

# COMP3421

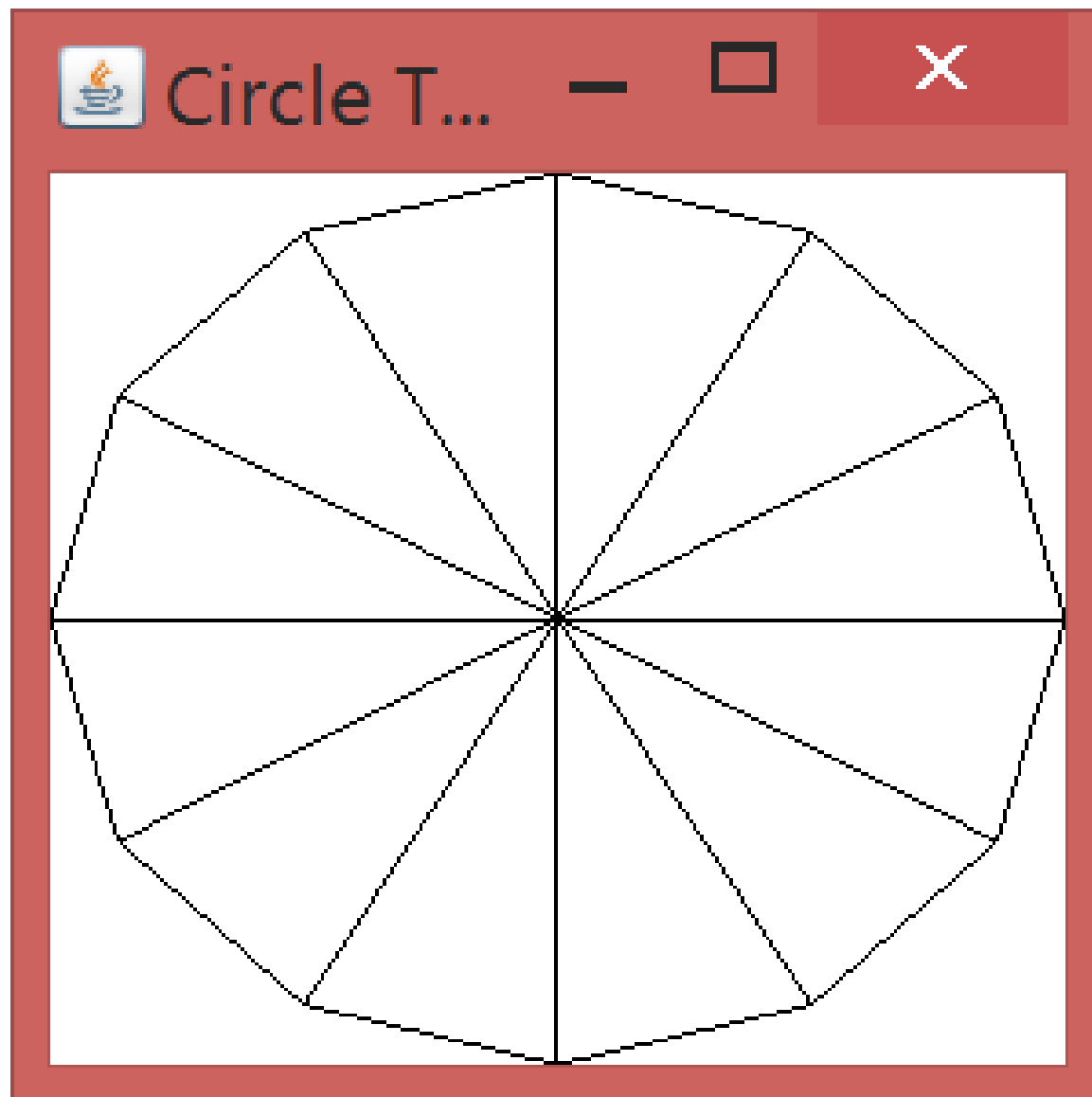
Week 2 - Transformations in 2D and Vector  
Geometry Revision

# Exercise

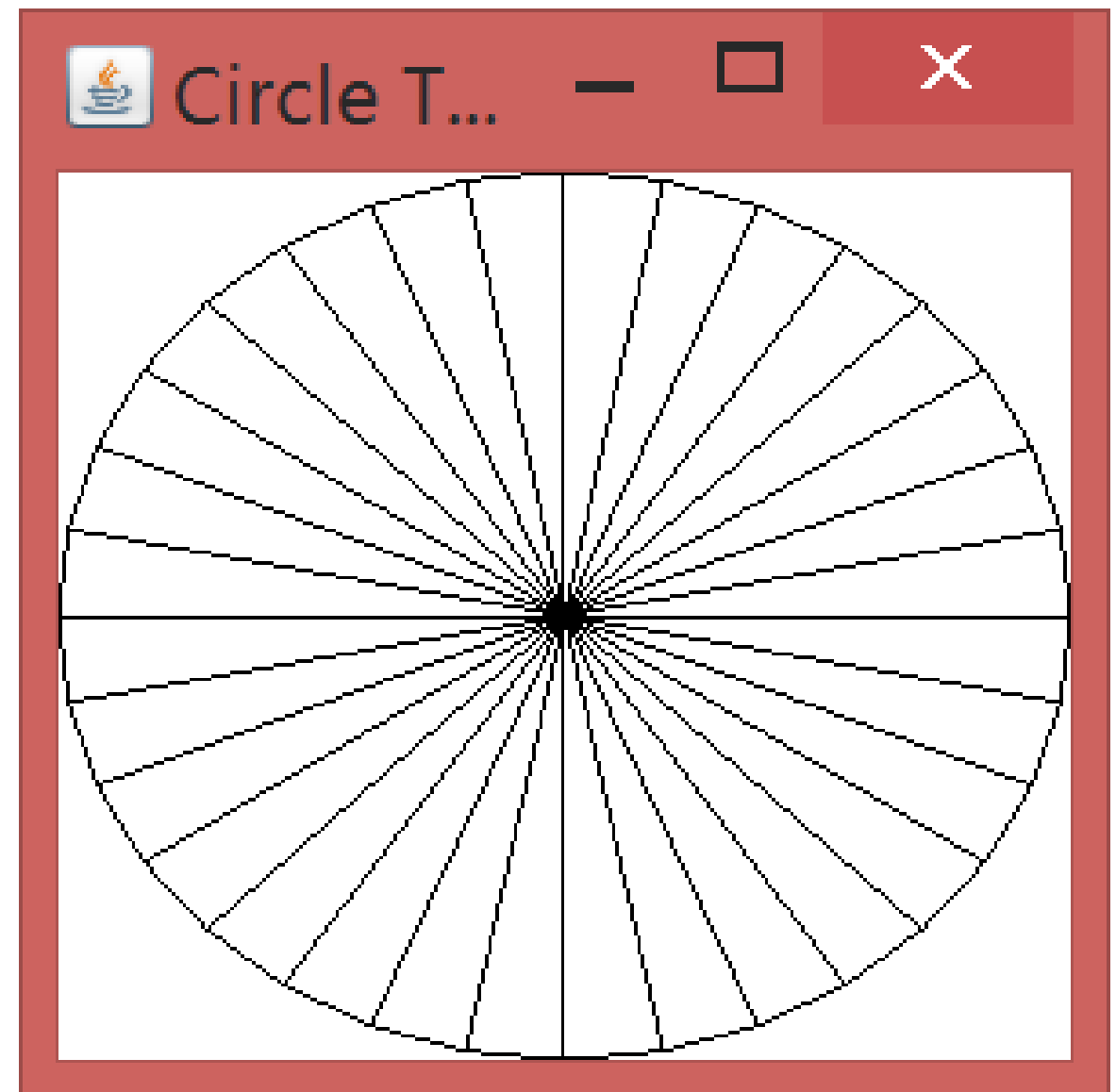
1. Write code to draw (an approximation) of the surface of a circle at centre  $0,0$  with radius 1 using triangle fans.
2. At home, modify the code to draw (an approximation) of the outline of a circle at centre  $(1,2)$  with radius 3 using a line strip.

# Solution

**12 Triangles**



**32 Triangles**



# Solution

We can generate points for increasing theta values using

$$x = \text{radius} * \cos(\text{theta})$$

$$y = \text{radius} * \sin(\text{theta})$$

Smaller increments give us more points/triangles and a more realistic, smoother 'circle'.

Note: Java math library uses radians, JOGL libraries tend to use degrees

See code for more details

# Transformation Matrices

GL defines a number of different matrices for transformations.

The two we will encounter are the **model-view** matrix and the **projection matrix**.

So far we have set the projection matrix, which tells GL what kind of camera we are using. We have used an **orthographic camera** (more on this later).

# glMatrixMode

You need to tell GL which matrix you are currently modifying:

```
// select projection matrix
gl.glMatrixMode(GL2.GL_PROJECTION) ;

// perform operations ...

// select model-view matrix
gl.glMatrixMode(GL2.GL_MODELVIEW) ;

// perform operations ...
```

Always make sure you have the **correct** matrix.

# Initialising Matrices

Always make sure you initialise your matrix when you use it for the first time.

We do this by setting it to the identity matrix (This is like setting a variable you are going to use for multiplication to 1)

```
//Specify which matrix you are using  
gl.glMatrixMode(...);  
//set it to the identity matrix  
gl.glLoadIdentity();
```

# Model-view transformation

The **model-view transformation** describes how the current **local** coordinate system maps to the **global** coordinate system.

It is useful to think of it as two transformations combined:

**model** transformation - local to world

**view** transformation - world to camera/eye

We will look at them separately.



# In OpenGL

To work with the model-view transform, first we select it:

```
gl.glMatrixMode (GL2.GL_MODELVIEW) ;
```

The should initialise it to the identity (i.e. no transformation) before we use it.

```
gl.glLoadIdentity() ;
```

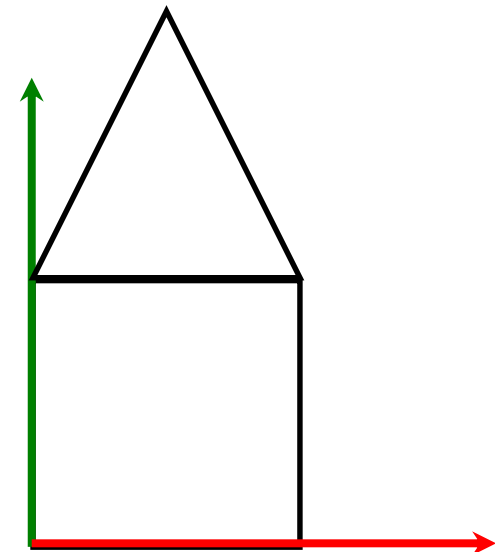
# Example

Drawing a house:

```
gl.glMatrixMode (GL2.GL_MODELVIEW) ;
```

```
gl.glLoadIdentity() ;
```

```
drawHouse() ;
```



# Transformations

We can then apply different transformations to the coordinate system:

```
gl.glTranslated(dx, dy, dz);
```

```
gl.glRotated(angle, x, y, z);
```

```
gl.glScaled(sx, sy, sz);
```

Subsequent drawing commands will be in the transformed coordinate system.

