5. 2D Geometry

In order to design and render complex scenes we require techniques for transforming points and vectors. Points are used to represent OpenGL primitives (`glVertex`) and vectors are used to represent surface normals (necessary for computing the illumination at a point).

5.1 Points and Vectors

5.2 Applications of the Scalar Product (Dot Product)

5.3 Convex and Concave Objects

5.4 Implicit Curves

5.5 Parametric Curves

5.6 2D Affine Transformations

5.7 2D Homogeneous Coordinates

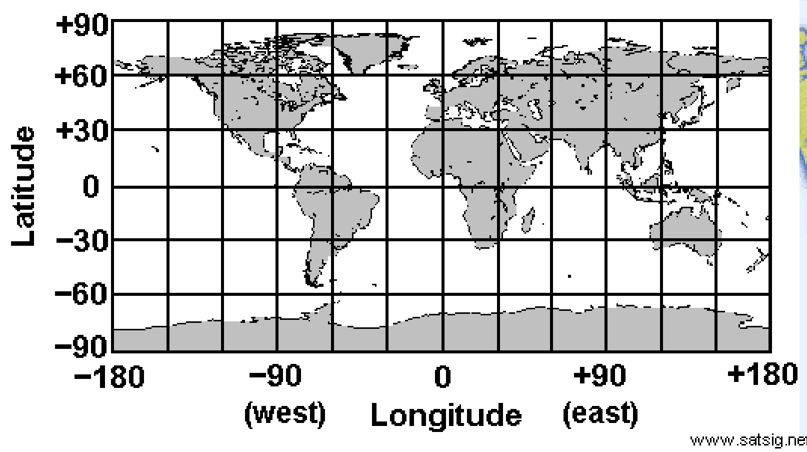
5.8 Notes & Examples

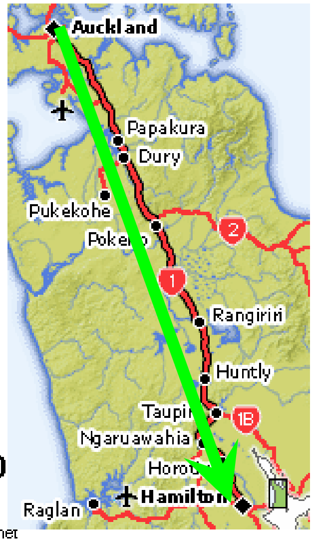
5.1. Points and Vectors

- A ***point*** is a position in space, e.g. Auckland
- A ***vector*** represents a displacement – a *difference* between two points.
- The only way to represent a point is with reference to the origin of a coordinate system. The vector from the origin of the coordinate system to the point is the ***position vector*** of the point.

Example: Describe where Hamilton is!

- 120km to the south-south-west of Auckland
- 37.43S Latitude, 175.19E Longitude



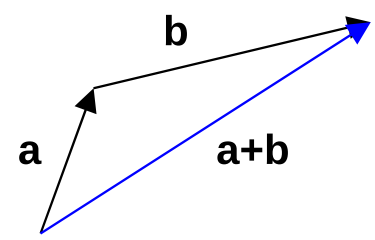


Points and Vectors (cont'd)

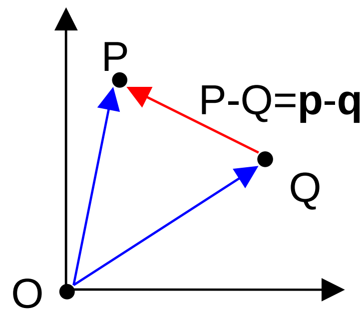
- Vectors are represented as 2-tuples (2D) or 3-tuples (3D) in a coordinate system.
- We denote the components of a vector **v** in 2D with v_1 and v_2 and of a vector **u** in 3D with u_1, u_2 and u_3:
- We denote vectors with small bold letters and points with capital letters, e.g. **p** is the position vector of the point P.

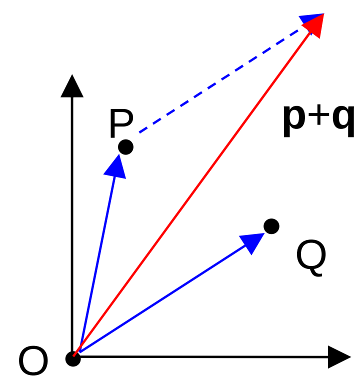
$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \qquad \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

Points and Vectors (cont'd)



- Operations on vectors
  - Can add, subtract and scale vectors
- Operations on points
  - Subtracting one point from another gives a vector (the displacement between these points)
  - Can **NOT** add two points (what is Auckland + Hamilton??)
  - But we can add and subtract their position vectors w.r.t. some origin.





Basic Operations on Vectors

- Addition
  - Represents the combined displacement
  - Implement by adding components

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

- Scaling
  - i.e. multiplication by a scalar
  - Defined such that **v** + **v** = 2**v**
  - Implement by multiplying all components by the scalar.

$$s\ \mathbf{u}=s\begin{pmatrix}u_1\\u_2\end{pmatrix}=\begin{pmatrix}s\ u_1\\s\ u_2\end{pmatrix}$$

- Subtraction
  - Addition of a negated vector, i.e. one in opposite direction.
  - Implement by subtracting components.

