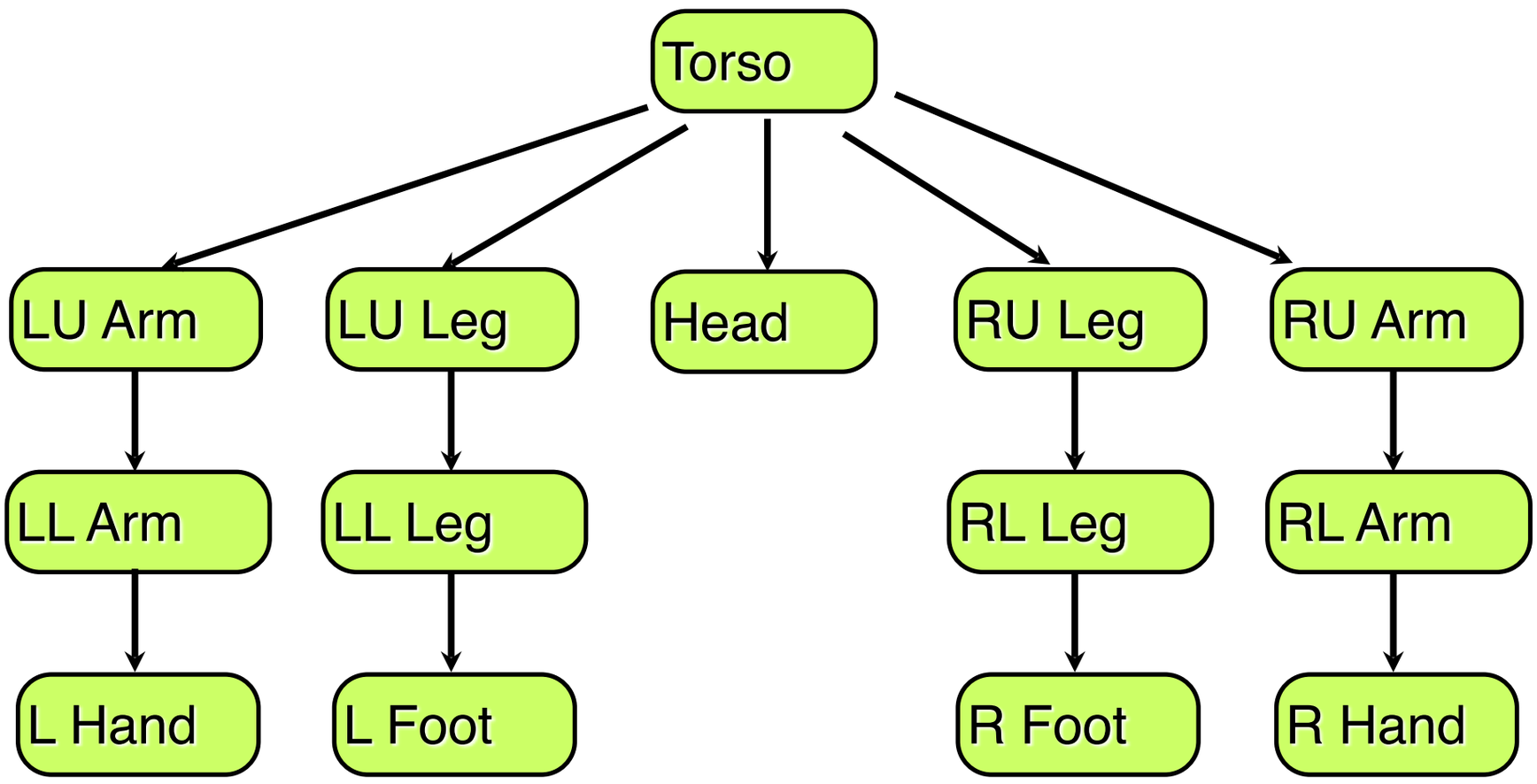
- Consider drawing and animating a figure such as this person:

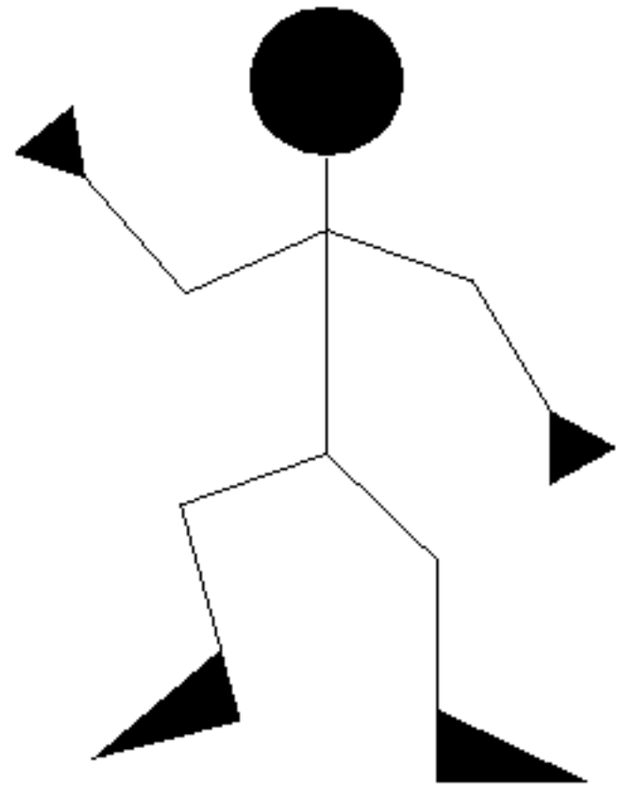
- We could calculate all the vertices based on the angles and lengths, but this would be long and error-prone.



Scene tree

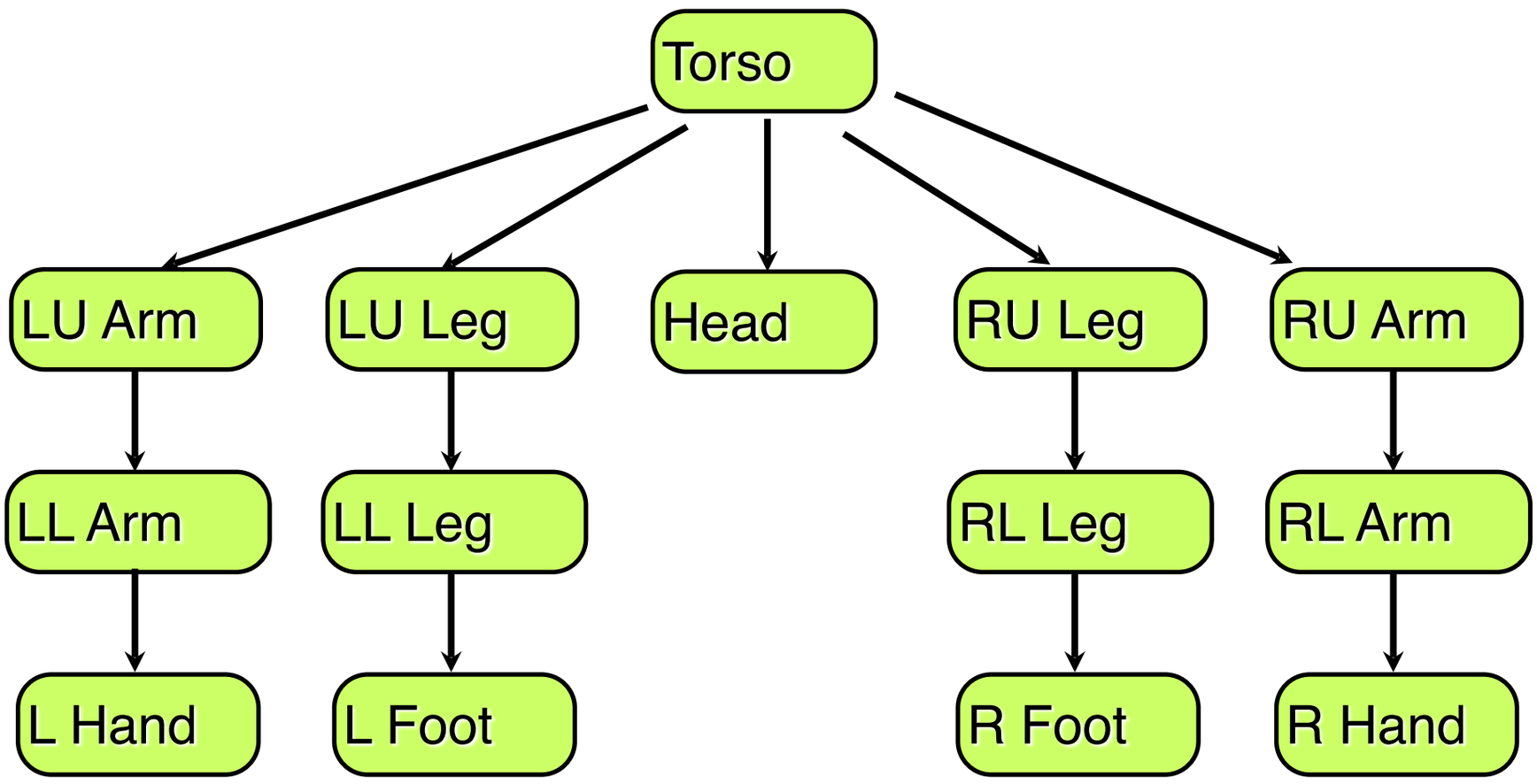
- To represent a complex scene we can use a scene tree. This tree describes how different objects in the scene are connected together:

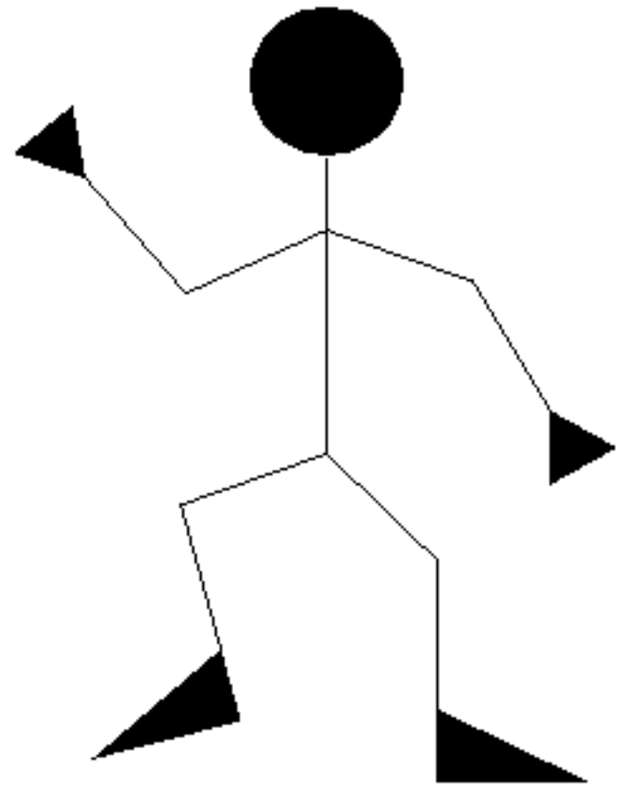




Scene tree

- Each node on the tree represents an object and each edge represents the transformation to get from the parent object's coordinate system to the child's.





Instance Transformation

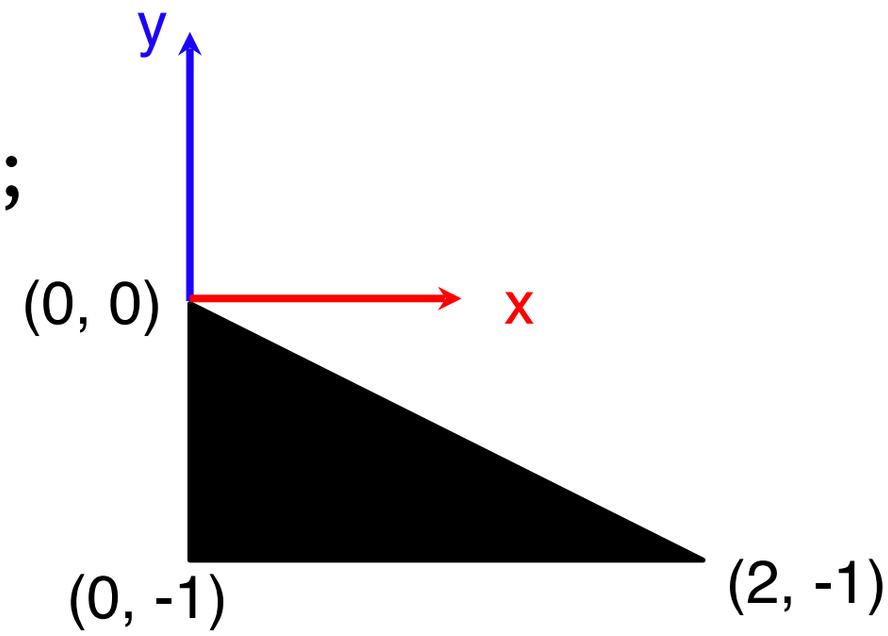
- **Usually** we want: translate(T), rotate(R), scale(S) : M = TRS

- We can specify objects once in a convenient local co-ordinate system

Coordinate system

- We create each part in its own local coordinate system:

```
Triangle2D foot
    = new Triangle2D(0,0, 0,-1, 2,-1);
```



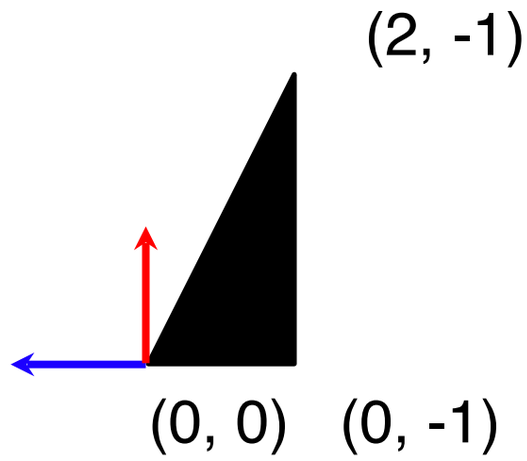
Coordinate system

- Then we transform the coordinate system:

  -translating

  -rotating

  -scaling



- And draw the part in it

Scene tree

- Each part computes its coordinate frame, draws itself within that frame, and passes that frame on to its children.
- When a node in the graph is transformed, all its children move with it.

Scene tree pseudocode

```
drawTree(frame) {
    compute new_frame by transforming frame:
        translation
        rotation
        scale

    draw this object

    for all children:
        child.drawTree(new_frame)
}
```

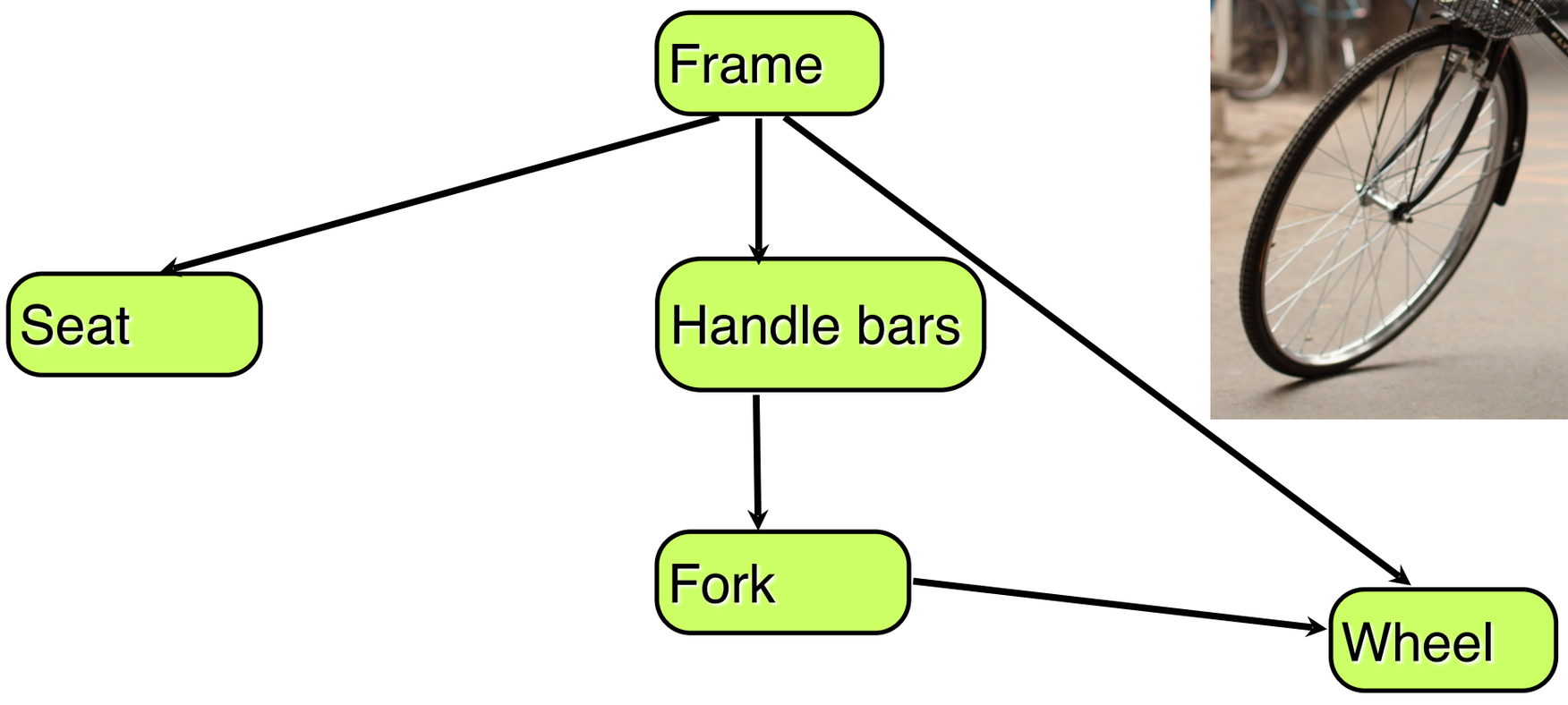
Scene graph

- We can generalise the concept of a scene tree to a scene graph

- A scene graph is a directed acyclic multi-graph
  - Each node can have multiple parents
  - Multiple edges can go from parent to child
  - Shared nodes are drawn multiple times