# glTranslated

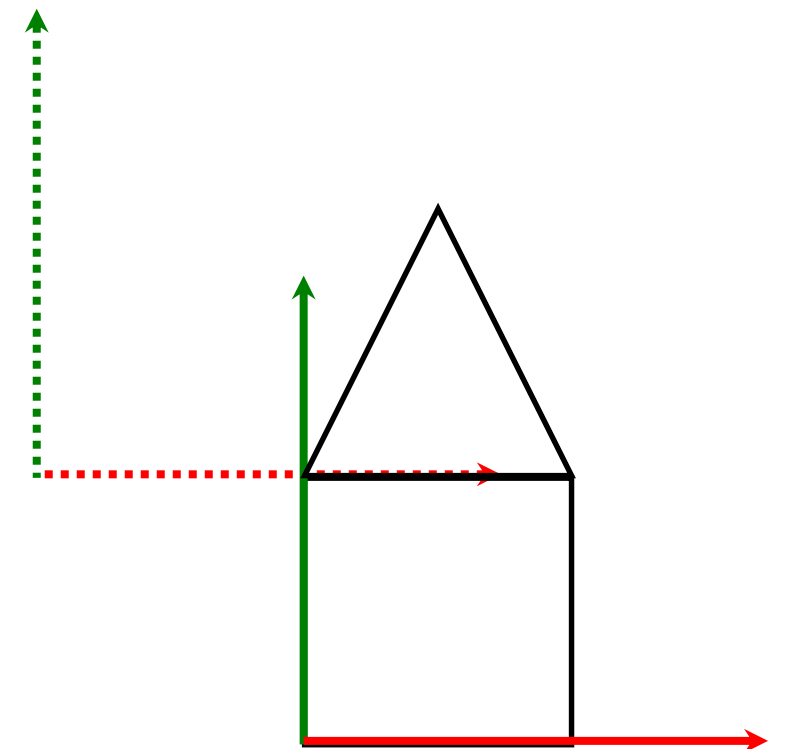
Translate the coordinate space by the specified amount along each axis.

```
gl.glMatrixMode(GL2.GL_MODELVIEW) ;  
gl.glLoadIdentity() ;
```

```
gl.glTranslated(1, -1, 0) ;
```

```
drawHouse() ;
```

In this case the **origin** of the co-ordinate frame moves.



# glRotated

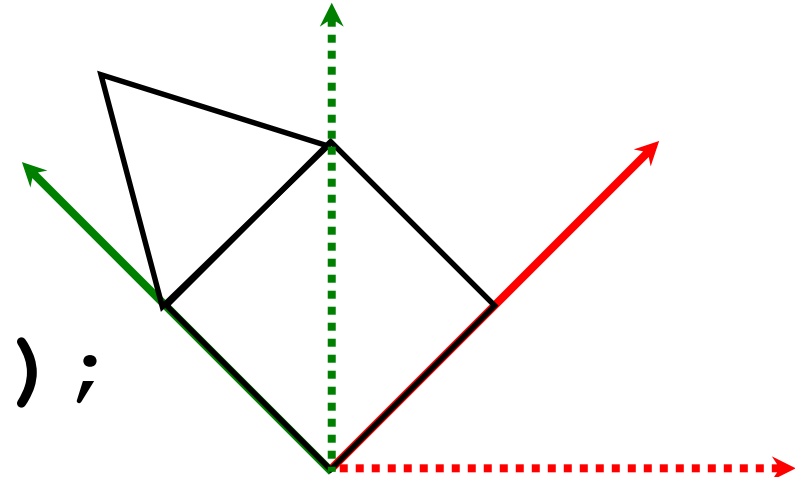
Rotate the coordinate space by the specified angle and axis.

```
gl.glMatrixMode(GL2.GL_MODELVIEW);  
gl.glLoadIdentity();
```

```
// rotate 45°  
// about the z-axis
```

```
gl.glRotated(45, 0, 0, 1);
```

```
drawHouse();
```



Notice, the **origin** of the co-ordinate frame doesn't move

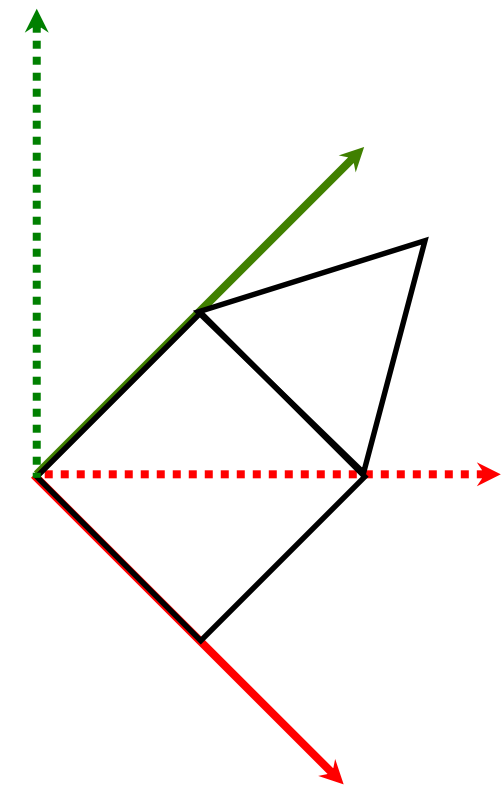
# glRotated

Angles are in degrees.

Positive rotations are rotating x towards y.

Negative rotations are rotating y towards x.

```
gl.glMatrixMode(  
    GL2.GL_MODELVIEW) ;  
gl.glLoadIdentity() ;  
  
// rotate -45°  
// about the z-axis  
gl.glRotated(-45, 0, 0, 1) ;  
  
drawHouse() ;
```



# glScaled

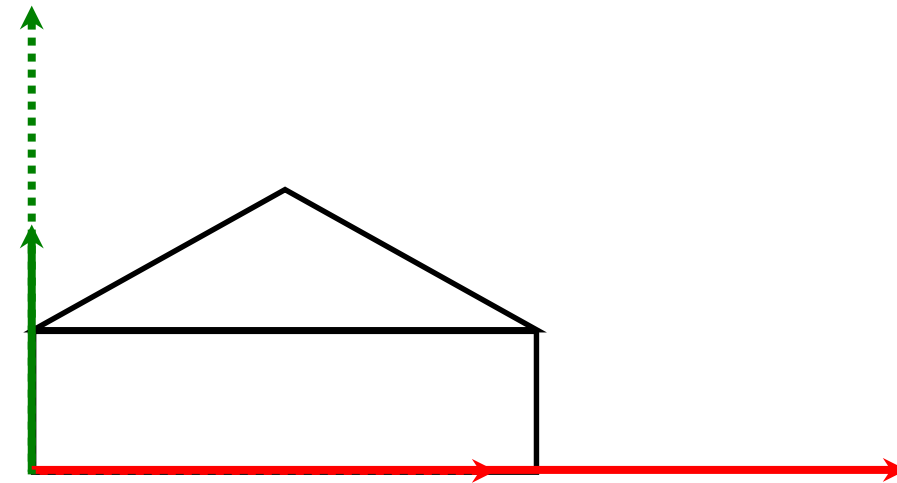
Scale the coordinate space by the specified amounts in the **x**, **y** and **z** (in 3d) directions.

```
gl.glMatrixMode(GL2.GL_MODELVIEW);
```

```
gl.glLoadIdentity();
```

```
gl.glScaled(2, 0.5, 1);
```

```
drawHouse();
```



Notice again, the **origin** of the co-ordinate doesn't move.

# glScaled

Negative scales create reflections.

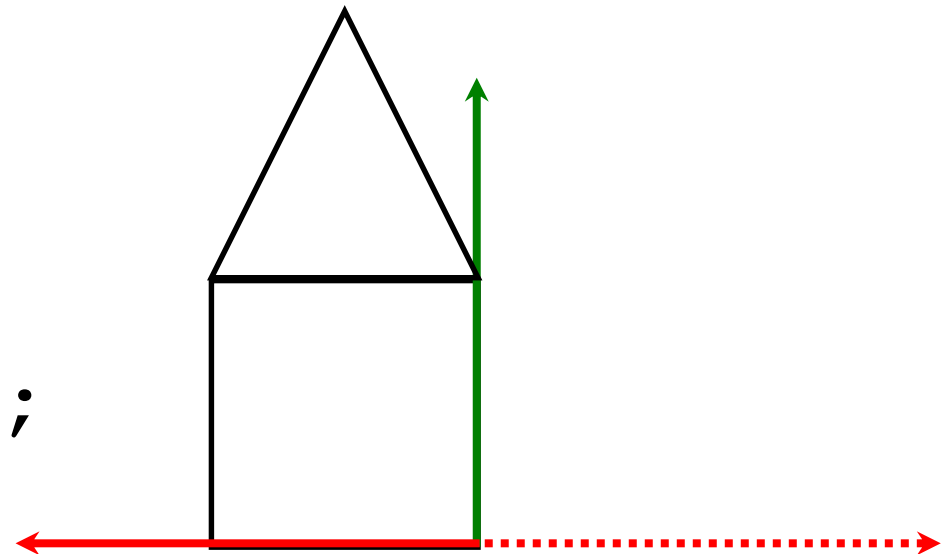
```
gl.glMatrixMode(GL2.GL_MODELVIEW);
```

```
gl.glLoadIdentity();
```

```
// flip horizontally
```

```
gl.glScaled(-1, 1, 1);
```

```
drawHouse();
```