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

- The magnitude of a vector
  - i.e. its "length" (2-norm).

$$|\mathbf{u}|=\sqrt{u_1^{\,2}+u_2^{\,2}}\quad,\ |s\mathbf{u}|=|s||\mathbf{u}|$$

- Normalisation
  - The process of creating a unit vector (length 1)
  - Scale by reciprocal of magnitude:

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

Basic Operations on Matrices

- Dimension of a matrix
  - A *m*×*n* matrix is a matrix has *m* rows and *n* columns.

$$\begin{array}{ll} \mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \end{pmatrix} & \text{row} \\ \text{column} & \end{array}$$

Example of a 2 ×3 matrix

- Addition/Subtraction
  - Implement by adding/subtracting components.

$$\mathbf{M} \pm \mathbf{N} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \pm \begin{pmatrix} n_{11} & n_{12} \\ n_{21} & n_{22} \end{pmatrix} = \begin{pmatrix} m_{11} \pm n_{11} & m_{12} \pm n_{12} \\ m_{21} \pm n_{21} & m_{22} \pm n_{22} \end{pmatrix}$$

- Scaling
  - Implement by multiplying all components by the scalar.

$$s\mathbf{M} = \begin{pmatrix} s\ m_{11} & s\ m_{12} \\ s\ m_{21} & s\ m_{22} \end{pmatrix}$$

Basic Operations on Matrices (cont'd)

- Transpose (indicated by ^{T}) of a matrix **M**
  - Swap m_{ij} and m_{ji} for all i,j.

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{pmatrix} \Rightarrow \mathbf{M}^T = \begin{pmatrix} m_{11} & m_{21} \\ m_{12} & m_{22} \\ m_{13} & m_{23} \end{pmatrix}$$

Algebraic rules for transposition:

$(\mathbf{M}^T)^T = \mathbf{M}$

$(s\mathbf{M})^T = s(\mathbf{M}^T)$

$(\mathbf{M}+\mathbf{N})^T = \mathbf{M}^T + \mathbf{N}^T$

$(\mathbf{MN})^T = \mathbf{N}^T\mathbf{M}^T$

Basic Operations on Matrices (cont'd)

- Determinant *det* **M** (also written | **M** |) of a matrix **M**
  - For a 2x2 matrix:

$$\begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

- Inverse $\mathbf{M}^{-1}$ of a matrix **M**
  - For a 2x2 matrix:

$$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}^{-1} = \frac{1}{m_{11}m_{22} - m_{12}m_{21}} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix}$$

Exercise: Prove that $\mathbf{M}^{-1}$ is the inverse of **M**, i.e. show $\mathbf{M}^{-1}\mathbf{M}=\mathbf{M}\mathbf{M}^{-1}=\mathbf{I}$

Basic Operations on Vectors and Matrices

- The transpose of a vector
  - Transpose of a row vector is a column vector and vice versa

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \Rightarrow \mathbf{u}^T = \begin{pmatrix} u_1 & u_2 \end{pmatrix}$$

- The dot product (scalar product)
  - Symmetry: **a•b=b•a**
  - Linearity: (**a**+**b**)•**c**=**a**•**c**+**b**•**c**
  - Homogeneity: (s**a**)•**b**=s(**a**•**b**)
  - $|b|^2$= **b•b**

$$\mathbf{u} \bullet \mathbf{v} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \bullet \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = u_1 v_1 + u_2 v_2 = \mathbf{u}^T \mathbf{v}$$

- Matrix multiplication
  - Multiplying an $l \times m$ and $m \times n$ matrix gives an $l \times n$ matrix with the elements $a_{ij} = b_{i1} c_{1j} + \ldots + b_{im} c_{mj} = \sum_{k=1}^{m} b_{ik} c_{kj}$ [Note: a_{ij}= row_i•column_j]

$$\mathbf{A} = \mathbf{B}\,\mathbf{C} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} b_{11}c_{11} + b_{12}c_{21} & b_{11}c_{12} + b_{12}c_{22} \\ b_{21}c_{11} + b_{22}c_{21} & b_{21}c_{12} + b_{22}c_{22} \end{pmatrix}$$

5.2 Applications of the Scalar Product (Dot Product)

- Angle between two vectors
- Projection of a vector
- Distance of a point to a line
- Reflections
- Area of a triangle

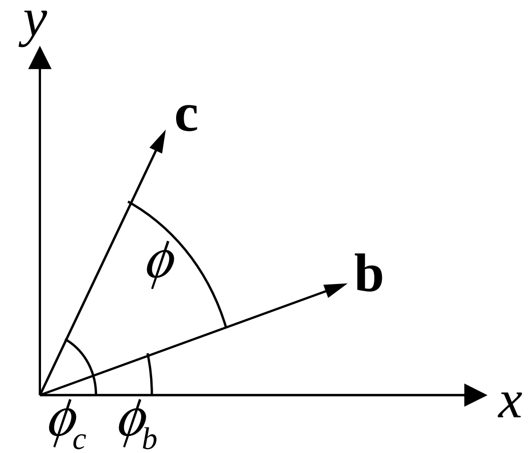
The Angle between Two Vectors

- The most important application of the dot product is to find the angle between two vectors (or two intersecting lines).

$$\mathbf{b} = \begin{pmatrix} |b|\cos\phi_b \\ |b|\sin\phi_b \end{pmatrix}, \qquad \mathbf{c} = \begin{pmatrix} |c|\cos\phi_c \\ |c|\sin\phi_c \end{pmatrix}$$

hence

$$\begin{aligned} \mathbf{b} \bullet \mathbf{c} &= |b||c|\cos\phi_b\cos\phi_c + |b||c|\sin\phi_b\sin\phi_c \\ &= |b||c|\cos(\phi_c - \phi_b) \\ &= |b||c|\cos\varphi \end{aligned}$$



Two non-zero vectors **b** and **c** with common start point are

less than	90^o apart	if **b•c**>0	
exactly	90^o apart	if **b•c**=0	[**b** and **c** are *orthogonal* (*perpendicular*)]
more than	90^o apart	if **b•c**<0	

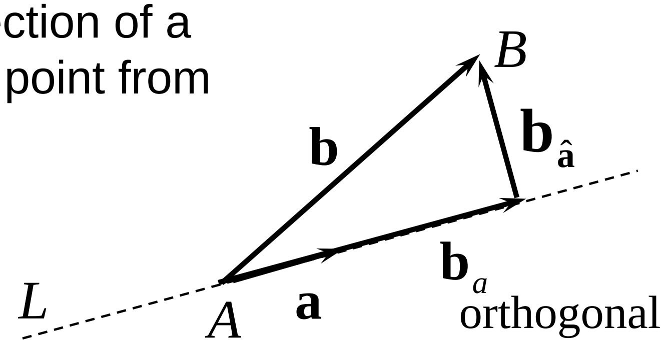
Projection of a Vector

- In many applications we must compute the projection of a vector onto another vector and the distance of a point from a line:

  Let *L* be a line through *A* in the direction of **a**.