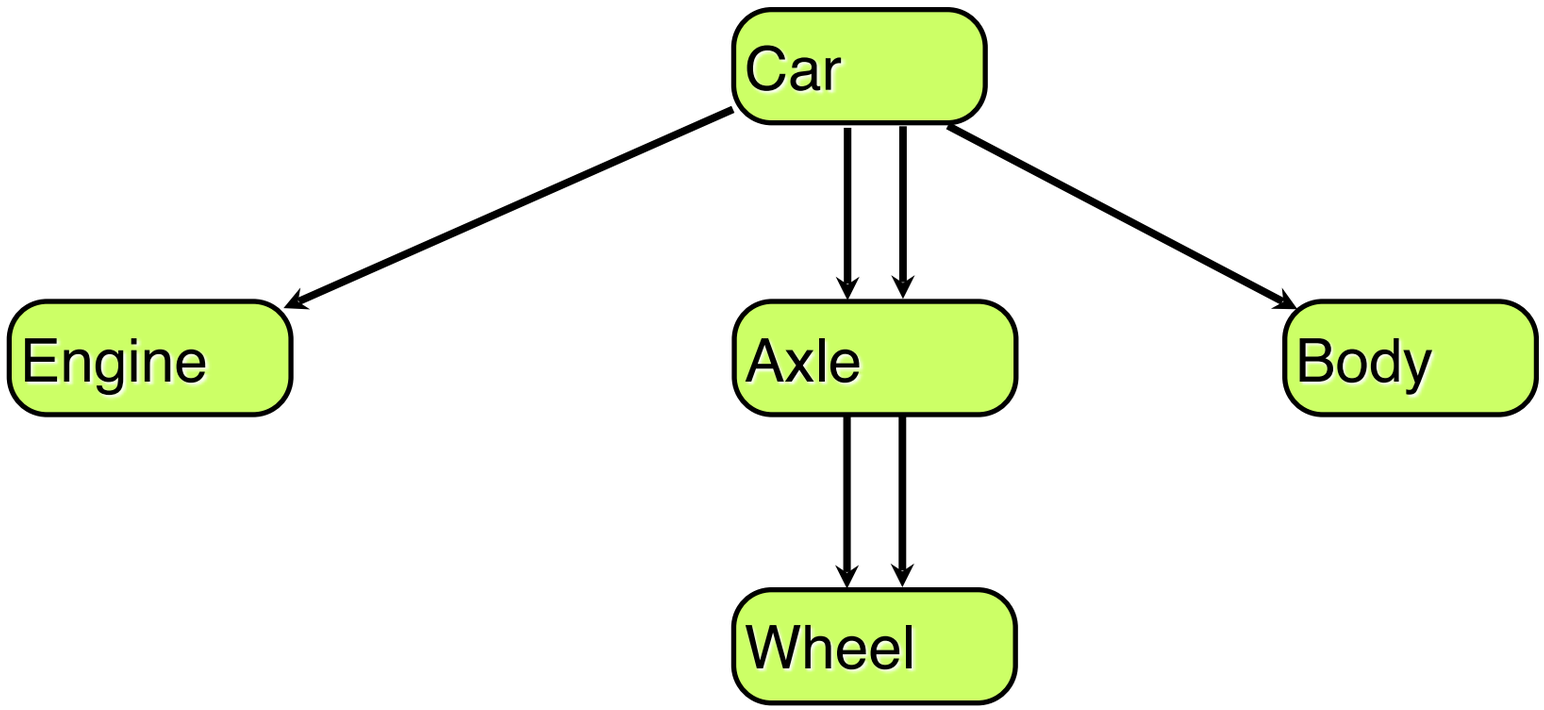
Scene graph

- Can be used to model some things (e.g. a bicycle)



Scene graph

- This only works if the car isn't animated (i.e. the front wheels don't rotate relative to the axle)



Camera

- So far we have assumed that the camera is positioned at the world coordinate (0, 0).

- It is useful to imagine the camera as an object itself, with its own position, rotation and scale.

View transform

- The world is rendered as it appears in the camera's local coordinate frame.

- The view transform converts the world coordinate frame into the camera's local coordinate frame.

- Note that this is the inverse of the transformation that would convert the camera’s local coordinate frame into world coordinates.

View transform

- Consider the world as if it was centered on the camera. The camera stays still and the world moves.
  - Moving the camera left
    = moving the world right
  - Rotating the camera clockwise
    = rotating the world anticlockwise
  - Growing the camera's view
    = shrinking the world

Transformation pipeline

- We transform in 2 stages

$$P_{camera} \xleftarrow{view} P_{world} \xleftarrow{model} P_{local}$$

- The model transform transforms points in the local coordinate system to the world coordinate system

- The view transform transforms points in the world coordinate system to the camera's coordinate system

View transform

- Mathematically if:

$$P_{world} = Trans(Rot(Scale(P_{camera})))$$

- Then the view transform is:

$$P_{camera} = Scale^{-1}(Rot^{-1}(Trans^{-1}(P_{world})))$$

Implementing a camera

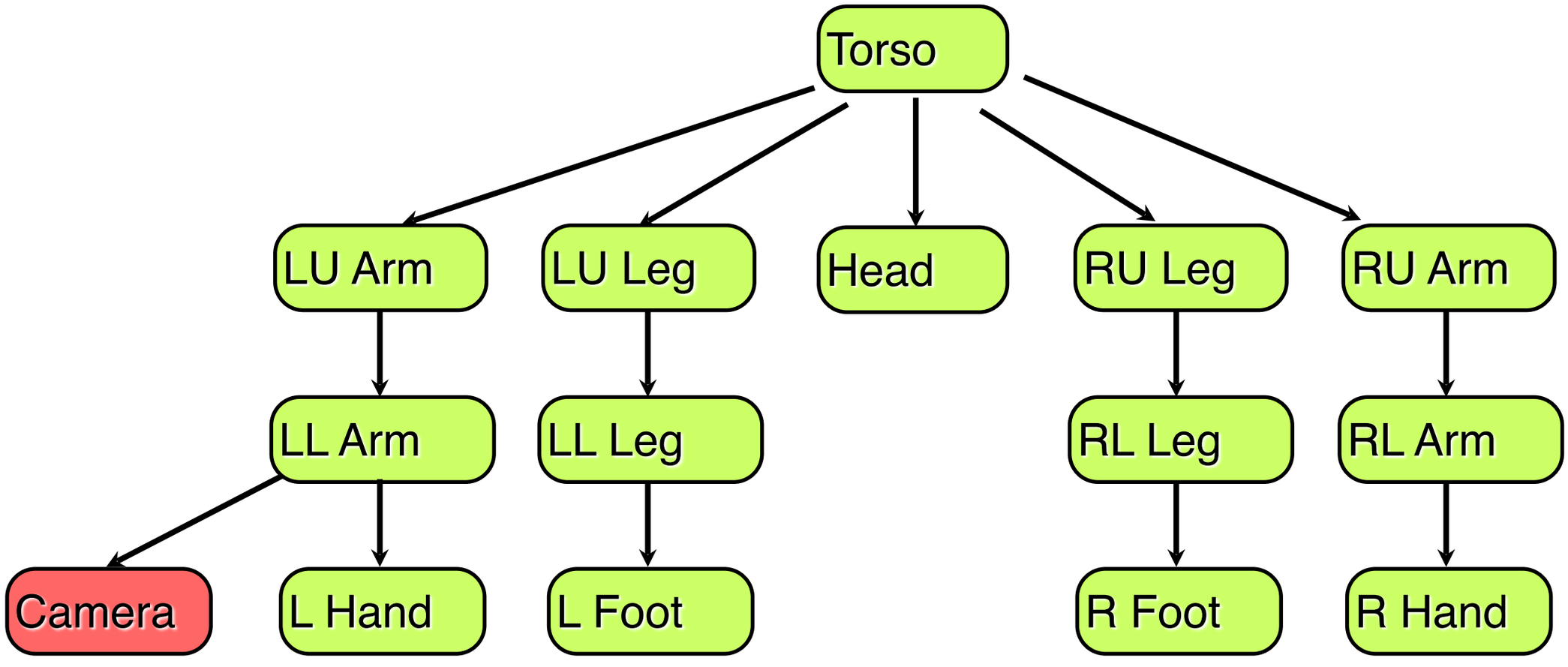
- We can compute the view transform in UNSWgraph like so:

```
CoordFrame2D viewFrame = CoordFrame2D.identity()
        .scale(1/myScale, 1/myScale)
        .rotate(-myAngle)
        .translate(-myPos.getX(), -myPos.getY());
```

- See Camera.java in the person example

In the scene tree

- Can we add the camera as an object in our scene tree?