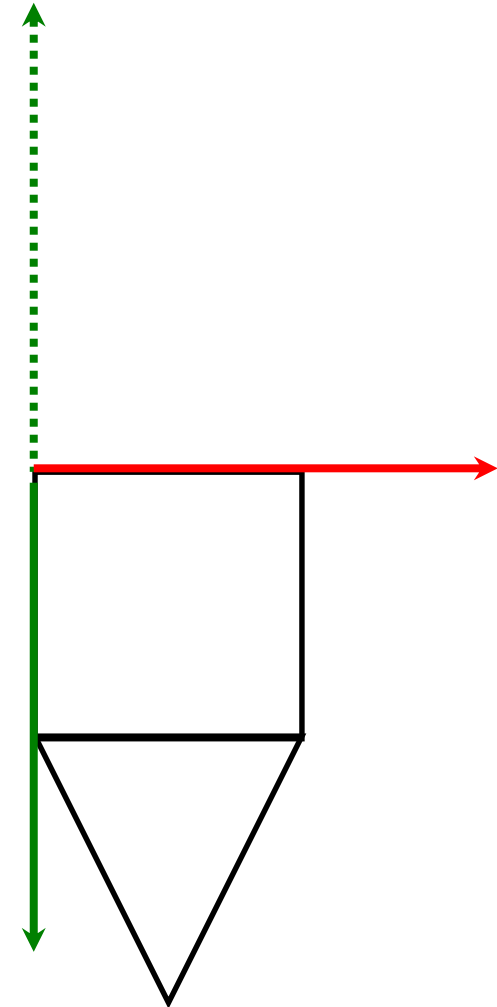




# glScaled

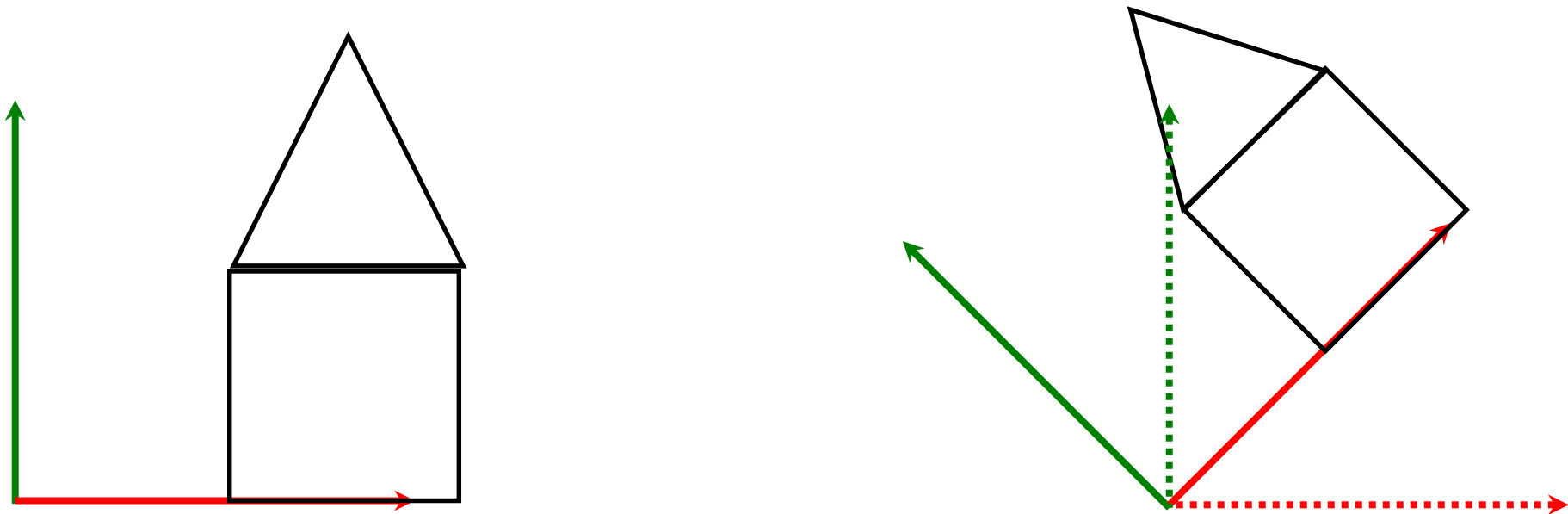
Negative scales create reflections.

```
gl.glMatrixMode(GL2.GL_MODELVIEW);  
gl.glLoadIdentity();  
// flip vertically  
gl.glScaled(1, -1, 1);  
drawHouse();
```



# glRotated

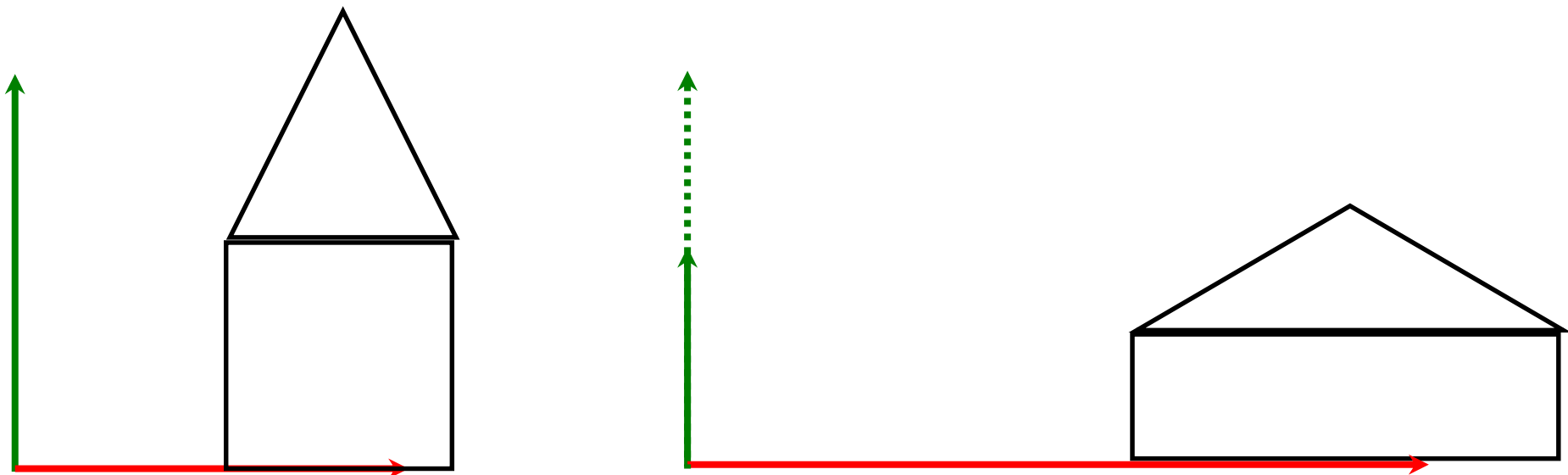
If the object is not located at the origin, it might not do what you expect when its co-ordinate frame is rotated.



The origin of the co-ordinate frame is the pivot point.

# glScaled

If the object is not located at the origin, the object will move further from the origin if its co-ordinated frame is scaled



Only points at the origin remain unchanged.

# Successive Transformations

We can think of transformations in two ways

1. An object being transformed or altered within a **fixed co-ordinate** system.
2. The co-ordinate system of the object being transformed. This is generally the way we will think of it.

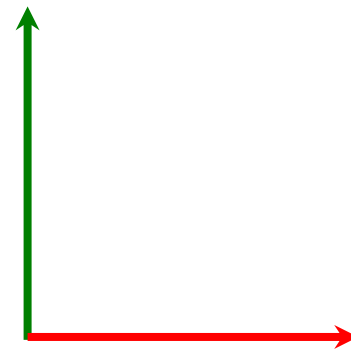
# Brown University Game

[http://graphics.cs.brown.edu/research/exploratory/freeSoftware/repository/edu/brown/cs/exploratories/applets/transformationGame/transformation\\_game\\_guide.html](http://graphics.cs.brown.edu/research/exploratory/freeSoftware/repository/edu/brown/cs/exploratories/applets/transformationGame/transformation_game_guide.html)

# Combining transforms

A sequence of transforms take place in successive coordinate systems:

```
gl.glLoadIdentity();
```

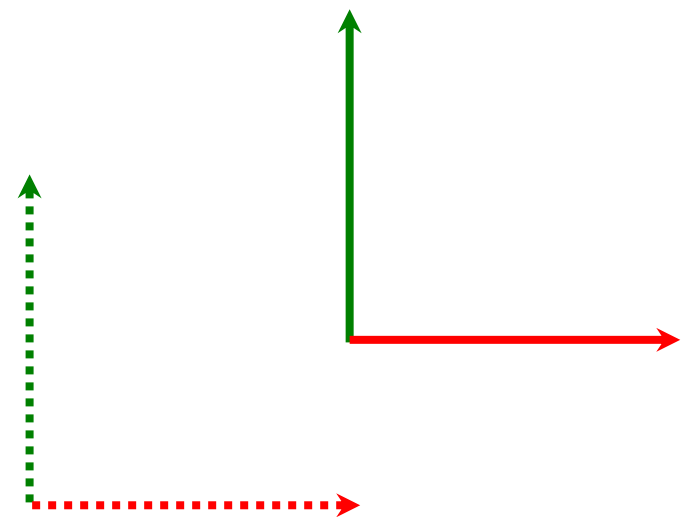


# Combining transforms

A sequence of transforms take place in successive coordinate systems:

```
gl.glLoadIdentity();
```

```
gl.glTranslated(2, 1, 0);
```



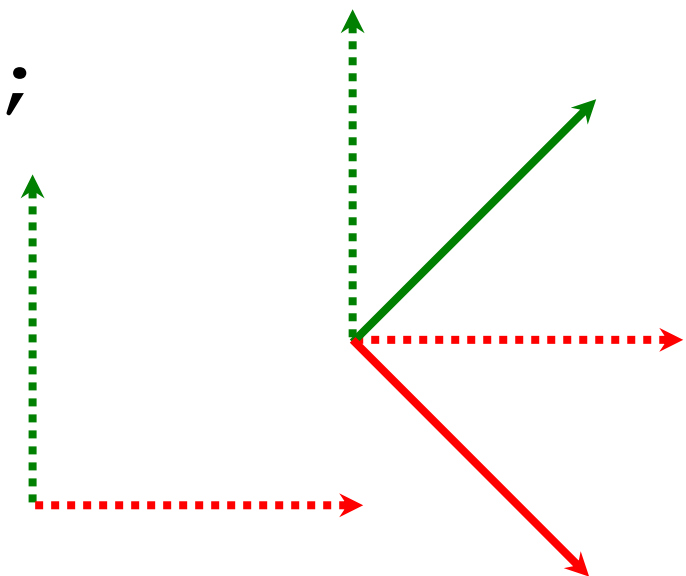
# Combining transforms

A sequence of transforms take place in successive coordinate systems:

```
gl.glLoadIdentity();
```

```
gl.glTranslated(2, 1, 0);
```

```
gl.glRotated(-45, 0, 0, 1);
```





# Combining transforms

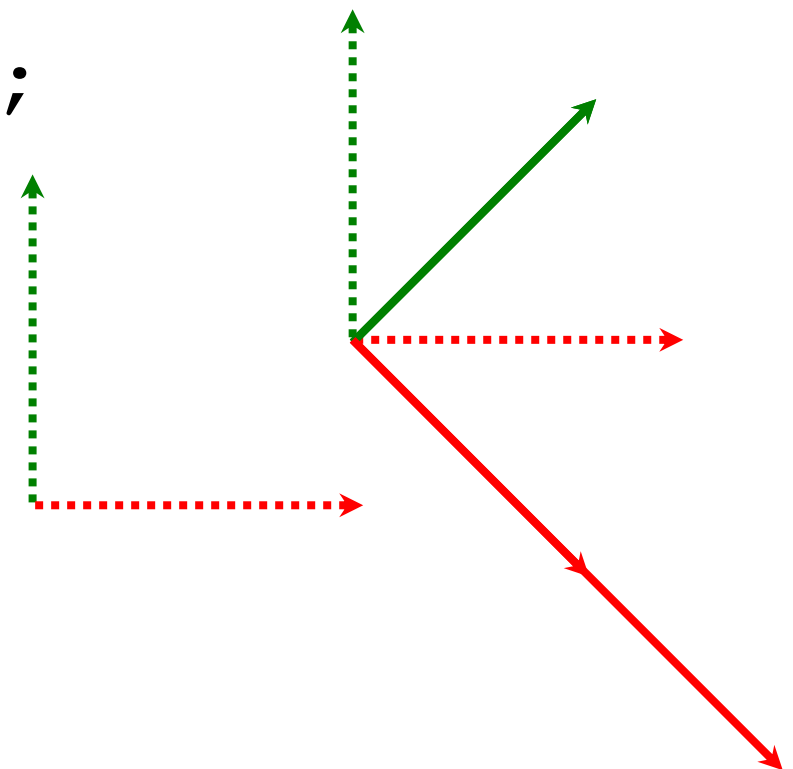
A sequence of transforms take place in successive coordinate systems:

```
gl.glLoadIdentity();
```

```
gl.glTranslated(2, 1, 0);
```

```
gl.glRotated(-45, 0, 0, 1);
```

```
gl.glScaled(2, 1, 1);
```