  Let **b** be the vector from *A* to a point *B*.

  We want to find $\mathbf{b}_\mathbf{a}$ the orthogonal projection of **b** onto **a**.



$$\mathbf{b} = \mathbf{b}_\mathbf{a} + \mathbf{b}_{\hat{\mathbf{a}}} = k\mathbf{a} + \mathbf{b}_{\hat{\mathbf{a}}} \text{ for some } k$$

$$\mathbf{b} \bullet \mathbf{a} = (k\mathbf{a} + \mathbf{b}_{\hat{\mathbf{a}}}) \bullet \mathbf{a} = k\mathbf{a} \bullet \mathbf{a} + \mathbf{b}_{\hat{\mathbf{a}}} \bullet \mathbf{a} = k(\mathbf{a} \bullet \mathbf{a})$$

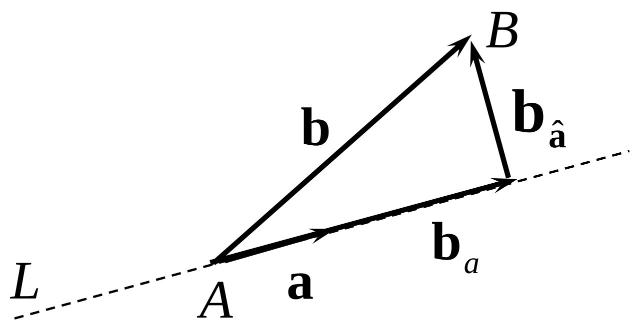
$$\Rightarrow k = \frac{\mathbf{b} \bullet \mathbf{a}}{\mathbf{a} \bullet \mathbf{a}}$$

$$\Rightarrow \mathbf{b}_\mathbf{a} = \frac{\mathbf{b} \bullet \mathbf{a}}{\mathbf{a} \bullet \mathbf{a}} \mathbf{a}$$

Note: at this point we don't know the value for $\mathbf{b}_{\hat{\mathbf{a}}} = \mathbf{b} - \mathbf{b}_\mathbf{a}$ but we know it is perpendicular to **a**.

Distance of a Point to a Line

- In the previous slide we computed $\mathbf{b}_\mathbf{a} = \frac{\mathbf{b} \bullet \mathbf{a}}{\mathbf{a} \bullet \mathbf{a}} \mathbf{a}$



hence the distance of B to the line L is

$$\left|\mathbf{b}_{\hat{\mathbf{a}}}\right| = \left|\mathbf{b} - \mathbf{b}_\mathbf{a}\right| = \left|\mathbf{b} - \frac{\mathbf{b} \bullet \mathbf{a}}{\mathbf{a} \bullet \mathbf{a}} \mathbf{a}\right|$$

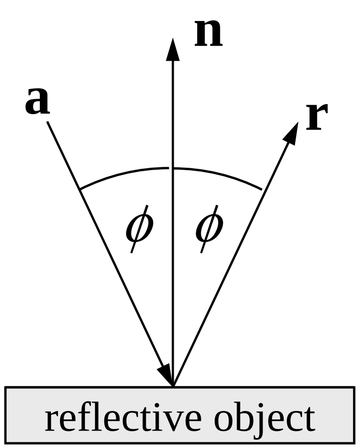
Reflections

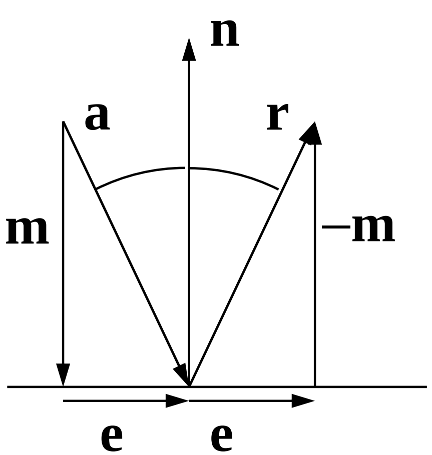
- Ray tracing is a popular rendering algorithm which displays a scene by tracing rays from the eye through each pixel of the screen into the scene (i.e. trace a light ray hitting the eye backwards → see 2nd part of this course!). If the scene contains reflective objects such as mirrors it is necessary to compute for a ray with direction **a** its reflection **r**.
  Let **n** be the surface normal at the point where the ray hits the object:

$$\mathbf{m} = \frac{\mathbf{a} \bullet \mathbf{n}}{\mathbf{n} \bullet \mathbf{n}} \mathbf{n} = \frac{\mathbf{a} \bullet \mathbf{n}}{|\mathbf{n}|^2} \mathbf{n} = (\mathbf{a} \bullet \hat{\mathbf{n}}) \hat{\mathbf{n}}$$

$$\Rightarrow \mathbf{r} = \mathbf{e} - \mathbf{m} = (\mathbf{a} - \mathbf{m}) - \mathbf{m} = \mathbf{a} - 2\mathbf{m}$$