In the scene tree

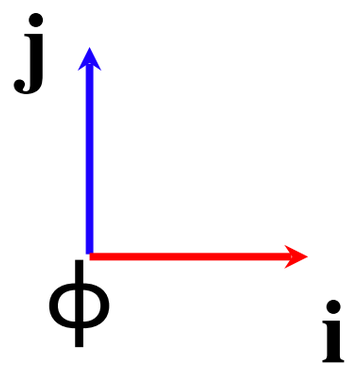
- We need to compute the camera's transformations in world coordinates (and then get the inverse) in order to compute the view transform.

- Your assignment task!

Coordinate frames

- We can now think of a coordinate frame in terms of vectors.

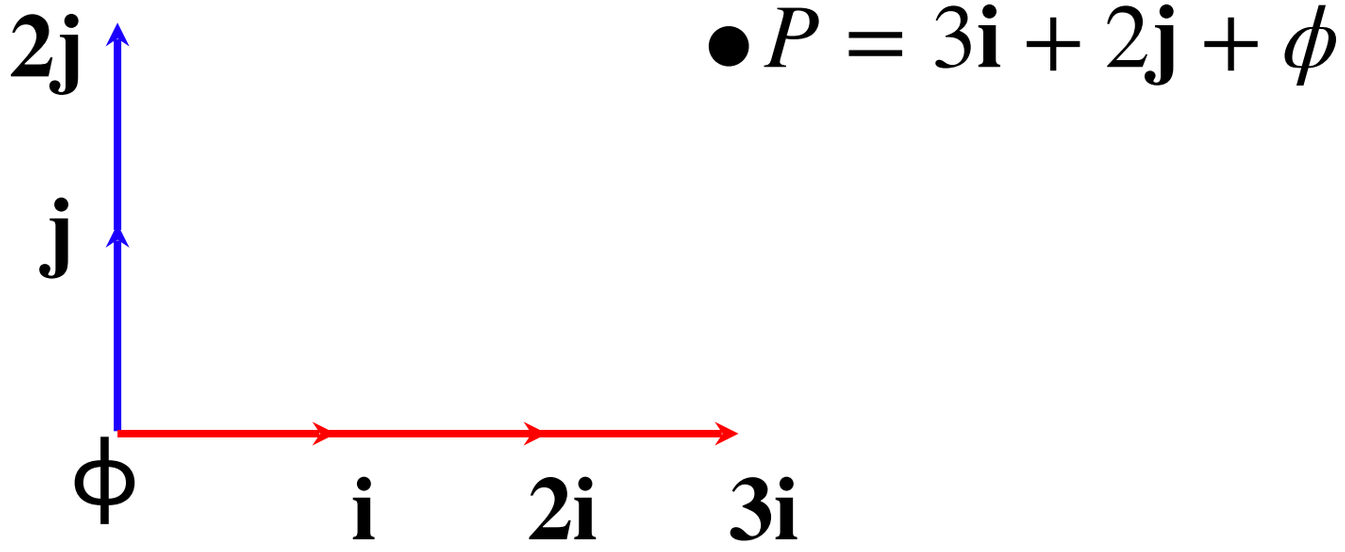
- A 2D frame is defined by:
  - an origin: ϕ
  - 2 axis vectors: $\mathbf{i}, \mathbf{j}$



Points

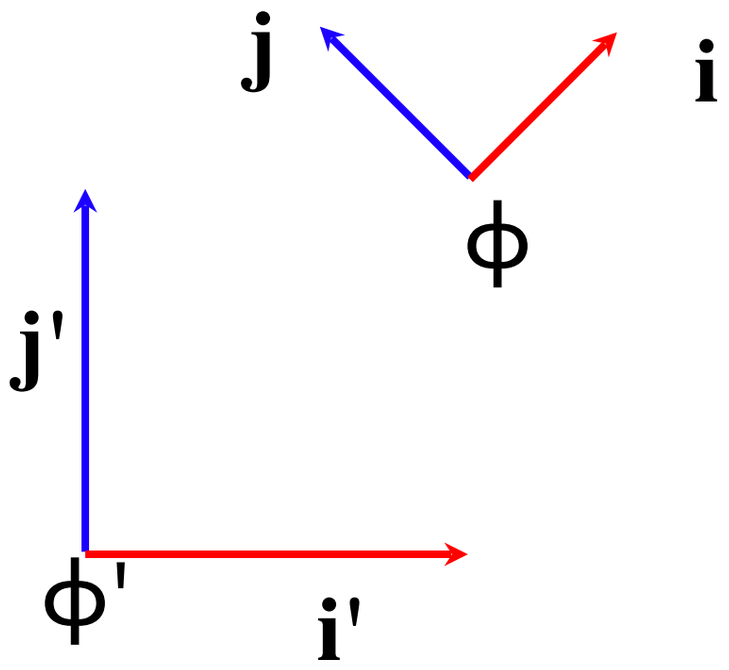
- A point in a coordinate frame can be described as a displacement from the origin:

$$P = p_1\mathbf{i} + p_2\mathbf{j} + \phi$$



Transformation

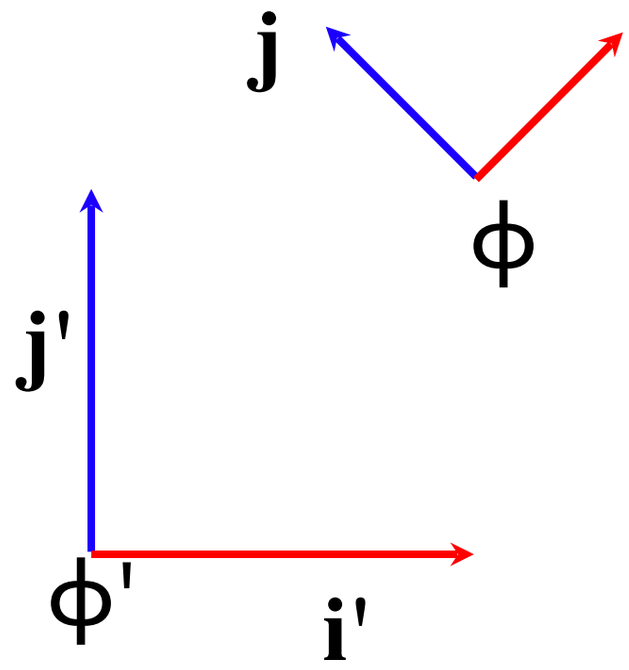
- To convert P to a different coordinate frame, we just need to know how to convert $\mathbf{i}$, $\mathbf{j}$ and ϕ.



$$\begin{aligned}\phi &= 1\mathbf{i}' + 1\mathbf{j}' + \phi' \\ \mathbf{i} &= 0.4\mathbf{i}' + 0.4\mathbf{j}' \\ \mathbf{j} &= -0.4i' + 0.4\mathbf{j}'\end{aligned}$$

Transformation

- To convert P to a different coordinate frame, we just need to know how to convert $\mathbf{i}$, $\mathbf{j}$ and ϕ.



$$
\begin{aligned}
P &= 3\mathbf{i} + 2\mathbf{j} + \phi \\
&= 3(0.4\mathbf{i}' + 0.4\mathbf{j}') + \\
&\quad 2(-0.4\mathbf{i}' + 0.4\mathbf{j}') + \\
&\quad 1\mathbf{i}' + 1\mathbf{j}' + \phi' \\
&= 1.4\mathbf{i}' + 3\mathbf{j}' + \phi'
\end{aligned}
$$

Transformation

- This transformation is much easier to represent as a matrix:

$$
\begin{array}{rcl}
 & & \begin{matrix} \mathbf{i} & \mathbf{j} & \phi \end{matrix} \\
P & = & \begin{pmatrix} 0.4 & -0.4 & 1 \\ 0.4 & 0.4 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \\
 & = & \begin{pmatrix} 1.4 \\ 3 \\ 1 \end{pmatrix}
\end{array}
$$

Homogenous coordinates

- We can use a single notation to describe both points and vectors.