# Combining transforms

A sequence of transforms take place in successive coordinate systems:

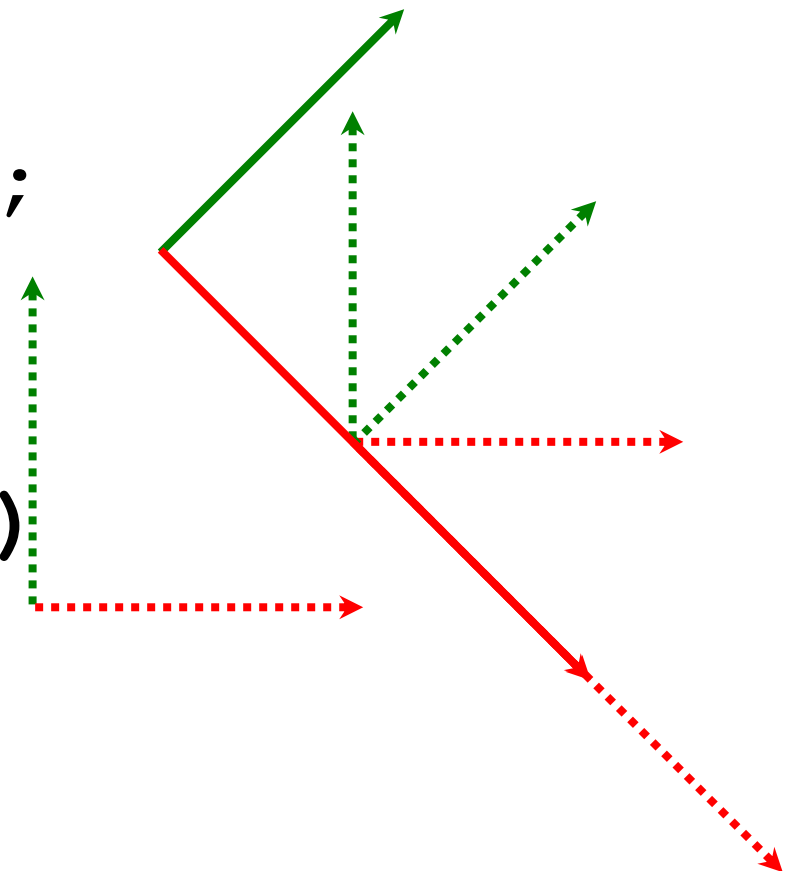
```
gl.glLoadIdentity();
```

```
gl.glTranslated(2, 1, 0);
```

```
gl.glRotated(-45, 0, 0, 1);
```

```
gl.glScaled(2, 1, 1);
```

```
gl.glTranslated(-0.5, 0, 0)
```



# Order matters

Note that the **order of transformations** matters.

translate then rotate  $\neq$  rotate then translate

translate then scale  $\neq$  scale then rotate

rotate then scale  $\neq$  scale then rotate

# Instance Transformation

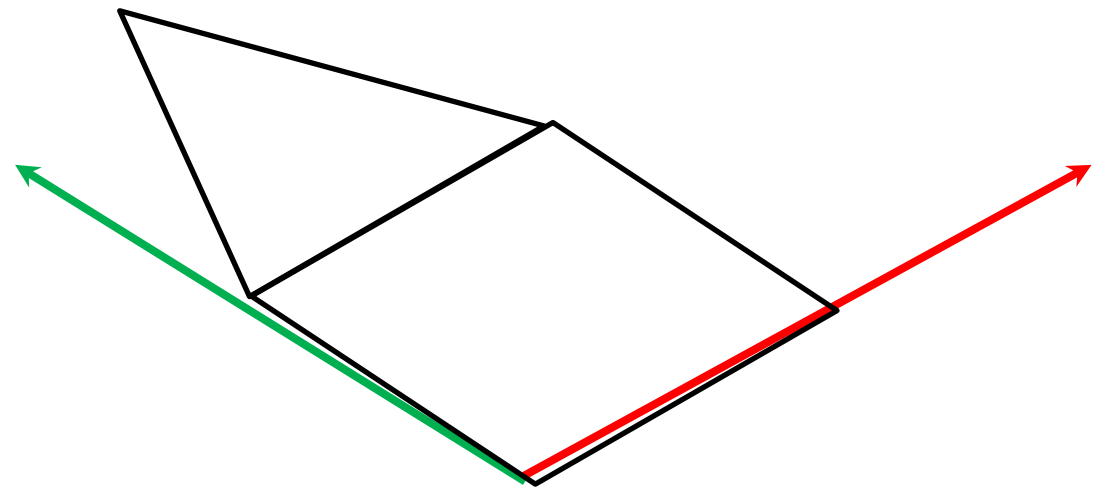
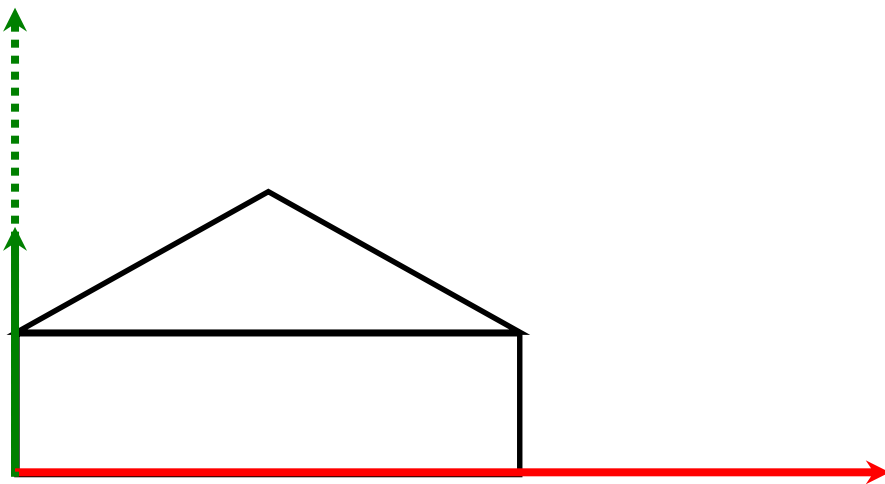
**Usually** we want:  $\text{translate}(T)$ ,  $\text{rotate}(R)$ ,  $\text{scale}(S)$  :  $M = TRS$

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

We can have multiple occurrences in the scene at the desired size orientation and location by applying the desired instance transformation

# Non-uniform Scaling then Rotating

If we scale by different amounts in the x direction to the y direction and then rotate, we get unexpected and often unwanted results. Angles are not preserved.



# Rotating about an arbitrary point.

So far all rotations have been about the origin.  
To rotate about an arbitrary point.

1. Translate to the point

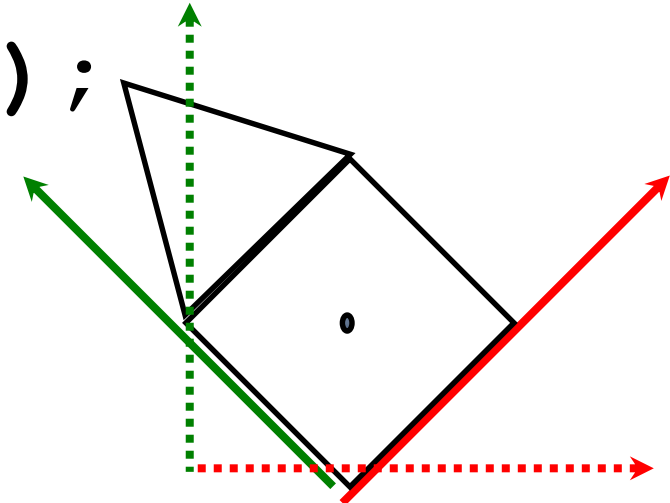
```
gl.glTranslated(0.5, 0.5, 0);
```

2. Rotate

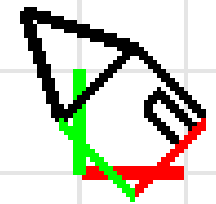
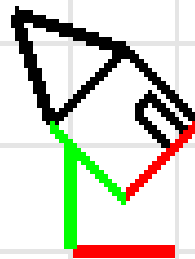
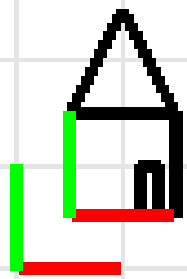
```
gl.glRotated(45, 0, 0, 1);
```

3. Translate back again

```
gl.glTranslated(-0.5, -0.5, 0);
```



# Rotating about an arbitrary point.



# Current Transformation (CT)

Calls to `glTranslate`, `glRotate` and `glScale` modify (post multiply – more on this later) the current transformation/co-ordinate frame.

Every time `glVertex2d()` is called, the fixed function pipeline transforms the given point by the *CT*.



# Push and pop

Often we want to store the current transformation/coordinate frame, transform it and then restore the old frame again.

GL provides a stack of matrices for this purpose. Push and pop using:

```
// store the current matrix
```

```
gl.glPushMatrix();
```

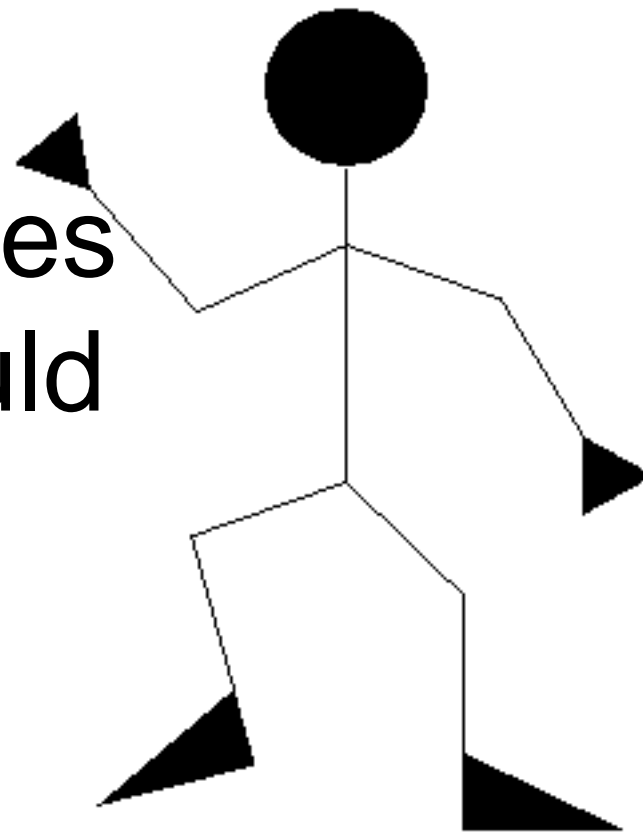
```
// restore the last pushed matrix
```

```
gl.glPopMatrix();
```

