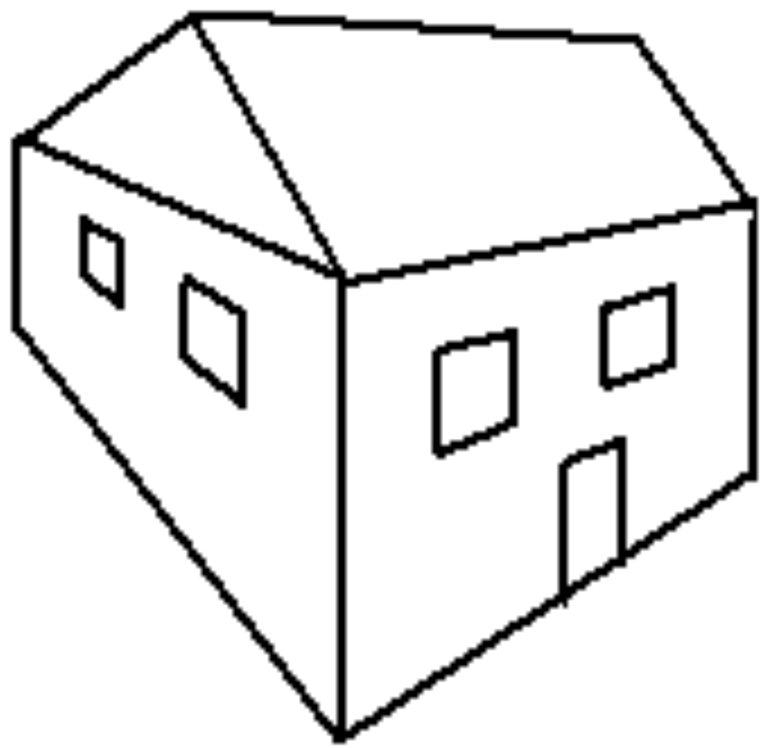
Computer Graphics

COMP H3016

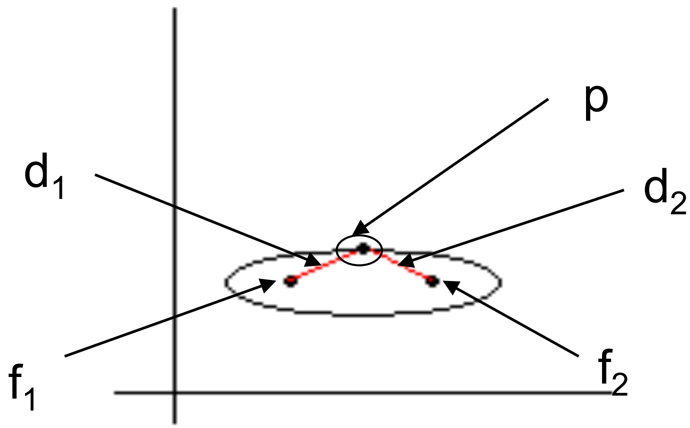
Lecturer: Simon McLoughlin

Lecture 2



Output primitives continued - Ellipses

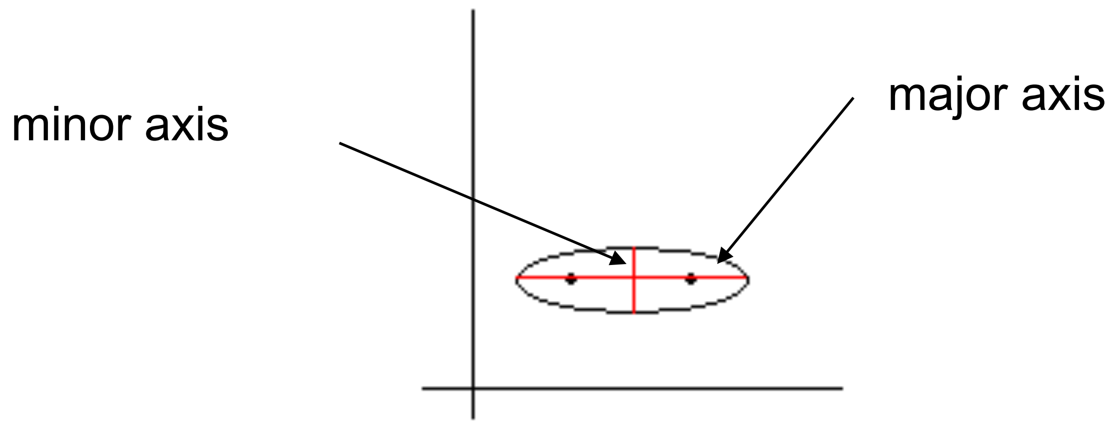
- Informally, an ellipse is an **elongated circle**
- **They are defined as the set of points who's distance from two points in the ellipse called the foci is constant when summed together**