$$= \mathbf{a} - 2(\mathbf{a} \bullet \hat{\mathbf{n}}) \hat{\mathbf{n}}$$

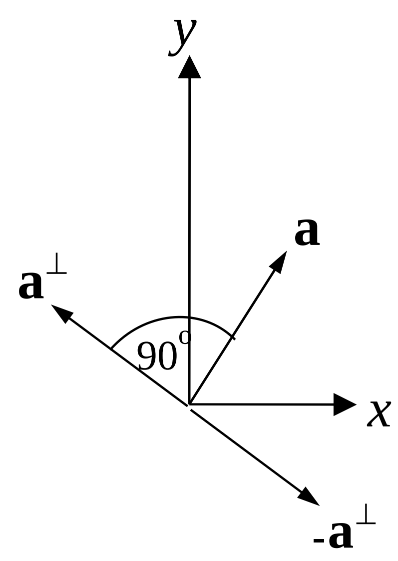




Orthogonal Vectors in 2D

- Let $\mathbf{a}=(a_1 , a_2)^T$ then in 2D we can find two vectors perpendicular to it
  - $\mathbf{a}^{\perp}=(-a_2 , a_1)^T$ (note $\mathbf{a}^{\perp}\bullet \mathbf{a=0}$)
  - $-\mathbf{a}^{\perp}=(a_2 , -a_1)^T$ (note $-\mathbf{a}^{\perp}\bullet \mathbf{a=0}$)

In the textbook $\mathbf{a}^{\perp}$ is called the “Perp” vector



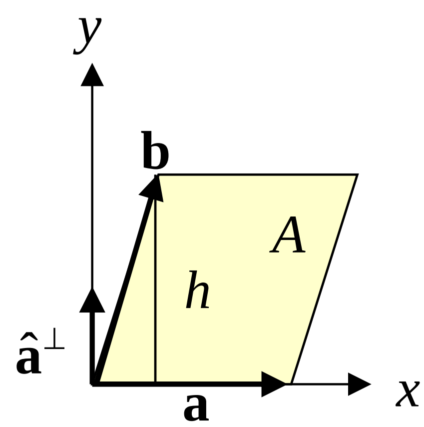
The Area of a Triangle (in 2D)

- The area of a parallelogram:

$$A = |\mathbf{a}| h$$

$$h = \left|\mathbf{b}_{\mathbf{a}^\perp}\right| = \left| \frac{\mathbf{a}^\perp \bullet \mathbf{b}}{\mathbf{a}^\perp \bullet \mathbf{a}^\perp} \mathbf{a}^\perp \right| = \frac{|\mathbf{a}^\perp \bullet \mathbf{b}|}{|\mathbf{a}|}$$

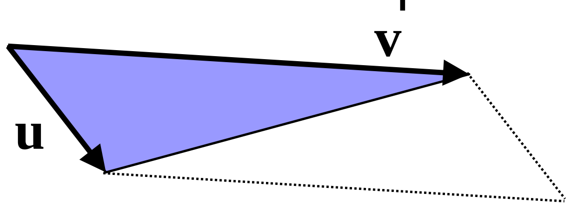
$$A = |\mathbf{a}| h = |\mathbf{a}| \frac{|\mathbf{a}^\perp \bullet \mathbf{b}|}{|\mathbf{a}|} = |\mathbf{a}^\perp \bullet \mathbf{b}|$$



NOTE: These formulas are only valid in 2D!!

Area of a triangle:

- Area is half the area of the parallelogram formed by two of its edges

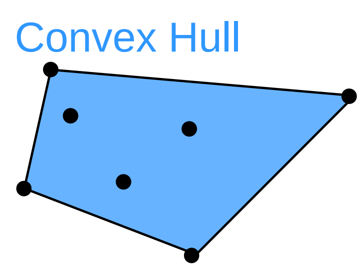


$$A = \frac{1}{2} |\mathbf{u}^\perp \bullet \mathbf{v}|$$

5.3 Convex and Concave Objects

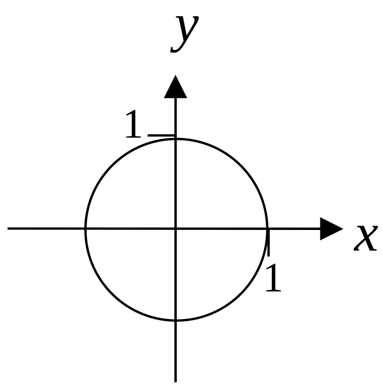
- A *Convex Polygon* is a polygon where any line connecting any pair of vertices lies entirely within the polygon (this is equivalent with: all interior angles between neighbouring edges are smaller or equal to 180 degree). If a polygon is not convex it is called concave.

- The *Convex Hull* of a set of points is the smallest convex set containing the points. [i.e. smallest convex polygon containing the points]



5.4 Implicit Curves

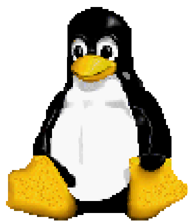
- Implicit curves
  - A 2D curve can be defined as the set of points $p=(x,y)^T$ fulfilling the mathematical equation $f(x,y)=0$.



Example:

$$x^2+y^2-1=0$$

defines a unit circle centred at the origin



Disadvantages:

- Modelling is non-intuitive (e.g. how to draw a penguin?)
- Difficult to draw: have to find a set of points fulfilling the equation (hard!) and connect them by line segments.

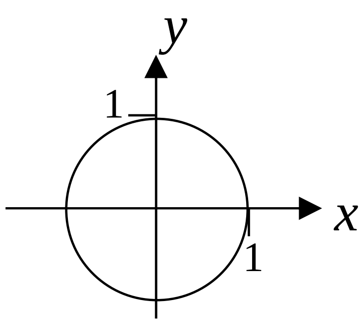
Advantages:

- Easy to compute normal **n** at a point $(x_0,y_0)^T$:

$$\mathbf{n} = \left(\frac{\partial f}{\partial x} \quad \frac{\partial f}{\partial y} \right)^T \Bigg|_{(x_0, y_y)}$$

5.5 Parametric Curves

- Parametric curves
  - A 2D curve defined by a set of points $\mathbf{p}(t)=(x(t), y(t))^T$ where $x(t)$ and $y(t)$ are functions of the parameter t (often called the "speed").
  - Have to specify the parameter interval $[t_{min}, t_{max}]$ for t.
    It's a good idea to specify curve such that $[t_{min}, t_{max}]=[0,1]$.