- Homogenous coordinates have an extra dimension representing the origin:

$$P = \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix} \qquad v = \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}$$

Includes Origin

Does not include origin

Points and vectors

- We can add two vectors to get a vector:

$$(u_1, u_2, 0)^\top + (v_1, v_2, 0)^\top = (u_1 + v_1, u_2 + v_2, 0)^\top$$

- We can add a vector to a point to get a new point:

$$(p_1, p_2, 1)^\top + (v_1, v_2, 0)^\top = (p_1 + v_1, p_2 + v_2, 1)^\top$$

- We cannot add two points.

$$(p_1, p_2, 1)^\top + (q_1, q_2, 1)^\top = (p_1 + q_1, p_2 + q_2, \mathbf{2})^\top$$

Affine transformations

- Transformations between coordinate frames can be represented as matrices:

$$Q = \mathbf{M}P$$

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} i_1 & j_1 & \phi_1 \\ i_2 & j_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

- Matrices in this form (note the 0s with the 1 at the end of the bottom row) are called affine transformations .

Affine transformations

- Similarly for vectors:

$$\mathbf{v} = \mathbf{M}\mathbf{u}$$

$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} i_1 & j_1 & \phi_1 \\ i_2 & j_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix}$$

Basic transformations

- All affine transformations can be expressed as combinations of four basic types:
  - Translation
  - Rotation
  - Scale
  - Shear

Affine transformations

- Affine transformations preserve straight lines:

$$\mathbf{M}(A + t\mathbf{v}) = \mathbf{M}A + t\mathbf{M}\mathbf{v}$$

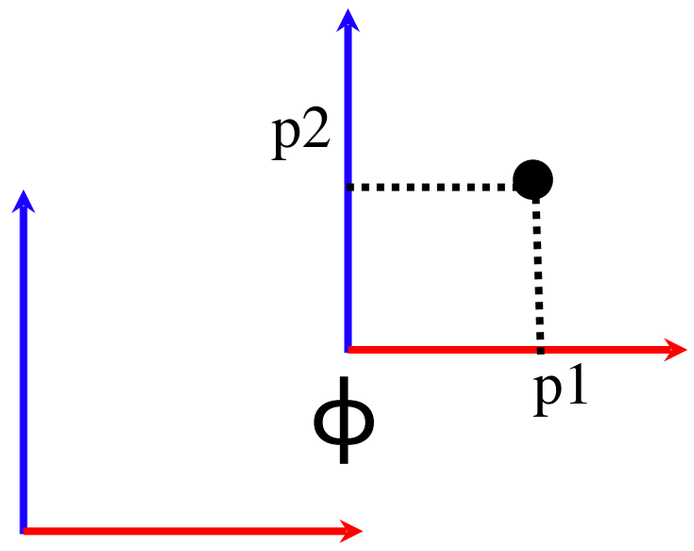
point vector

- They maintain parallel lines
- They maintain relative distances on lines (ie midpoints are still midpoints etc)
- They don’t always preserve angles or area

2D Translation

- To translate the origin to a new point ϕ.

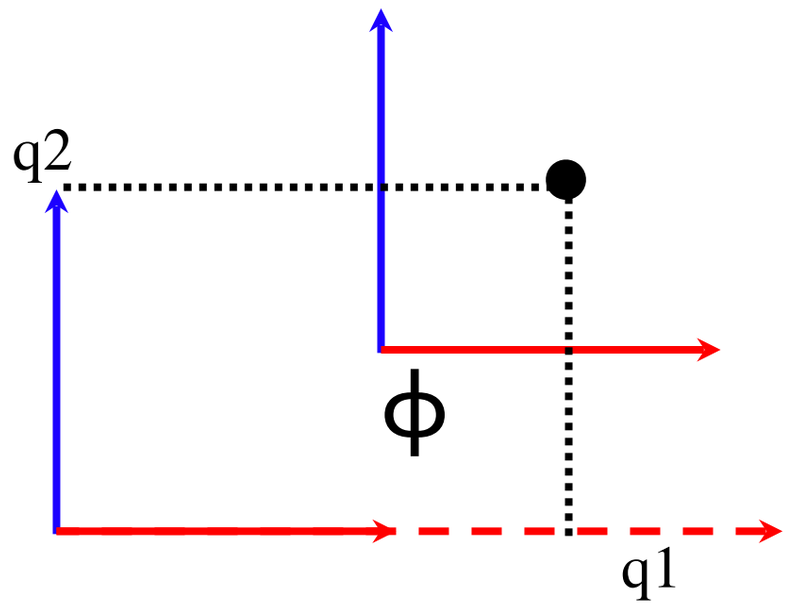
$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \phi_1 \\ 0 & 1 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



2D Translation

- To translate the origin to a new point ϕ.

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \phi_1 \\ 0 & 1 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



Exercise

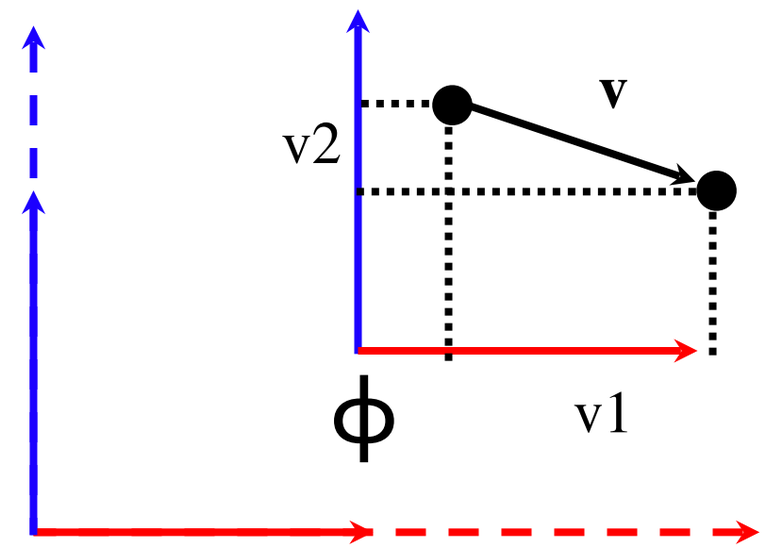
- Translate by (1,0.5) then plot point P = (0.5,0.5) in local frame.

- What is the point in world co-ordinates?

2D Translation

- Note: translating a vector has no effect.

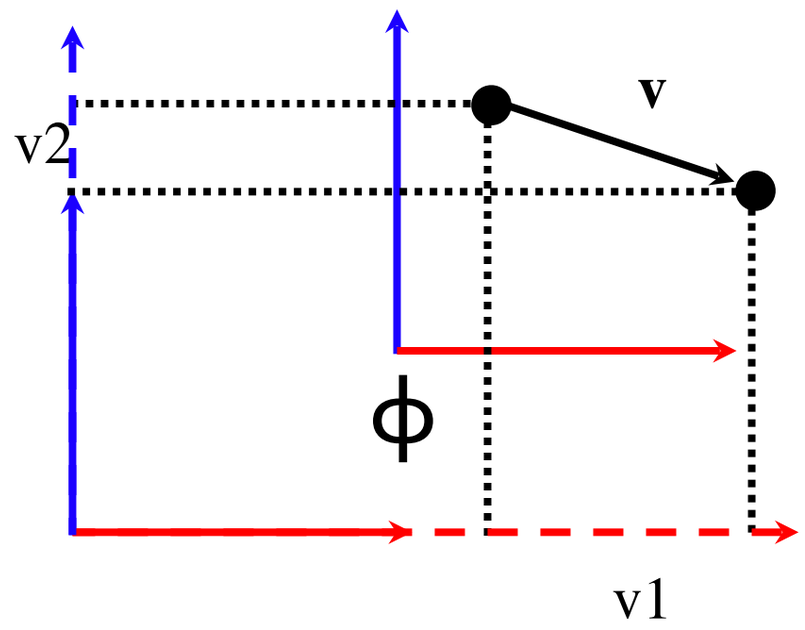
$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \phi_1 \\ 0 & 1 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}$$



2D Translation

- Note: translating a vector has no effect.