- In the diagram above the ellipse is defined as all points in the x-y plane who's distance from f_1 plus the distance from f_2 is equal to $d_1 + d_2$

Output primitives - Ellipses

- A line through the two foci f1, and f2 from one side of the ellipse to the other is called the **major axis** of the ellipse
- A line orthogonal and through the midpoint of the major axis from one ellipse side to the other is called the **minor axis**

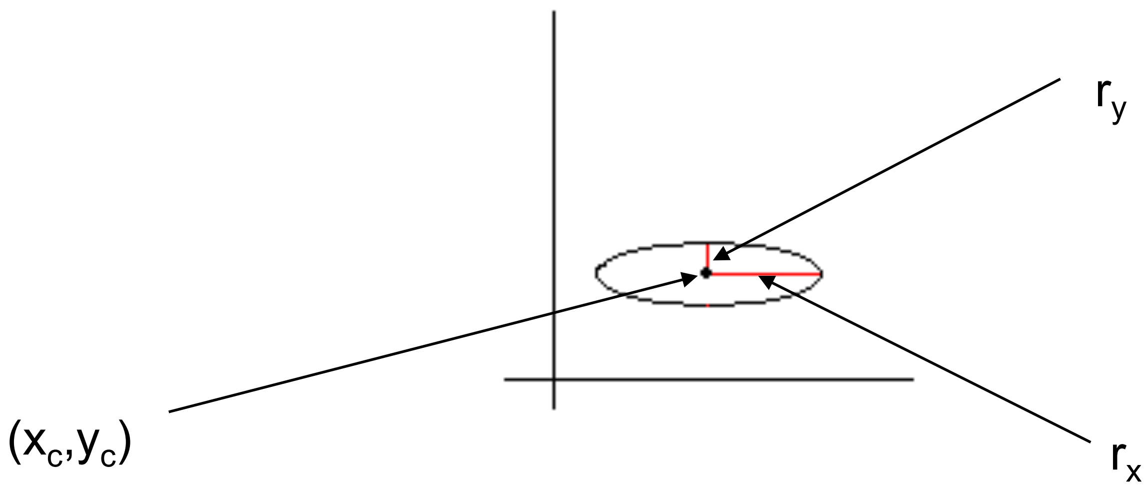


- If the major axis and minor axis are oriented to be aligned with the x and y axis (like above) the ellipse is said to be in **“standard position”**

Output primitives - Ellipses

- An ellipse in standard position has equation of the following form

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1$$



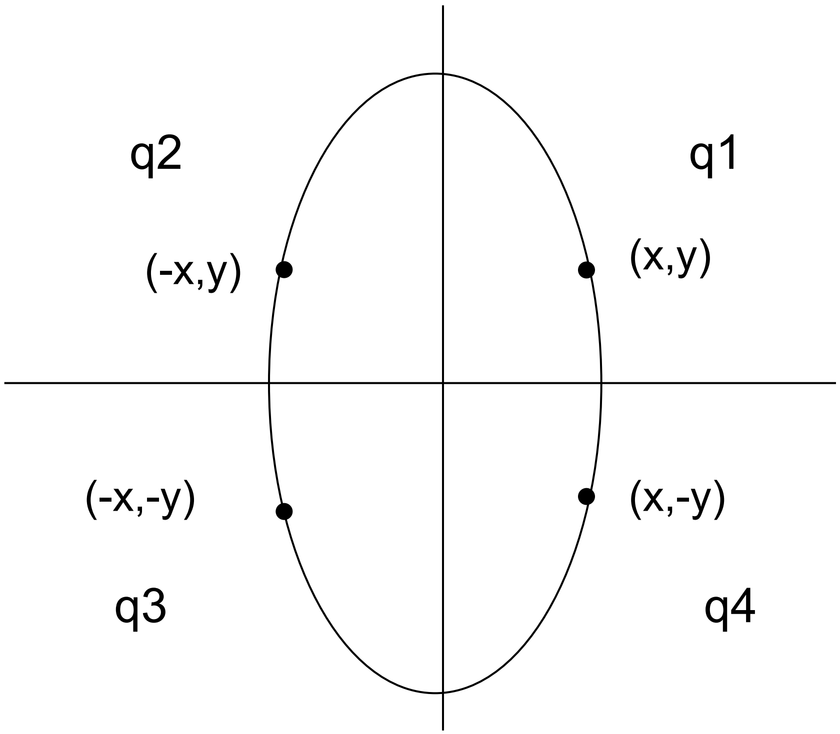
- r_x is the semi-major axis and r_y the semi minor axis, (x_c, y_c) the center

Scan conversion for ellipses – The Ellipse Drawing algorithm

- Now that we have defined what ellipses are and given an equation for an ellipse lets look at **scan converting them**
- The ellipse drawing algorithm is **similar to the circle algorithm**
- **Given (x_c,y_c), r_x and r_y** we determine points (x,y) on the ellipse using the **equation for an ellipse**
- We firstly compute the points for the ellipse with centre $(x_c,y_c) = (0,0)$ and then **add (x_c,y_c) to the computed coordinates to translate the ellipse**
- Ellipses are **only symmetric about the quadrant axis** so we need to compute coordinates for a complete quadrant as opposed to an octant for circles

What about symmetry?