Example: $$p(t) = \begin{pmatrix} \cos 2\pi t \\ \sin 2\pi t \end{pmatrix} \quad , \quad t \in [0,1]$$

defines a unit circle centred at the origin.

Disadvantages:
- Modelling is still non-intuitive (e.g. how to draw a penguin?)

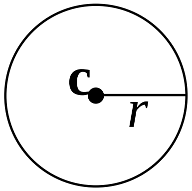
Advantages
- Can compute tangent at a point by the derivative of the components $\mathbf{p}'(t) = \begin{pmatrix} \frac{dx}{dt} & \frac{dy}{dt} \end{pmatrix}^T$
- Easy to "splice" curve segments together.
- Can draw curve by computing points on the curve and connecting them by line segments.

Parametric Curves (cont'd)

- Examples for parametric curves (in all cases $t \in [0,1]$):

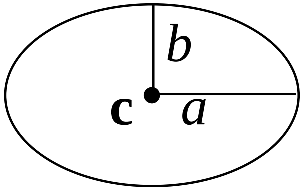
Curve with centre **c** and radius *r*:

$$p(t) = \begin{pmatrix} c_1 + r\cos(2\pi t) \\ c_2 + r\sin(2\pi t) \end{pmatrix}$$



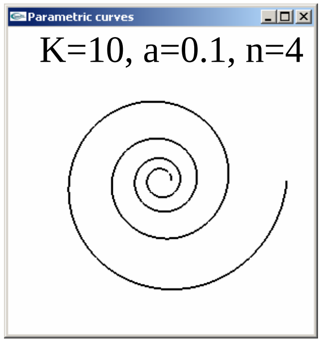
Ellipsoid with centre **c,** long axis *a* and short axis *b*:

$$p(t) = \begin{pmatrix} c_1 + a\cos(2\pi t) \\ c_2 + b\sin(2\pi t) \end{pmatrix}$$



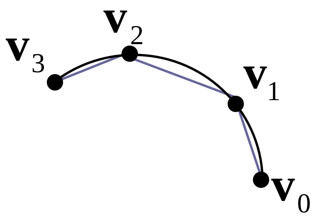
Logarithmic spiral with centre **c** and *n* revolutions:

$$p(t) = \begin{pmatrix} c_1 + f(\theta)\cos(\theta) \\ c_2 + f(\theta)\sin(\theta) \end{pmatrix} \text{ where } f(\theta) = Ke^{a\theta}\ ,\ \theta = 2\pi nt$$



Parametric Curves (cont'd)

- How to draw a parametric curve?
  - Compute (n+1) points $\mathbf{v}_i = \mathbf{p}\left(\frac{i}{n}\right)$ for $i = 0, \ldots, n$ on the curve
  - Connect the points with line segments



G:\...\ParametricCurve.h

```
// A virtual class for a 2D parametric curve
// No instances of this class can be constructed
class CParametricCurve
{
public:
    CParametricCurve(){}
    virtual ~CParametricCurve(){}
    void draw();                // draws curve as a line strip
    virtual void computePointOnCurve(float t, float& x, float& y)=0;
                                // computes curve point p(t)=(x(t), y(t))
protected:
    float vertices[n+1][2];
    void init();                // computes n vertices on the curve
};
```

Parametric Curves (cont'd)

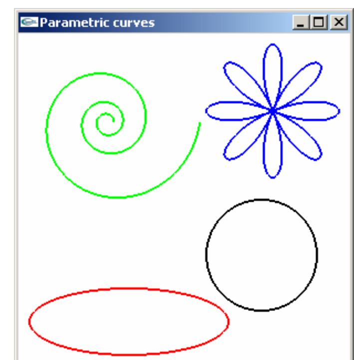
G:\...\ParametricCurve.h

```
class CParametricCircle:public CParametricCurve
{
public:
    CParametricCircle();
    CParametricCircle(float centreX, float centreY, float radius);
    virtual ~CParametricCircle(){}
    void computePointOnCurve(float t, float& x, float& y);
private:
    float cx,cy;     // x-coordinate and y-coordinate of the centre
    float r;         // radius
};
```

```
void CParametricCurve::init(){            // compute (n+1) points on the curve
  for(int i=0;i<=n;i++)
    computePointOnCurve((float) i/(float) n,vertices[i][0],vertices[i][1]);}

void CParametricCurve::draw(){            // draw line segments
  glBegin(GL_LINE_STRIP);
    for(int i=0;i<=n;i++) glVertex2fv(vertices[i]);
  glEnd();}

void CParametricCircle::computePointOnCurve(float t, float& x, float& y){
  x=cx+r*cos(t*2.0*PI);
  y=cy+r*sin(t*2.0*PI);}
```



5.6 2D Affine Transformations

Affine Transformations transform a pair of parallel straight lines to another pair of parallel straight lines and preserve ratios of distances. Assume for now that the transformations apply only to *points* but with an origin and an underlying vector space defined.

Examples of affine transformations:

- Scaling about Origin
  - For any point $\mathbf{p} = (p_1, p_2)^T$, scale p_1 by factor s_1, p_2 by factor s_2.

$$\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}$$

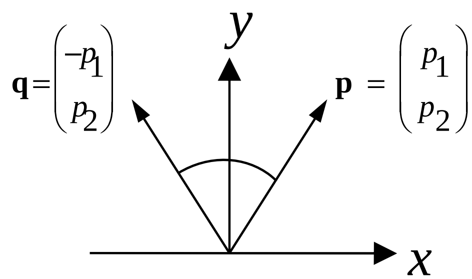
i.e. $\mathbf{q} = \mathbf{M}_{scale}\mathbf{p}$

where $\mathbf{M}_{scale} = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix}$