# Scene Graphs

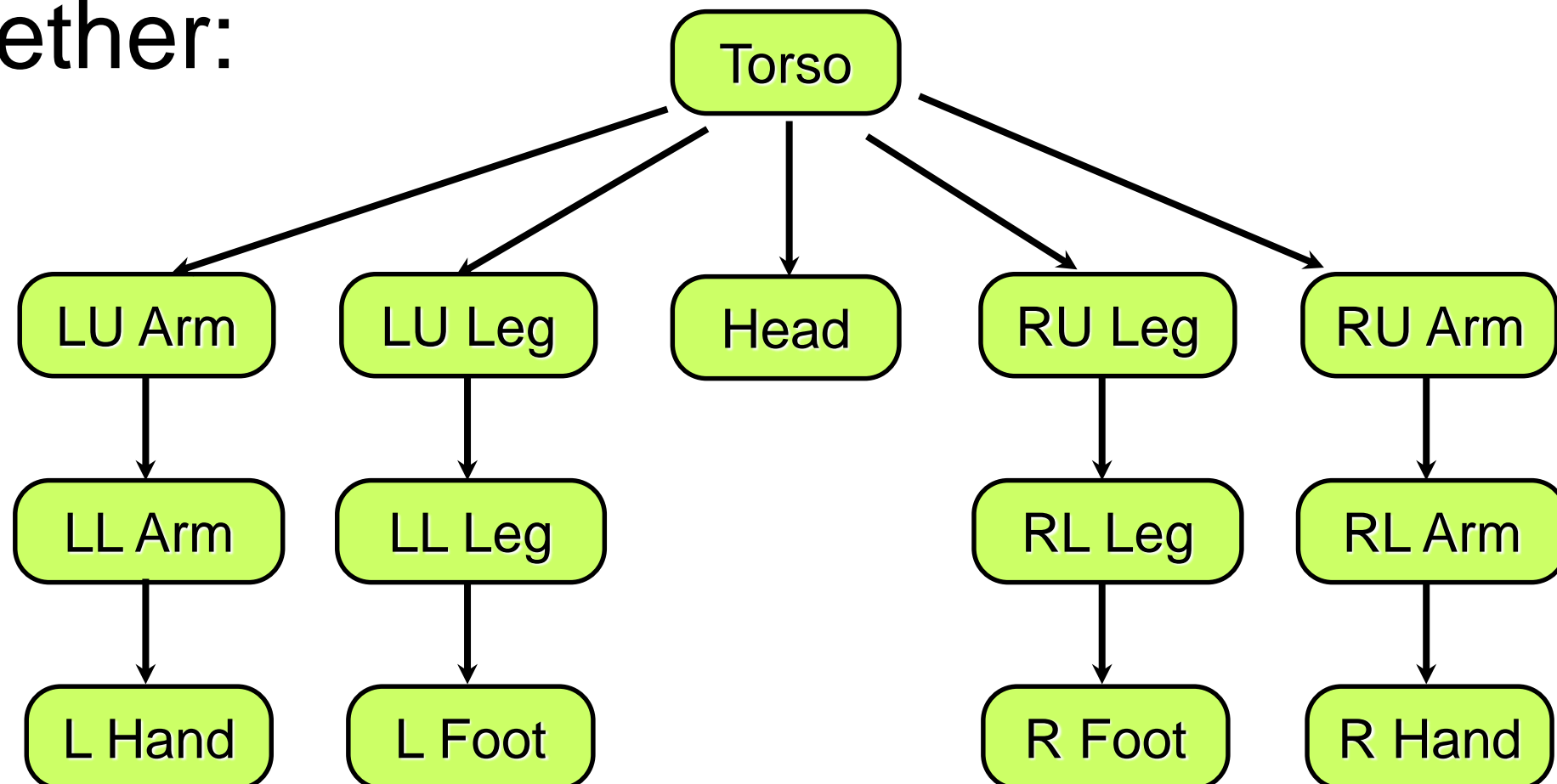
Consider drawing and animating a figure such as this person:

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



# Scene graph

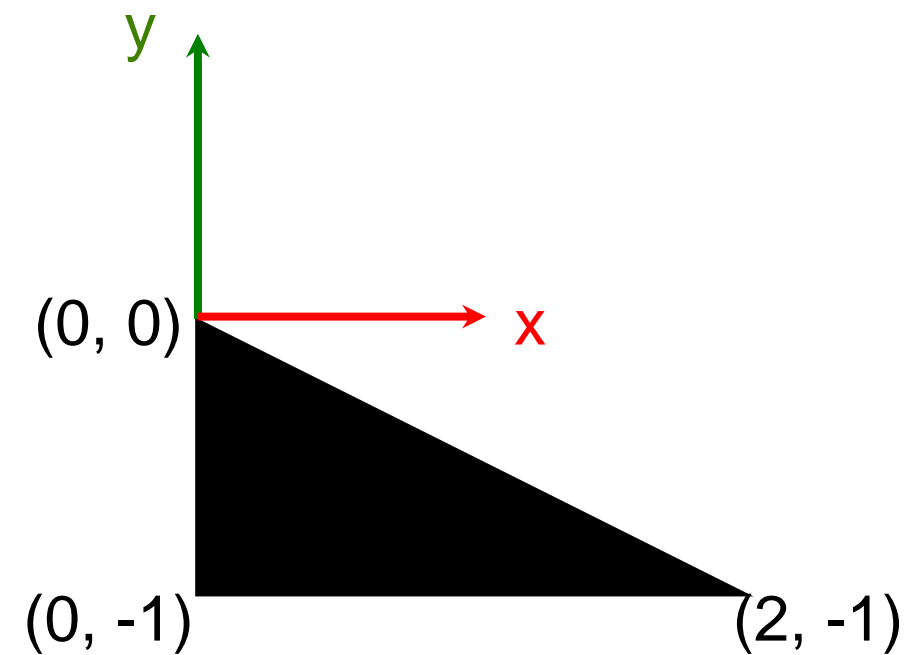
To represent a complex scene we use a **scene graph**. This tree describes how different objects in the scene are connected together:



# Coordinate system

We draw each part in its own local coordinate system:

```
// draw a foot  
gl.glBegin(GL2.GL_POLYGON) ;  
  
    gl.glVertex2d(0, 0) ;  
    gl.glVertex2d(0, -1) ;  
    gl.glVertex2d(2, -1) ;  
  
gl.glEnd() ;
```



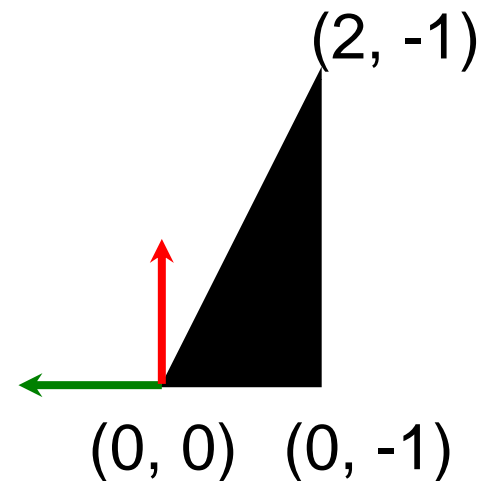
# Coordinate system

Then we transform the coordinate system:

translating

rotating

scaling



To get it into the position we want.

But from the object's point of view, nothing has changed.

# Scene graph

Each part draws itself in its own local coordinate frame and then transforms the coordinate frame to draw its subparts appropriately.

When a node in the graph is moved, all its children move with it.

# Scene graph pseudocode

```
drawTree() {  
    push model-view matrix  
  
    translate to new origin  
    rotate  
    scale  
  
    draw this object  
  
    for all children:  
        child.drawTree()  
  
    pop matrix  
}
```

# Camera

So far we have assumed world coordinate  $(0, 0)$  is the centre of the world window.

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

# View transform

Mathematically if:

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

Then the view transform is:

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

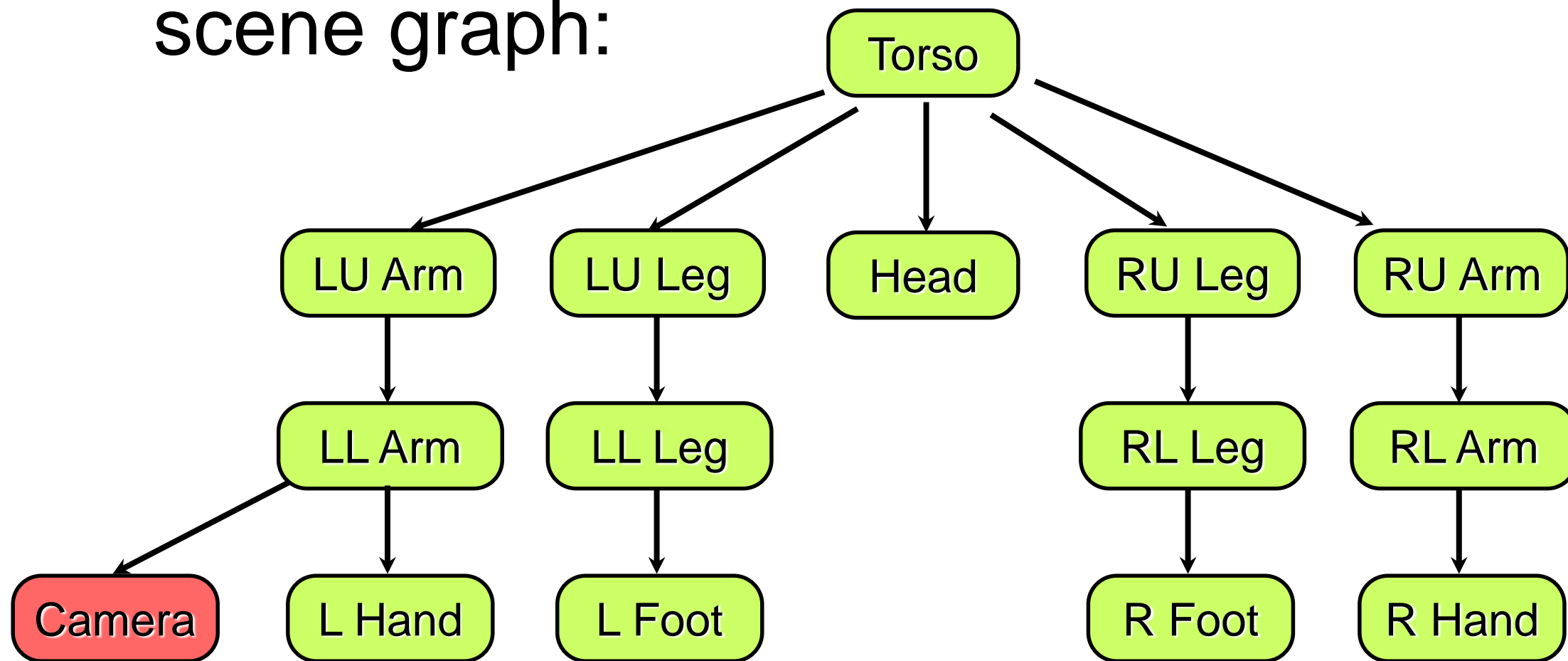
# Implementing a camera

To implement a camera, we need to apply the view transform **before** the model transform:

```
gl.glMatrixMode(GL2.GL_MODELVIEW) ;  
gl.glLoadIdentity() ;  
  
// apply the view transform  
gl.glScaled(1.0 / cameraScale, ...);  
gl.glRotated(-cameraAngle, 0, 0, 1);  
gl.glTranslated(-camX, -camY, 0);  
  
// apply the model transform + draw...
```

# In the scene graph

We can add the camera as an object in our scene graph:



# In the scene graph

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

We can do this by working recursively up the scene graph.