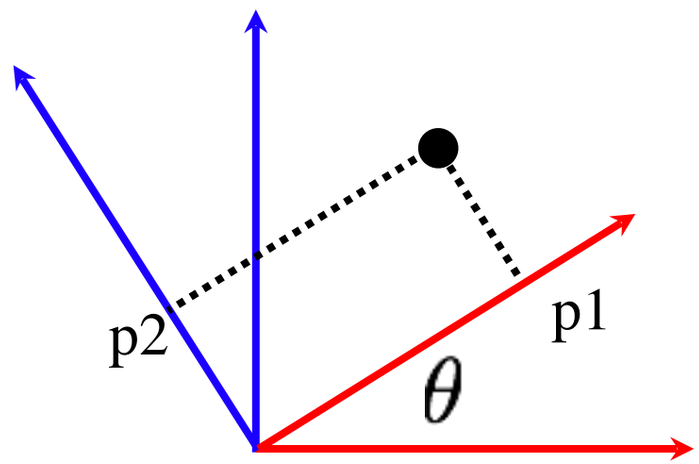
$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \phi_1 \\ 0 & 1 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}$$



2D Rotation

- To rotate a point about the origin:

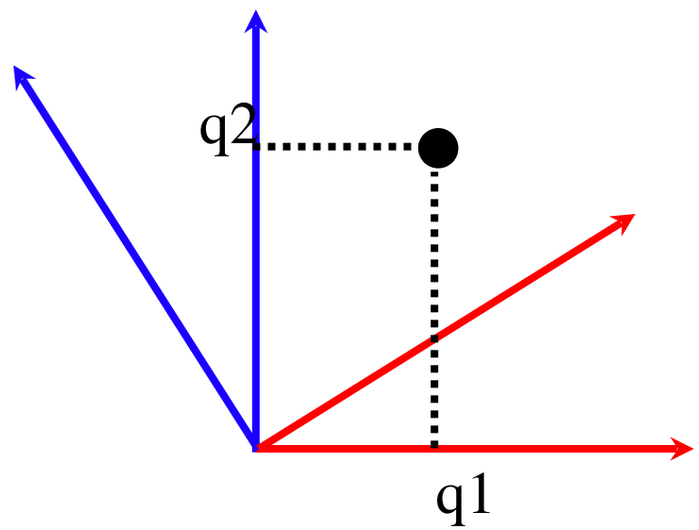
$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



2D Rotation

- To rotate a point about the origin:

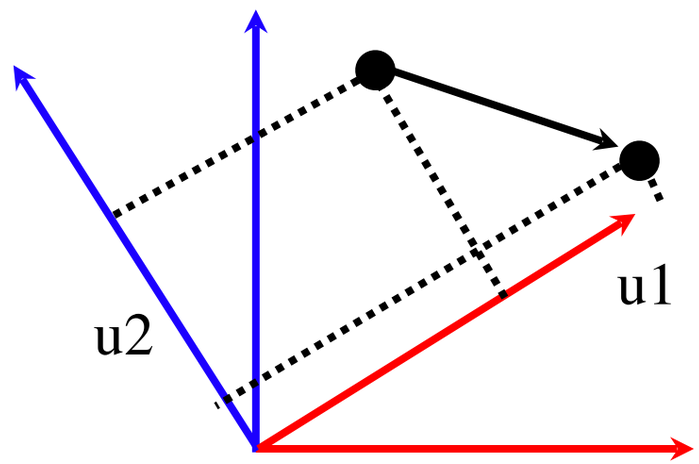
$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



2D Rotation

- Likewise to rotate a vector:

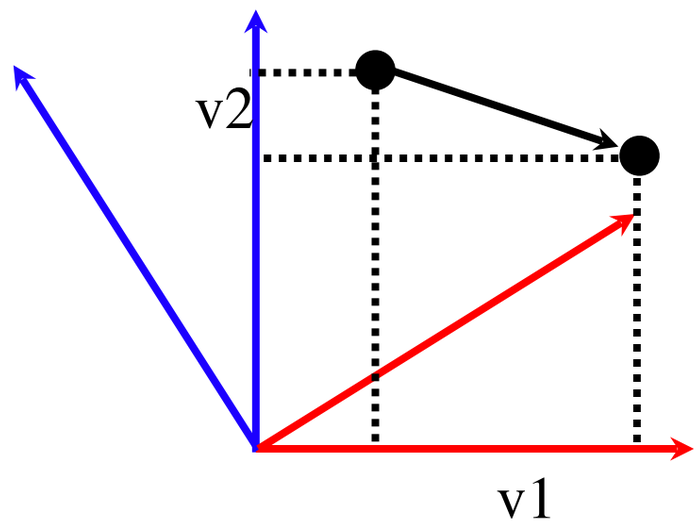
$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix}$$



2D Rotation

- Likewise to rotate a vector:

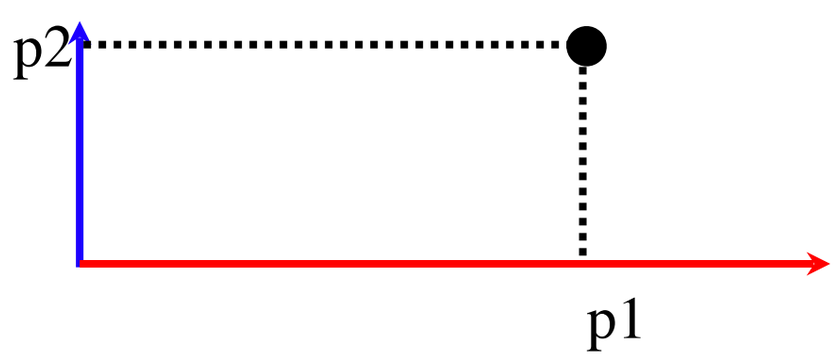
$$\begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix}$$



2D Scale

- To scale a point by factors (sx, sy) about the origin:

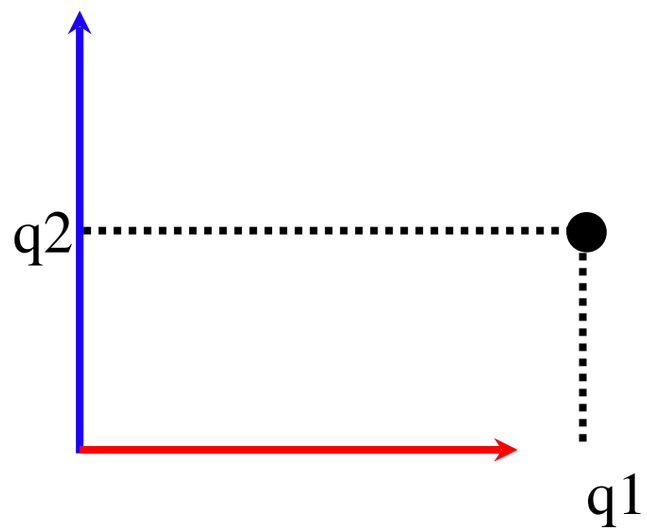
$$\begin{pmatrix} s_x p_1 \\ s_y p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



2D Scale

- To scale a point by factors (sx, sy) about the origin:

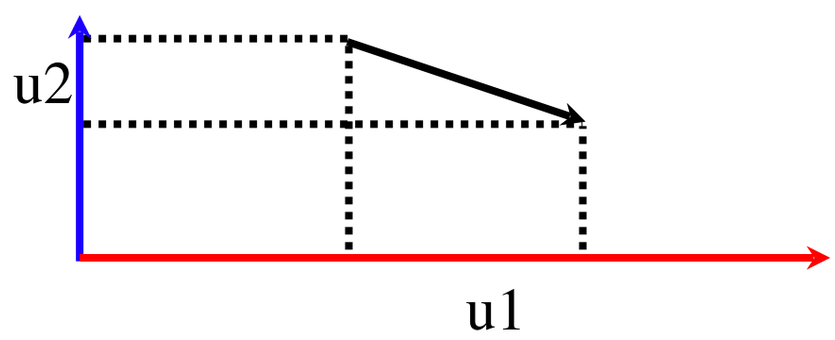
$$\begin{pmatrix} s_x p_1 \\ s_y p_2 \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$



2D Scale

- Likewise to scale vectors:

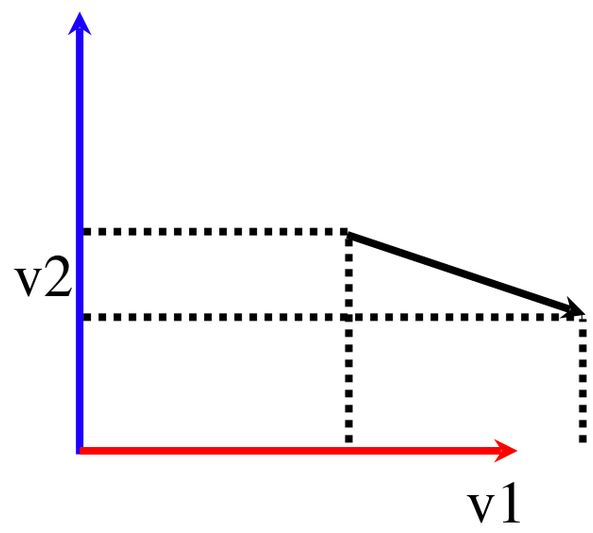
$$\begin{pmatrix} s_x v_1 \\ s_y v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}$$



2D Scale

- Likewise to scale vectors:

$$\begin{pmatrix} s_x v_1 \\ s_y v_2 \\ 0 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix}$$