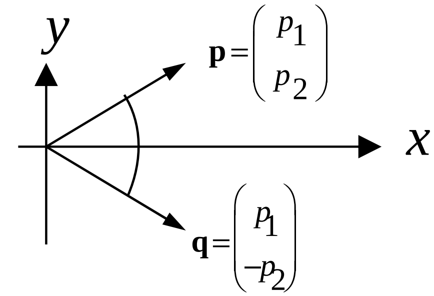
- Translation ("movement")
  - Add a vector **t** to all points in the scene, i.e. **q** = **p** + **t**

2D affine transformations (cont'd)

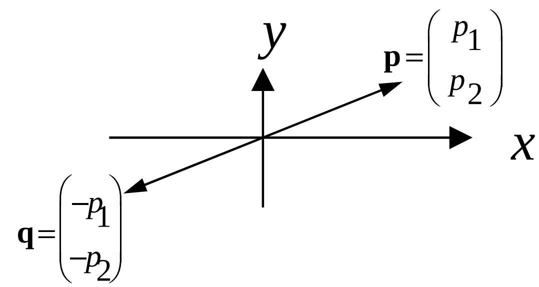
- Note that "scaling" includes "reflection" if s_1 and/or s_2 is negative:
  - Reflection at the y-axis: $\mathbf{q} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{p}$



  - Reflection at the x-axis $\mathbf{q} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \mathbf{p}$

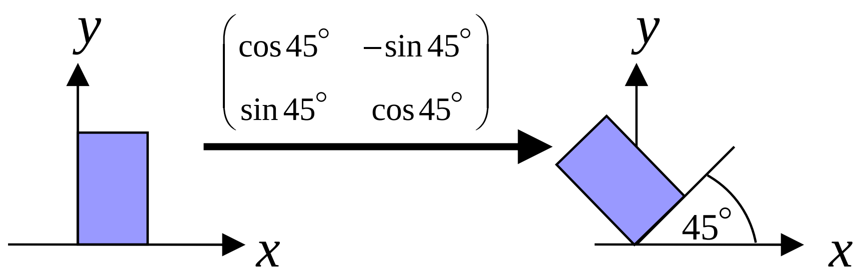


  - Reflection at the origin $\mathbf{q} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \mathbf{p}$



2D affine transformations (cont'd)

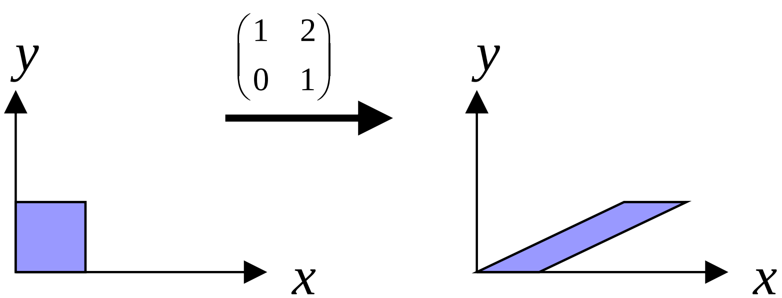
- Rotation about origin by an angle θ (right-handed i.e. anticlockwise)
  - Proof: UDOO



$$\mathbf{q} = \mathbf{M}_{rotate}\mathbf{p}$$

$$\text{where } \mathbf{M}_{rotate} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

- Shearing



$$\mathbf{q} = \mathbf{M}_{shear}\mathbf{p}$$

$$\text{where } \mathbf{M}_{shear} = \begin{pmatrix} 1 & xShear \\ yShear & 1 \end{pmatrix}$$

Some properties of affine transformations (both 2D & 3D)

- Straight lines are preserved
- Parallel lines remain parallel
- Proportional distances are preserved
- Any closed *area* in 2D or *volume* in 3D is multiplied by $|\det M|$ (unchanged by translation)
- Any arbitrary affine transformation can be represented as a sequence of shearing, scaling, rotation and translation
- **Affine transformations do not in general commute (i.e. $\mathbf{T_1 T_2 \neq T_2 T_1}$)**
- Transformations *are* associative, i.e. $\mathbf{T}_1(\mathbf{T}_2\mathbf{T}_3) = (\mathbf{T}_1\mathbf{T}_2)\,\mathbf{T}_3$

5.7 2D Homogeneous Coordinates

Translation is a nuisance - don't have a matrix representation for it.

So we introduce *homogeneous coordinates* as a way of "unifying" the representation of translation with the other transformations.

- The idea
- Geometric interpretation
- Converting from HC to ordinary coordinates
- Composition of transformations

The idea

Represent the ordinary 2D point $(x, y)^T$ as a homogeneous coordinate point $(x, y, 1)^T$.

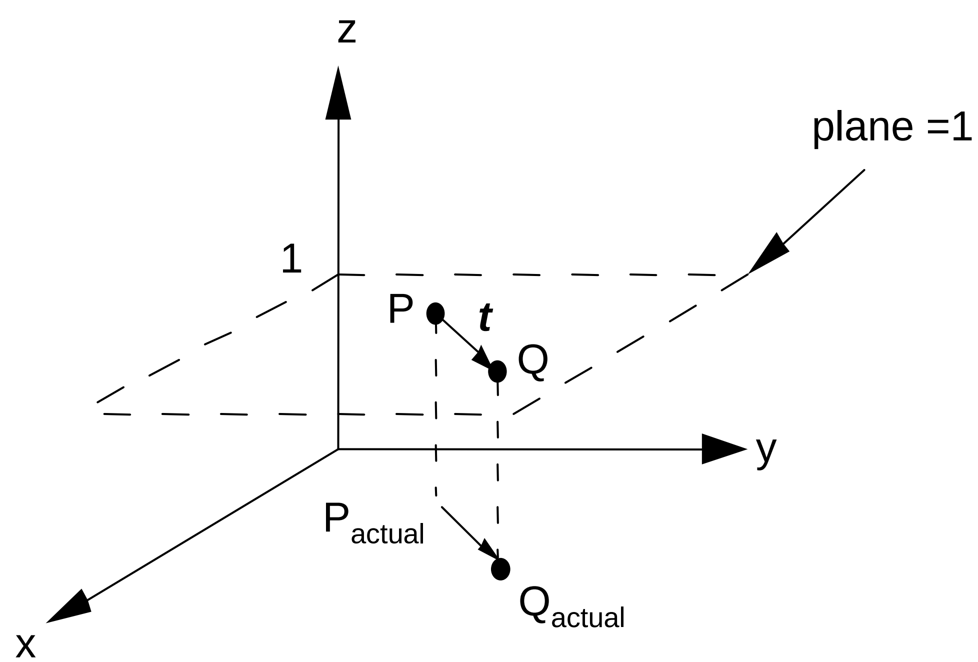
Then can do translation by:

$$\begin{pmatrix} q_x \\ q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

and the other transformations by

$$\begin{pmatrix} q_x \\ q_y \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