We will cover the maths necessary to do this calculation in the rest of this and the following lecture.

# Assignment 1

Game Engine: Scene Graph (Tree)

Provided code: Fill in Code in TODO  
comments/tags

GameObject – node in the n-ary tree

- each node has t, r, s
- n children
- 1 parent

Subclasses: polygonalGameObject etc.

# Assignment 1

Automarking

JUnit 4 Unit Tests

diff image files that you output with  
required image output

Tutor subjective marking

MyCoolGameObject

Bonus Game



# Homework

Draw a possible scene graph for the sailing game given with assignment 1

# Coordinate frames

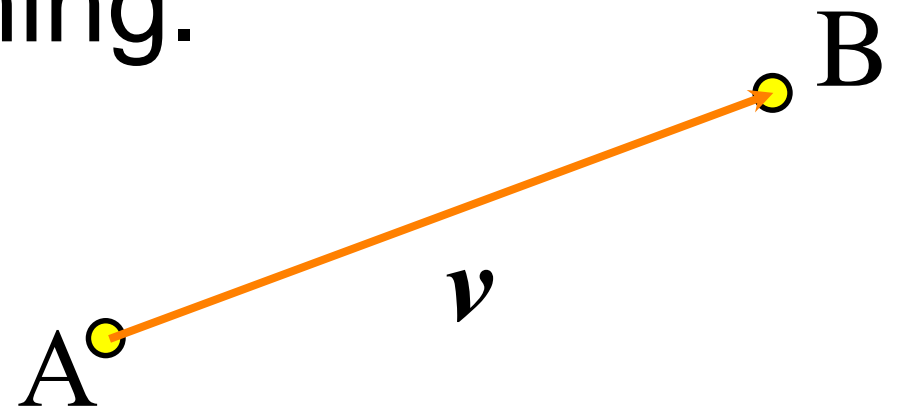
To represent **coordinate frames** and easily convert points in one frame to another we use **vectors** and **matrices**.

Some **revision** first.

# Vectors

Having the right vector tools greatly simplifies geometric reasoning.

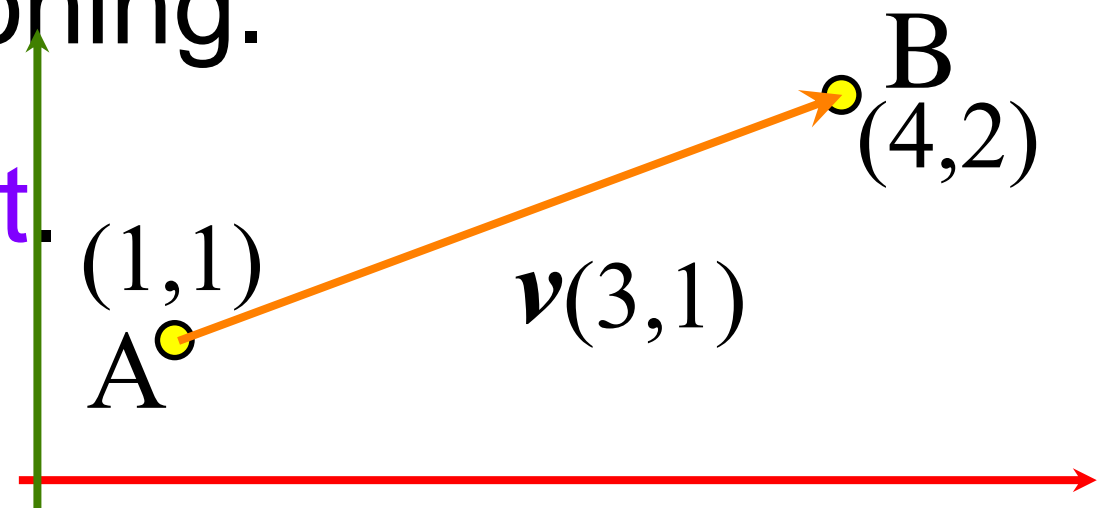
A **vector** is a **displacement**.



# Vectors

Having the right vector tools greatly simplifies geometric reasoning.

A **vector** is a **displacement**.



We represent it as a tuple of values in a particular coordinate system.

# Points vs Vectors

Vectors have

- length and direction
- no position

Points have

- position
- no length, no direction

# Points and Vectors

The **sum** of a point and a vector is a point.

$$P + \mathbf{v} = Q$$

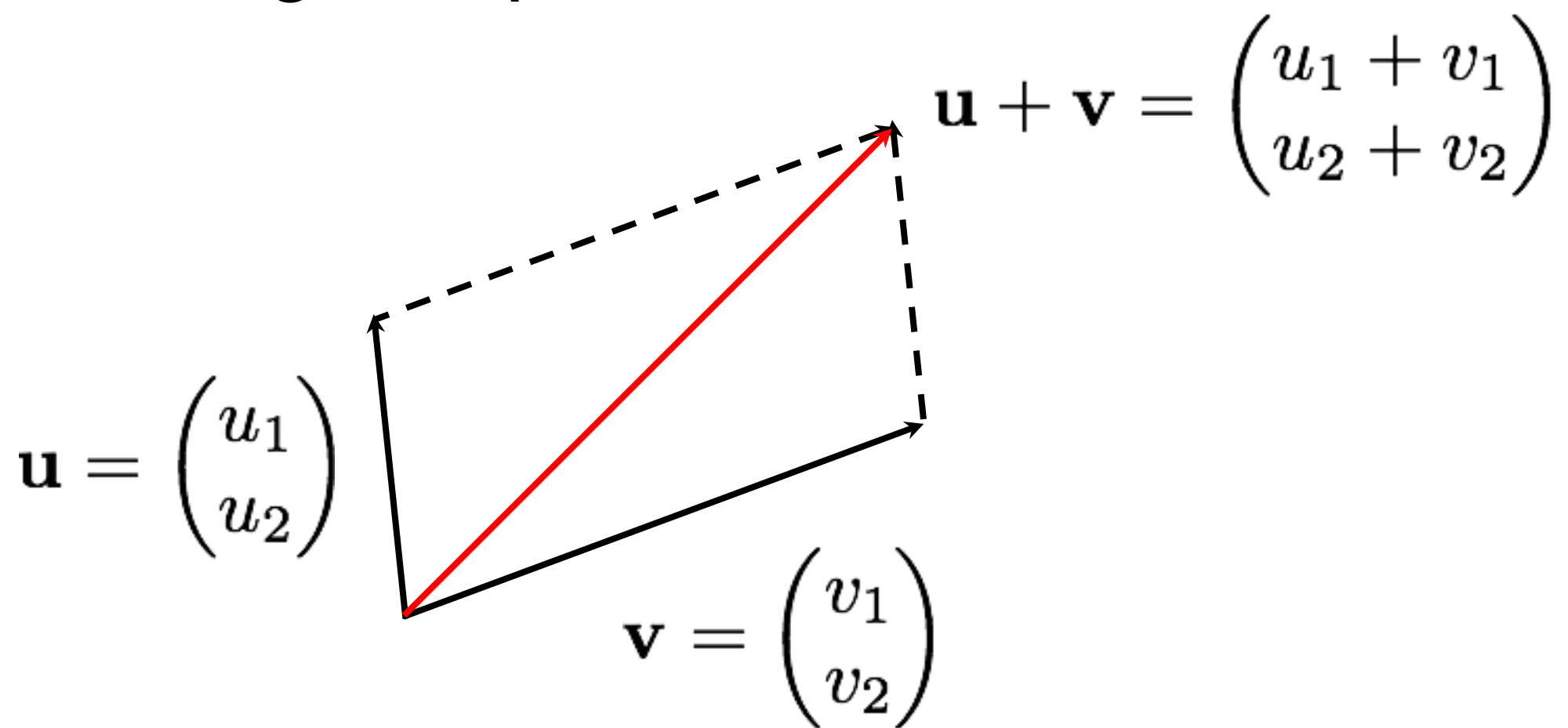
Which is the same as saying

The **difference** between two points is a vector:

$$\mathbf{v} = Q - P$$

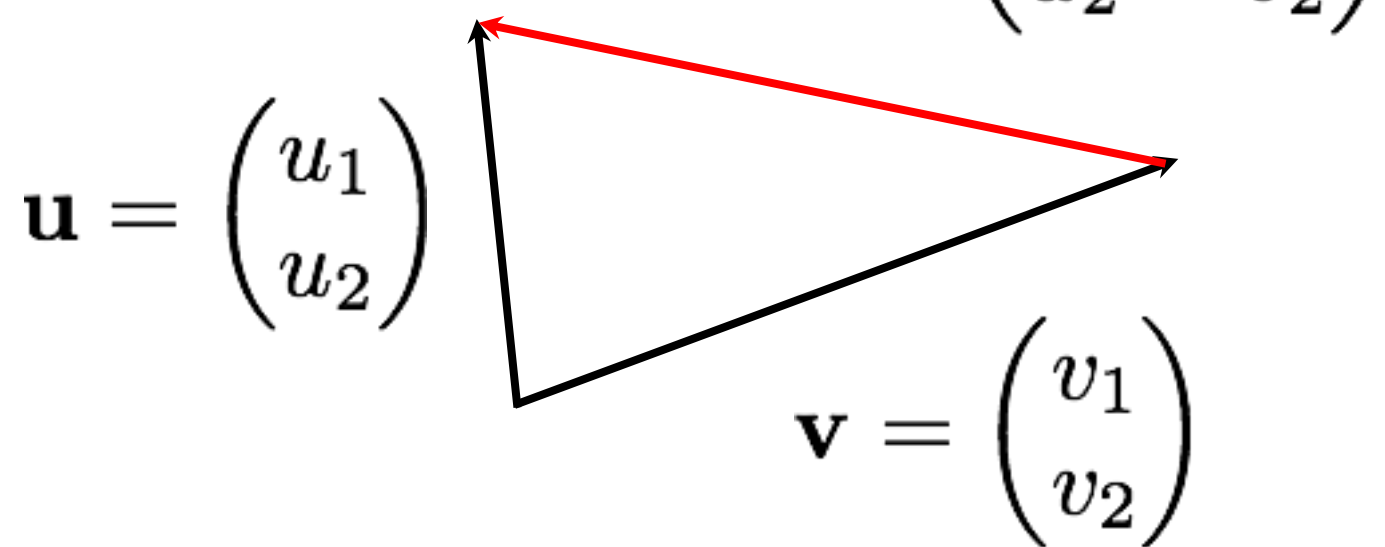
# Adding vectors

By adding components:



# Subtracting vectors

By subtracting components:

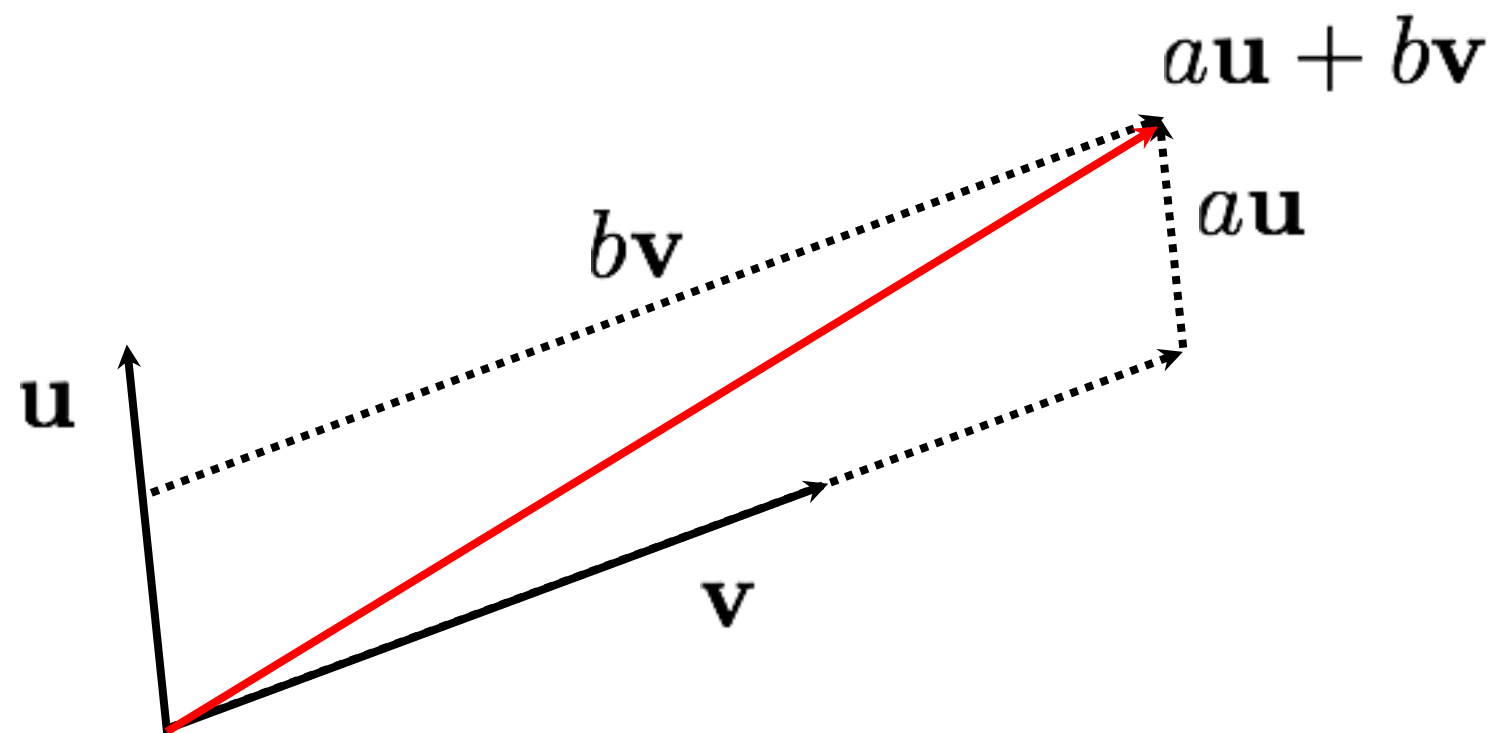
$$\mathbf{u} - \mathbf{v} = \begin{pmatrix} u_1 - v_1 \\ u_2 - v_2 \end{pmatrix}$$

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$
$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$



# Linear combinations

Any equation of the form:

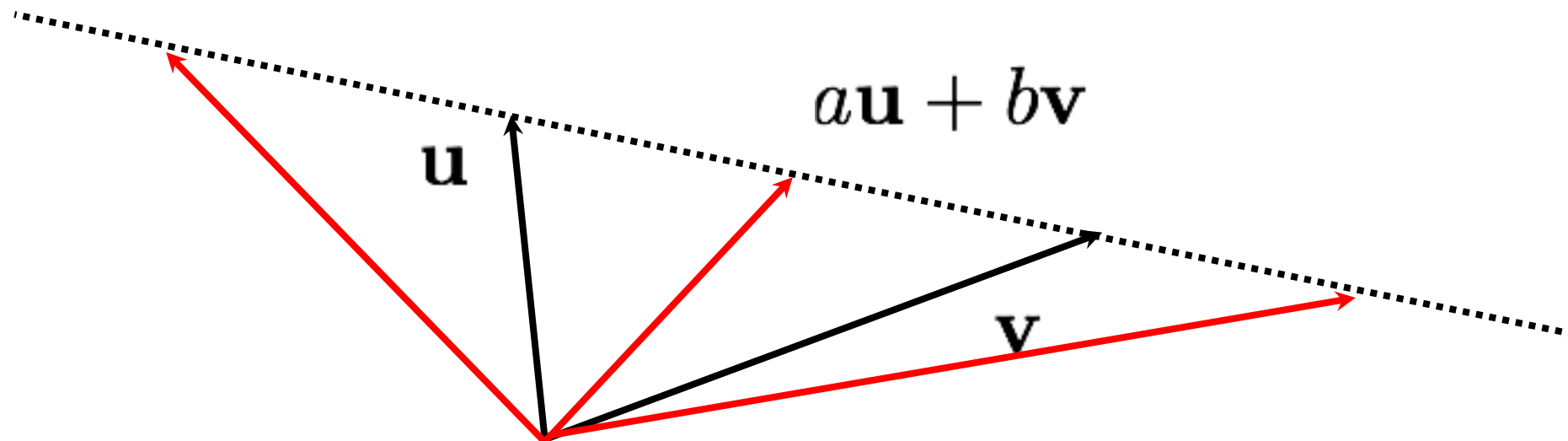
$$a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \dots + a_n \mathbf{v}_n$$



# Affine combinations

A linear combination where:

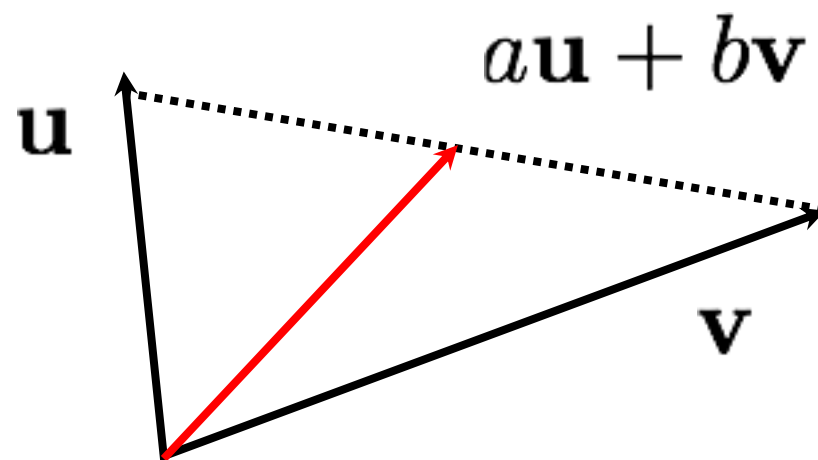
$$a_1 + a_2 + \dots + a_n = 1$$



# Convex combinations

An affine combination where:

$$0 \leq a_i \leq 1$$



# Magnitude

Magnitude (i.e. length)

$$|\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

Normalisation:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$$

$$|\hat{\mathbf{v}}| = 1$$

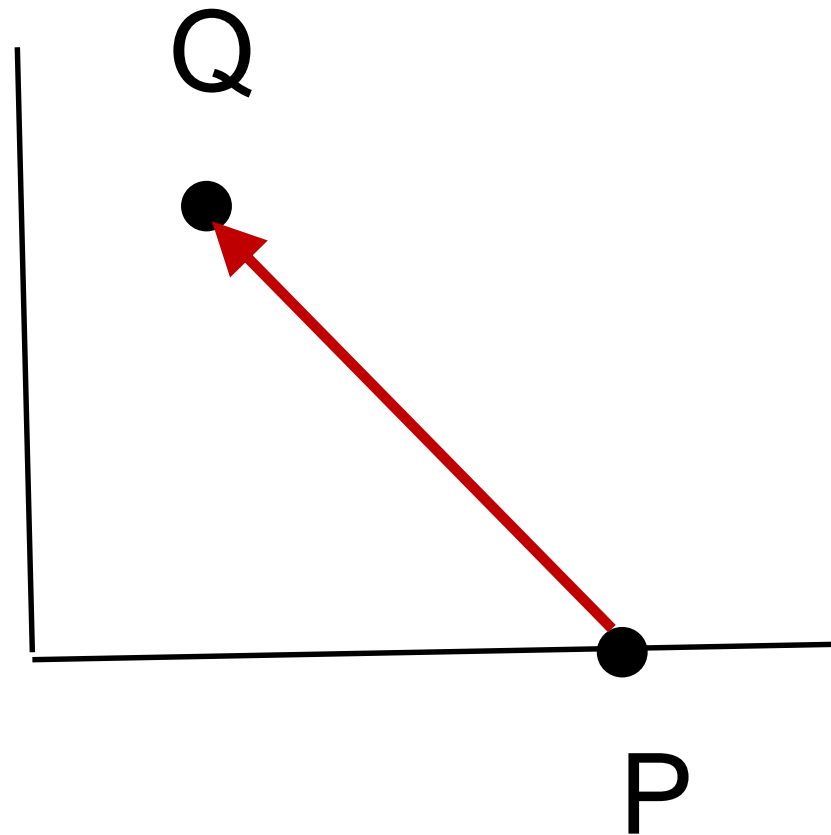
# Exercises

1. What is the vector  $\mathbf{v}$  from  $P$  to  $Q$  if  $P = (4,0)$ ,  $Q = (1,3)$  ?
2. Find the magnitude of the vector  $(1,1)$
3. Normalise the vector  $(8,6)$

# Solutions

1. What is the vector  $\mathbf{v}$  from P to Q if  $P = (4,0)$ ,  $Q = (1,3)$  ?

$$\begin{aligned}\mathbf{v} &= \mathbf{Q} - \mathbf{P} \\ &= (1,3) - (4,0) \\ &= (1 - 4, 3 - 0) \\ &= (-3,3)\end{aligned}$$



# Solutions

2. Find the magnitude of the vector (1,1)

$$|(1,1)| = \text{sqrt}(1^2 + 1^2)$$

$$= \text{sqrt}(2)$$

$$= 1.4$$

Magnitude is 1.4

# Solutions

3. Normalise the vector (8,6)

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$$

$$|(8,6)| = \text{sqrt}(8^2 + 6^2)$$

$$= \text{sqrt}(64+36)$$

$$= 10$$

Normalised vector is (8, 6) / 10

$$= (0.8, 0.6)$$



# Dot product

Definition:  $\mathbf{u} \cdot \mathbf{v} = u_1v_1 + u_2v_2 + \dots + u_nv_n$

Example:  $(1,2) \cdot (-1,3) = 1 \cdot -1 + 2 \cdot 3 = 5$

Properties:

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

$$(a\mathbf{u}) \cdot \mathbf{v} = a(\mathbf{u} \cdot \mathbf{v})$$

$$\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$$

$$\mathbf{u} \cdot \mathbf{u} = |\mathbf{u}|^2$$

# Angle between vectors

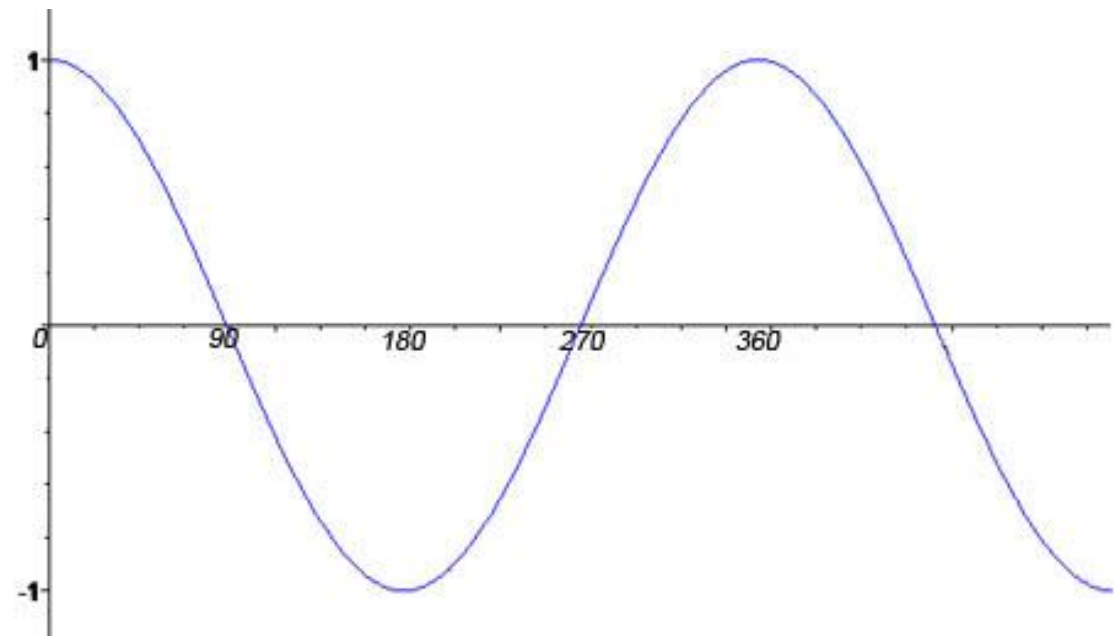
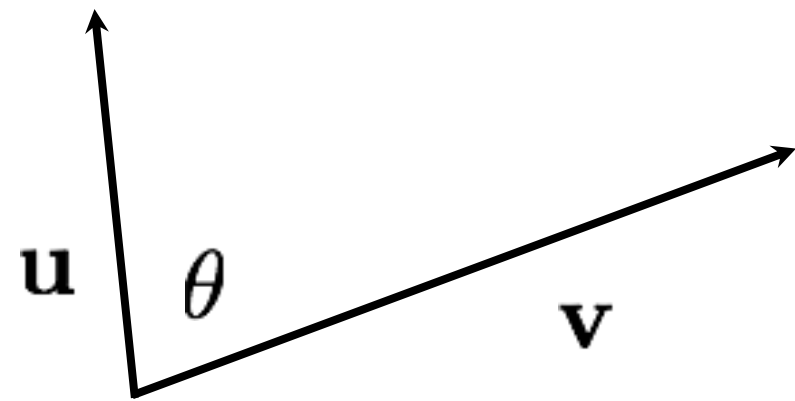
$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| |\mathbf{v}| \cos \theta$$

$$\cos \theta = \hat{\mathbf{u}} \cdot \hat{\mathbf{v}}$$

$$\mathbf{u} \cdot \mathbf{v} > 0 \implies \theta < 90^\circ$$

$$\mathbf{u} \cdot \mathbf{v} = 0 \implies \theta = 90^\circ$$

$$\mathbf{u} \cdot \mathbf{v} < 0 \implies \theta > 90^\circ$$



# Normals in 2D

If two vectors are perpendicular, their dot product is 0.

If  $n = (n_x, n_y)$  is a normal to

$$p = (x, y)$$

$$p \cdot n = x n_x + y n_y = 0$$

So either unless one is the 0 vector

$$n = (y, -x) \text{ or } n = (-y, x)$$

# Cross product

Only defined for 3D vectors:

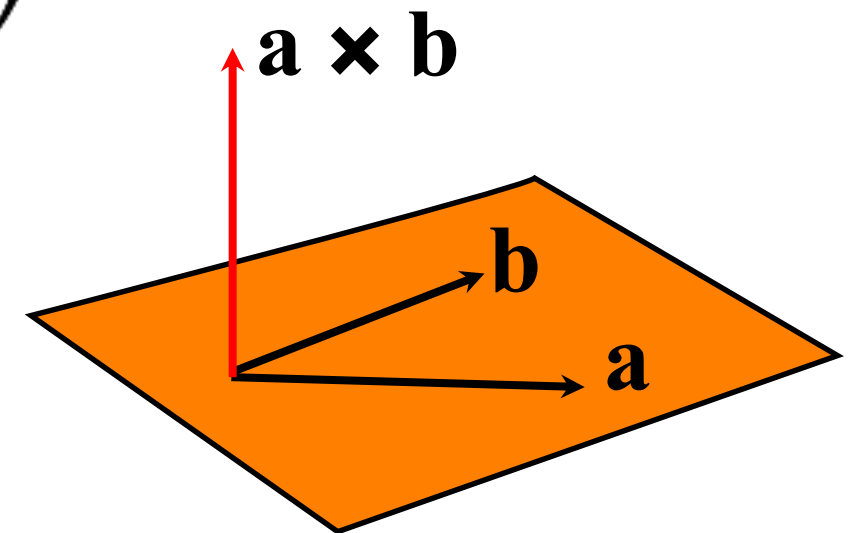
$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix}$$

Properties:

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$$

$$\mathbf{a} \cdot (\mathbf{a} \times \mathbf{b}) = 0$$

$$\mathbf{b} \cdot (\mathbf{a} \times \mathbf{b}) = 0$$



Can use to find normals

# **$\mathbf{a \times b}$ vs $\mathbf{b \times a}$**

Assume we have a right-handed coordinate system.

Curl the fingers of your right hand from  **$\mathbf{a}$**  to  **$\mathbf{b}$** .  **$\mathbf{a \times b}$**  will point in the direction of your thumb.

If you curl the fingers of your right hand from  **$\mathbf{b}$**  to  **$\mathbf{a}$**  you will get  **$\mathbf{b \times a}$**  which should point in the opposite direction to  **$\mathbf{a \times b}$** .

# Memory Aid

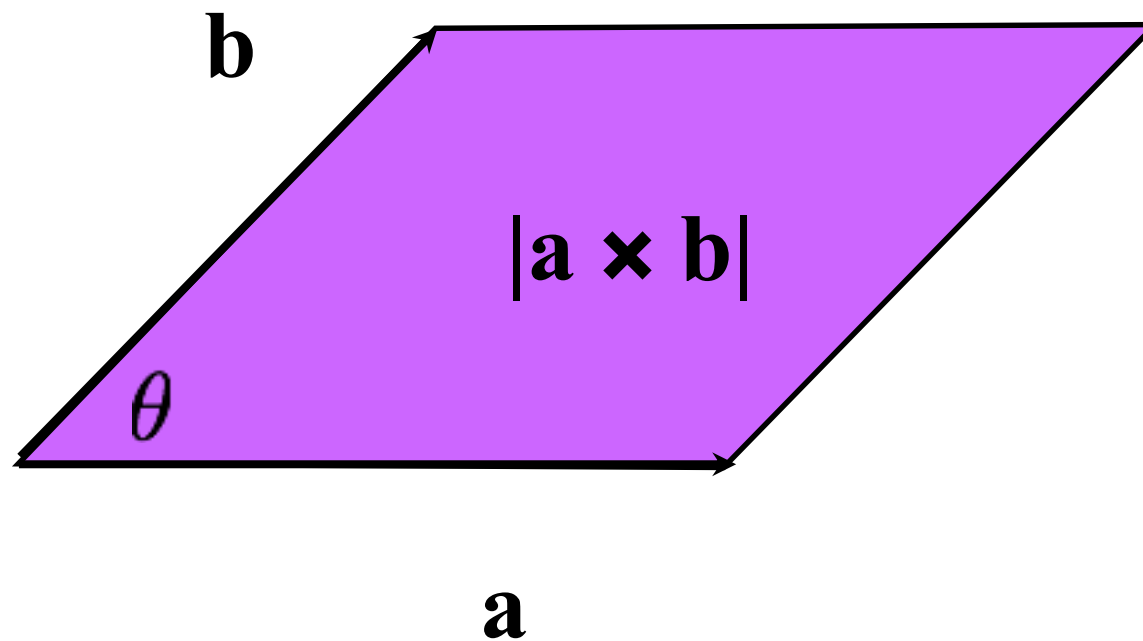
$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

$$= a_2b_3 - a_3b_2 + a_3b_1 - a_1b_3 + a_1b_2 - a_2b_1$$

# Cross product

The magnitude of the cross product is the area of the parallelogram formed by the vectors:

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin \theta$$



# Exercises

1. Find the angle between vectors  $(1,1)$  and  $(-1,-1)$
2. Is vector  $(3,4)$  perpendicular to  $(2,1)$ ?
3. Find a vector perpendicular to vector  $\mathbf{a}$  where  $\mathbf{a} = (2,1)$
4. Find a vector perpendicular to vectors  $\mathbf{a}$  and  $\mathbf{b}$  where  $\mathbf{a} = (3,0,2)$   $\mathbf{b} = (4,1,8)$



# Solutions

1. Find the angle between vectors  $(1,1)$  and  $(-1,-1)$

$$\cos \theta = \hat{\mathbf{u}} \cdot \hat{\mathbf{v}}$$

$$|(1,1)| = \sqrt{2}$$

$$|(-1,-1)| = \sqrt{2}$$

$$\begin{aligned}\cos(t) &= (1/\sqrt{2}, 1/\sqrt{2}) \cdot (-1/\sqrt{2}, -1/\sqrt{2}) \\ &= -1\end{aligned}$$

$t = 180$  degrees (ie anti-parallel)

# Solutions

2. Is  $(3,4)$  perpendicular to  $(2,1)$ ?

$$(3,4) \cdot (2,1) = 6 + 4 = 10$$

$10 \neq 0$  so not perpendicular (  $< 90$ degrees)

3. Find a vector perpendicular to vector **a**  
where **a** =  $(2,1)$

$(-1,2)$  or  $(1,-2)$

# Solutions

4. Find a vector perpendicular to vectors **a** and **b** where **a** = (3,0,2) **b** = (4,1,8)

$$\mathbf{a} \times \mathbf{b} = (0-2, 8-24, 3-0)$$

$$= (-2, -16, 3) \text{ OR } \mathbf{b} \times \mathbf{a} = (2, 16, -3)$$

# Homework

Revise matrix multiplication for next week.