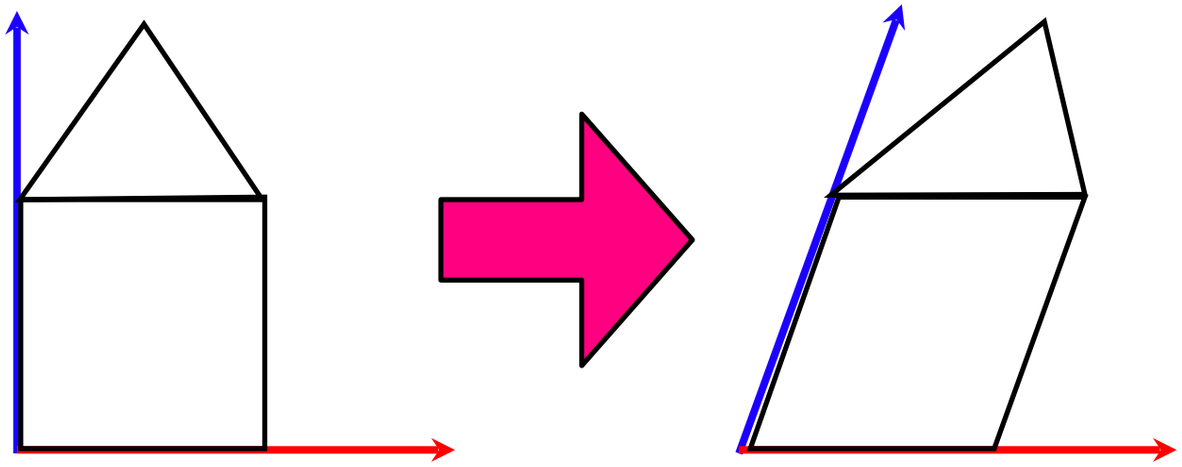


Shear

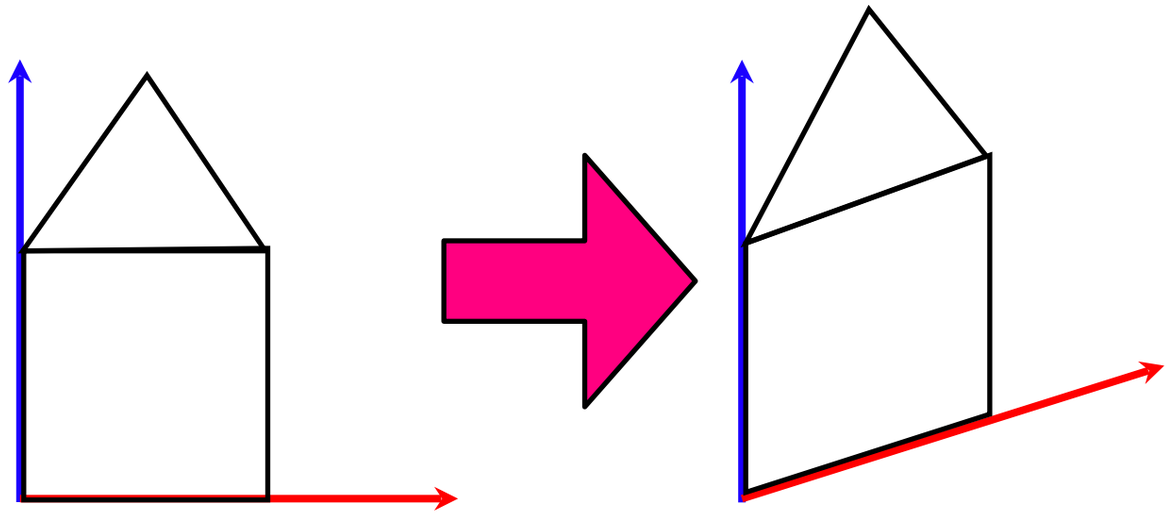
- Shear is the unwanted child of affine transformations.
- It can occur when you scale axes non-uniformly and then rotate.
- It does not preserve angles.
- Usually it is not something you want.
- It can be avoided by always scaling uniformly.

Shear

Horizontal:



Vertical:



2D Shear

- Horizontal:

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

- Vertical:

$$\begin{pmatrix} q_1 \\ q_2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ v & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$$

Matrices in UNSWgraph

- The Matrix3 class represents a 3x3 matrix.
- There are methods for creating translation, rotation, scale and shear matrices
- Matrices are stored in column-major order
- See Matrix3.java

Exercise

- What would the matrix for scaling -1 in the x and y direction look like?

- What would the matrix for rotating by 180 degrees look like?

Composing transformations

- We can combine a series of transformations by post-multiplying their matrices. The composition of two affine transformations is also affine.

- Eg: Translate, then rotate, then scale:

$$\mathbf{M} = \mathbf{M_T}\mathbf{M_R}\mathbf{M_S}$$

$$Q = \mathbf{M}P = \mathbf{M_T}\mathbf{M_R}\mathbf{M_S}P$$

Composing Transformations

- This is exactly what UNSWgraph does in the CoordFrame2D class.

- CoordFrame2D stores a matrix.

- The translate, rotate and scale methods take that matrix and post-multiply it by the corresponding transformation matrix.

- See CoordFrame2D.java

Exercise

- What would the matrix be for the transform defined by this coordinate frame?

```
CoordFrame2D frame =
    CoordFrame2D.identity()
    .translate(1,2)
    .rotate(90);
```

Exercise

- Suppose we continue from our last example and do the following

```
frame = frame.scale(2,2);
```

- What's the matrix for this transformation?

Decomposing transformations

- Every 2D affine transformation can be decomposed as:

$$\mathbf{M} = \mathbf{M}_{\text{translate}}\mathbf{M}_{\text{rotate}}\mathbf{M}_{\text{scale}}\mathbf{M}_{\text{shear}}$$

- If scaling is always uniform in both axes, the shear term can be eliminated:

$$\mathbf{M} = \mathbf{M}_{\text{translate}}\mathbf{M}_{\text{rotate}}\mathbf{M}_{\text{scale}}$$

Decomposing transformations

- To decompose the transform, consider the matrix form:

$$\begin{pmatrix} i_1 & j_1 & \phi_1 \\ i_2 & j_2 & \phi_2 \\ 0 & 0 & 1 \end{pmatrix}$$

axes axes origin

Decomposing transformations

- Assuming uniform scaling and no shear

$$\text{translation} = (\phi_1, \phi_2, 1)^\top$$

$$\text{rotation} = atan2(i_2, i_1)$$

$$\text{scale} = |\mathbf{i}|$$

- Note: atan2(i2,i1) is atan(i2/i1) aka $\tan^{-1}$(i2/i1) adjusting for i1 being 0. If i1 == 0 (and i2 is not) we get 90 degrees if i2 is positive or -90 if i2 is negative.

Example

$$\begin{pmatrix} 0 & -2 & 1 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

Translation: $(1,2,1)^{\top}$

Rotation: $atan2(2,0) = 90°$

Scale: $|\mathbf{i}| = |\mathbf{j}| = 2$

Also we can tell that axes are still

perpendicular as $\mathbf{i} \cdot \mathbf{j} = 0$

Exercise

$$\begin{pmatrix} 1.414 & -1.414 & 0.5 \\ 1.414 & 1.414 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

- What are the axes of the coordinate frame this matrix represents? What is the origin? Sketch it.
- What is the scale of each axis?
- What is the angle of each axis?
- Are the axes perpendicular?