Geometric interpretation



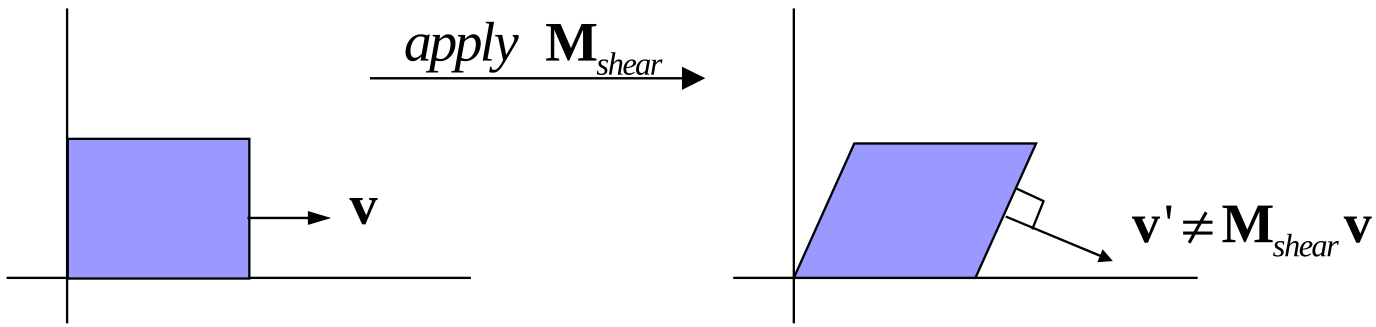
Can see that the translation in 2D is implemented as a shear in 3D.

Converting from HC to Ordinary Coordinates

- More generally, we regard all homogenous coordinate points $(w\, p_1, w\, p_2, w)^T$, $w \neq 0$, as representing the same ordinary coordinate point $(p_1, p_2)^T$.
- Hence, in general, the homogeneous coordinate point $(a,b,\mathrm{c})^T$ converts to the ordinary coordinate point $(a/c, b/c)^T$.
- With all the transformations so far, c will equal 1, but we will see a couple of examples later where this is not the case.

5.8 Notes & Examples

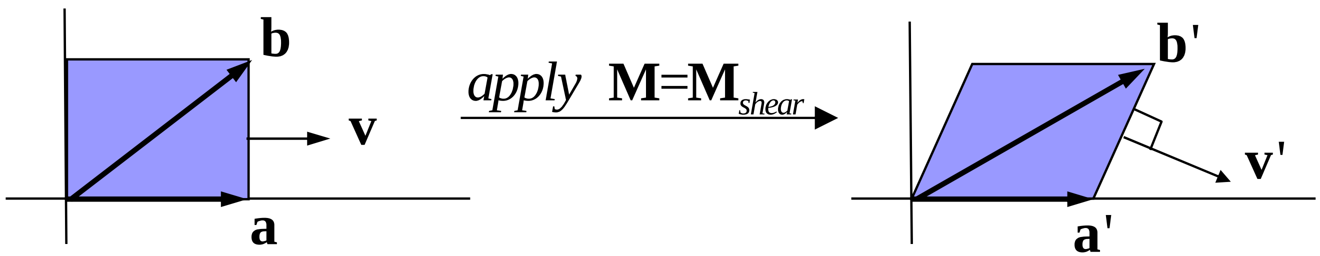
Be careful when transforming vectors. ***Doesn't work for position vectors or surface normals*** (i.e. vectors perpendicular to given surfaces).



$$\mathbf{v} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \mathbf{M}_{\text{shear}} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \Rightarrow \mathbf{M}_{\text{shear}}\mathbf{v} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \neq \mathbf{v}'$$

Transforming Vectors (cont'd)

- How do we find the transformed surface normal **v**' ?



$\mathbf{v}$ is perpendicular to $(\mathbf{b}-\mathbf{a}) \Leftrightarrow (\mathbf{b}-\mathbf{a}) \bullet \mathbf{v} = 0 \Leftrightarrow (\mathbf{b}-\mathbf{a})^T \mathbf{v} = 0$

$\mathbf{v}'$ must be perpendicular to $(\mathbf{b}'-\mathbf{a}') \Leftrightarrow (\mathbf{b}'-\mathbf{a}') \bullet \mathbf{v}' = 0 \Leftrightarrow (\mathbf{b}'-\mathbf{a}')^T \mathbf{v}' = 0$

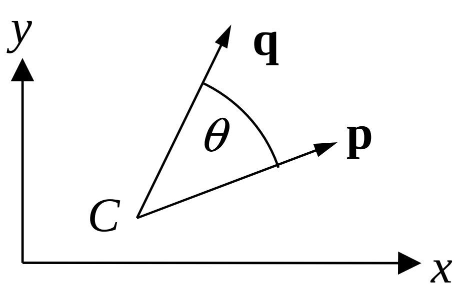
$$(\mathbf{b}'-\mathbf{a}')^T \mathbf{v}' = (\mathbf{Mb}-\mathbf{Ma})^T \mathbf{v}' = \left(\mathbf{M}(\mathbf{b}-\mathbf{a})\right)^T \mathbf{v}' = (\mathbf{b}-\mathbf{a})^T \mathbf{M}^T \mathbf{v}'$$