Assignment 1

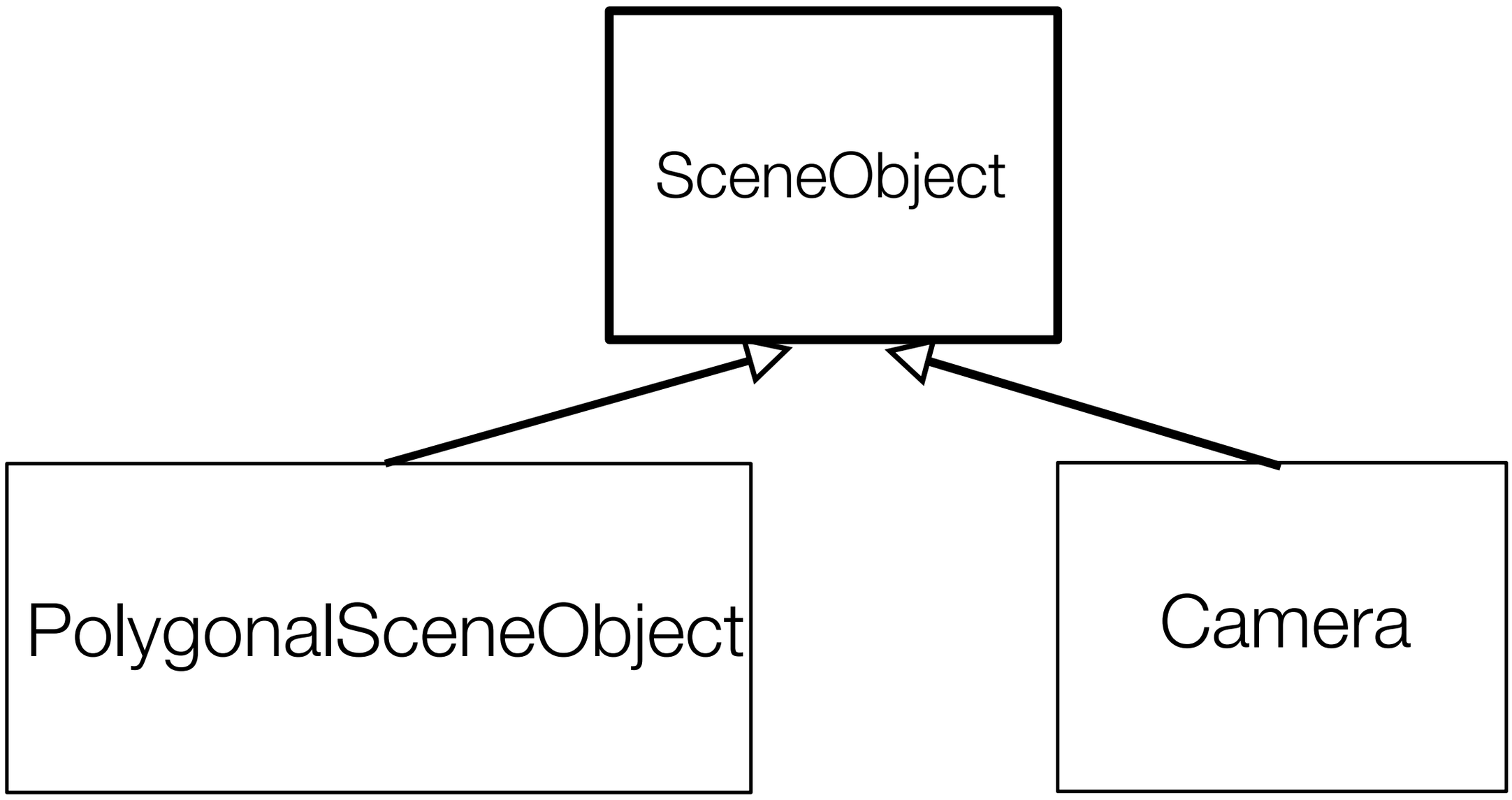
- Game Engine: Scene Tree

- Provided code: Fill in Code in TODO comments/ tags

- SceneObject : node in the n-ary tree

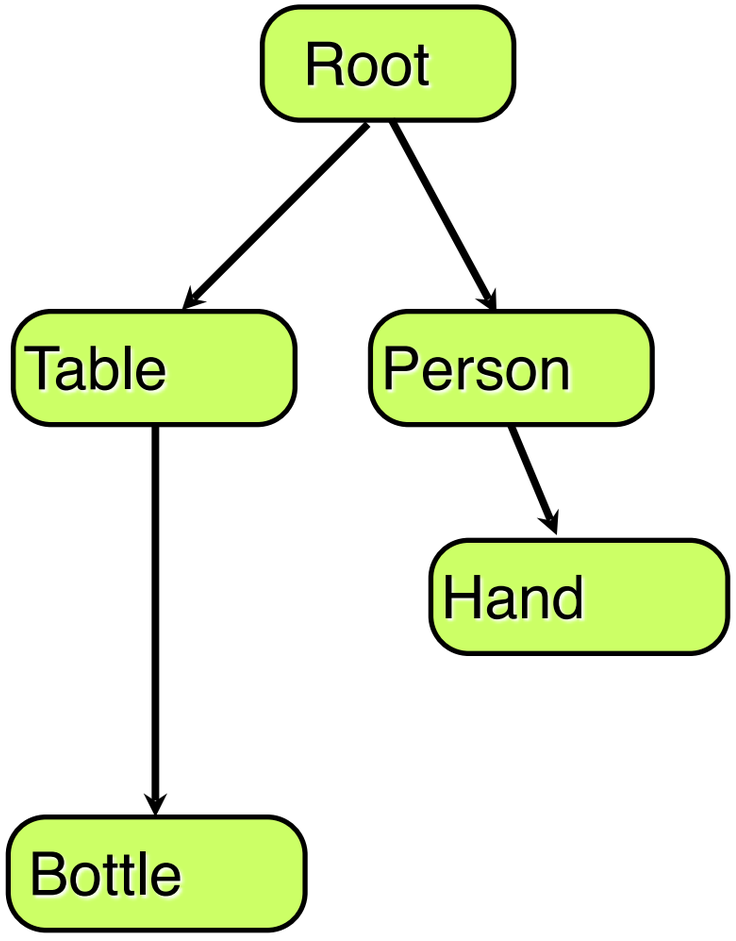
  - each node has t, r, s (uniform scaling)
  - 0..n children
  - 1 parent unless root object.

Scene Graph Class Diagram



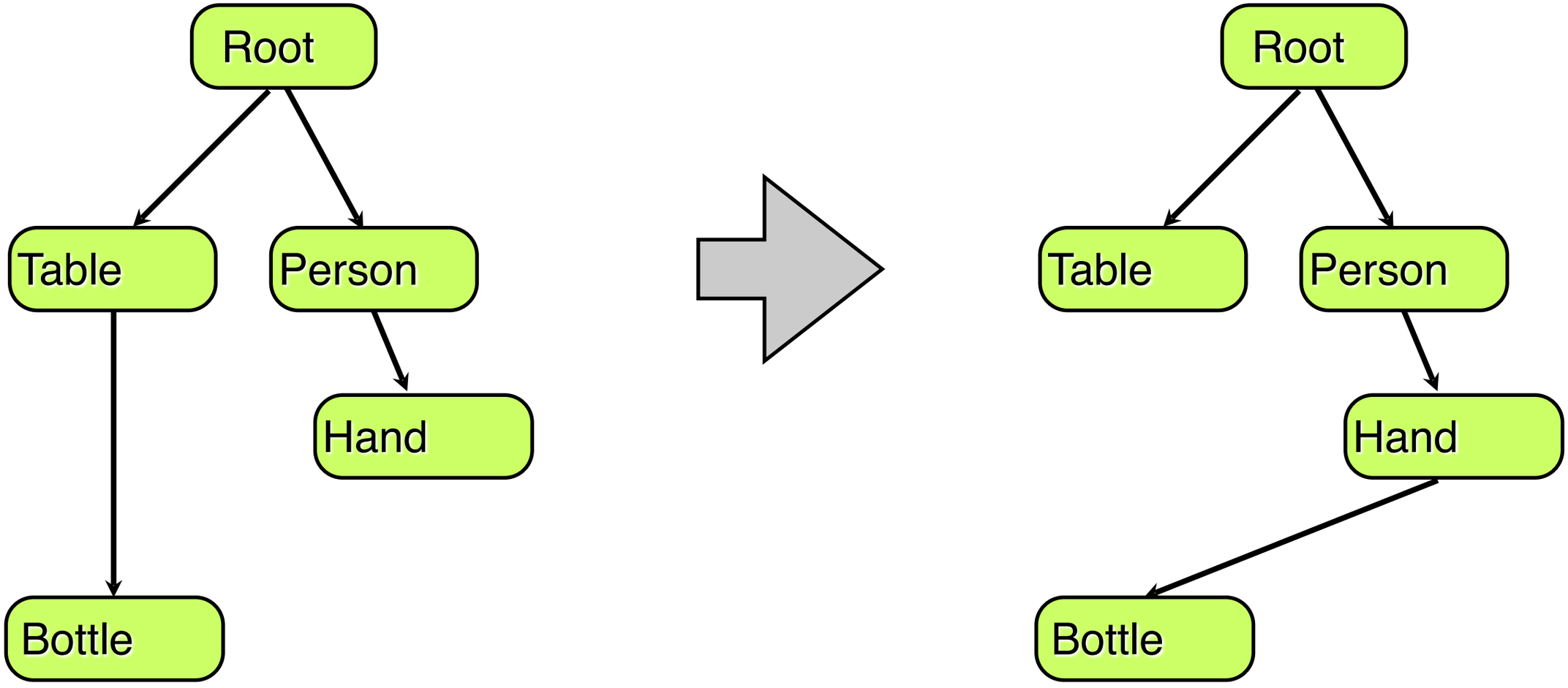
Scene trees

- Recall that scene trees describe how objects are connected by transformations in a scene?



Reparenting

- What if the person picks up the bottle?
- What's the new transformation?



Reparenting

- In assignment this will be performed by the setParent() method that you must implement.

- Make sure you understand the maths before you implement it!

- Exercise in next week's tute

Camera

- In addition to the transformation properties inherited from SceneObject the Camera has an aspect ratio

- The aspect ratio is the ratio of the width to the height.

- When calculating the view transform, you will need to take that into account.