Choose $\mathbf{v}' = \left(\mathbf{M}^T\right)^{-1} \mathbf{v}$ then $(\mathbf{b}'-\mathbf{a}')^T \mathbf{v}' = (\mathbf{b}-\mathbf{a})^T \mathbf{M}^T \left(\mathbf{M}^T\right)^{-1} \mathbf{v} = (\mathbf{b}-\mathbf{a}) \bullet \mathbf{v} = 0$

$$\boxed{\mathbf{v}' = \left(\mathbf{M}^T\right)^{-1} \mathbf{v} = \left(\mathbf{M}^{-1}\right)^T \mathbf{v} \text{ for any surface normal } \mathbf{v} \text{ and linear transf. } \mathbf{M}}$$

Composition of transformations

- With homogeneous coordinates, it's now much easier to compose multiple transformations into a single one.
- Consider for example the problem of rotating some object about its centre point C.
  - translate the whole object so that its centre is at the origin, rotate about the origin, and then translate back.
  - Hence, transformation is $(q_1 \quad q_2 \quad 1)^T = \mathbf{M}(p_1 \quad p_2 \quad 1)^T$



where

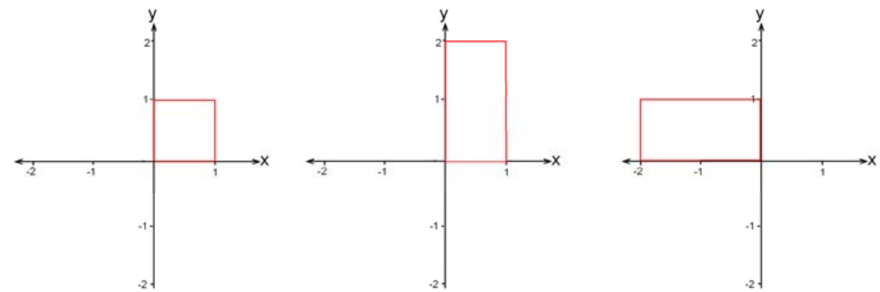
$$\mathbf{M} = \begin{pmatrix} 1 & 0 & c_1 \\ 0 & 1 & c_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -c_1 \\ 0 & 1 & -c_2 \\ 0 & 0 & 1 \end{pmatrix}$$

  - Can multiply the three component matrices to get the composite transformation matrix M, and then apply M to all points in the object.
  - UDOO: Work out **M** – show that it is equivalent to a rotation of θ followed by a single (different) translation.

Example 1

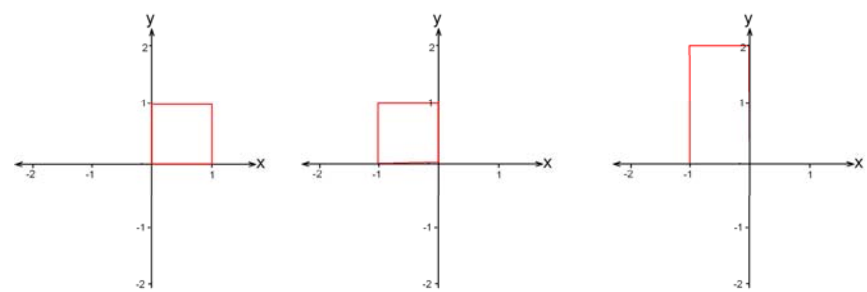
- In general affine transformations do not commute:
  - First scale by (1,2), then rotate 90°

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



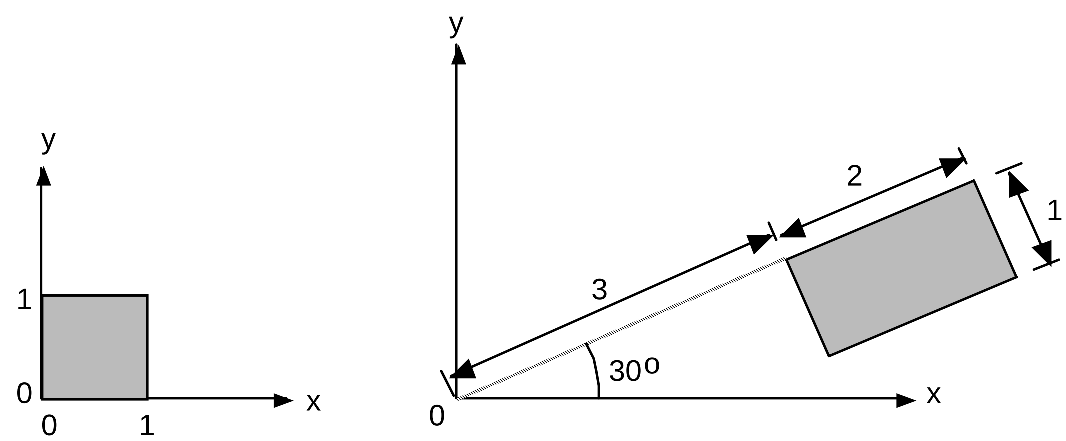
  - First rotate 90° then scale by (1,2)

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



Example 2

- Find the homogenous coordinate transformation matrix that transforms the figure on the left to the figure on the right



- Often easier to do these backwards, then take inverse. In this case, starting with figure on right:
  - Rotate -30º , shift by (-3,1) , scale by (1/2, 1)
- Hence required transformation from right to left is:
  - R(30) T(3,-1) S(2,1)
  - Easy to convert to HC matrix expression [UDOO]

Example 3

- Given is the 2D scene in part (a) of the image below. Write down the **homogeneous 2D transformation matrix M**, which transforms the object shown in (a) into the object in part (b) of the image. You are allowed to write the transformation matrix as a product of simpler matrices (i.e. you are not required to multiply the matrices).