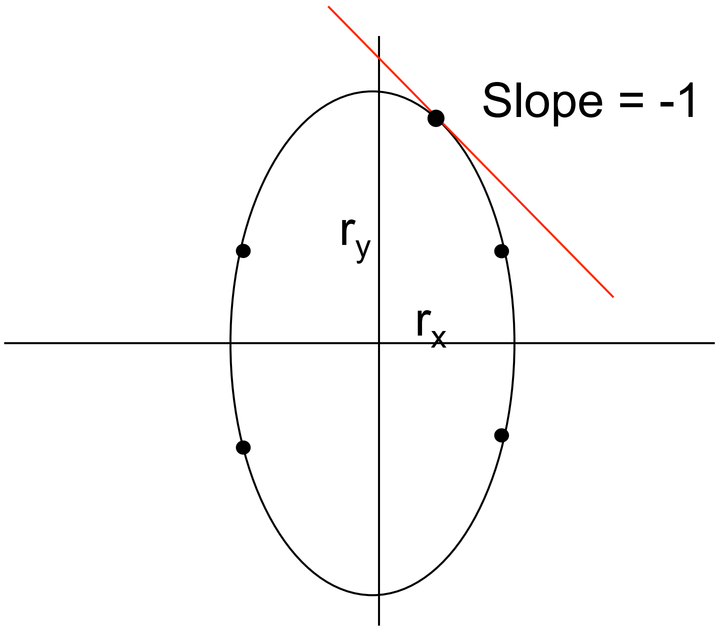
- The shape of the ellipse is similar in each quadrant



- We can use this symmetry so we **only need to calculate the positions on the boundary in one quadrant**

Scan conversion for ellipses

- The algorithm is applied through the first quadrant in **two parts**
- We will only consider ellipses where $\mathbf{r_x < r_y}$ like the one below

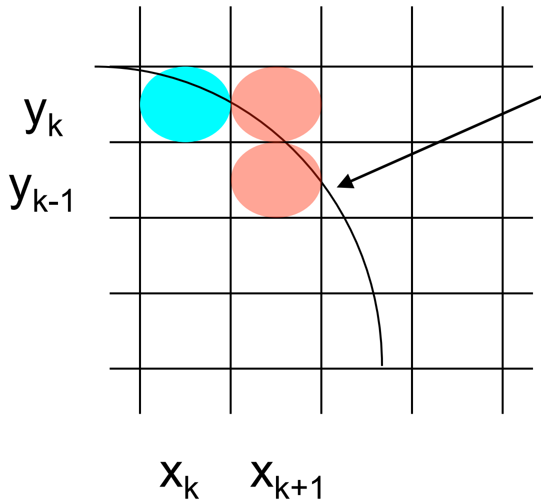


- We start with the point $(0, r_y)$ and calculate the next pixel position the same way as the circle except using the equation of the ellipse with ellipse center = (0,0)

$$r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

Scan converting ellipses – The Ellipse drawing algorithm

- Just like the circle **all points inside the ellipse have ellipse equation less than zero**, **all points on the boundary equal to zero** and **all points outside the ellipse greater than zero**



$$r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

- BUT, each time we calculate the next pixel position we have to evaluate the slope of the tangent of the ellipse also

- **When the slope changes to -1 we have to change the increment from an x increment to y decrement and calculate the next x pixel position as x_k or x_{k+1}**

Scan converting ellipses

- The next pixel position changes from:

$$f_{ellipse}(x_k + 1, y_k)$$

OR

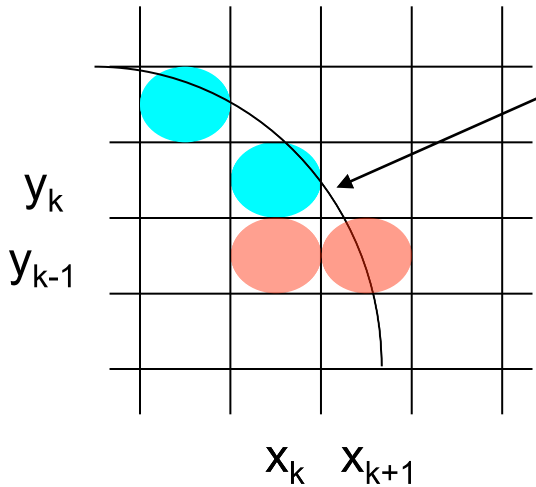
$$f_{ellipse}(x_k + 1, y_{k-1})$$

Slope <= -1 $\longrightarrow$

$$f_{ellipse}(x_k, y_k - 1)$$

OR

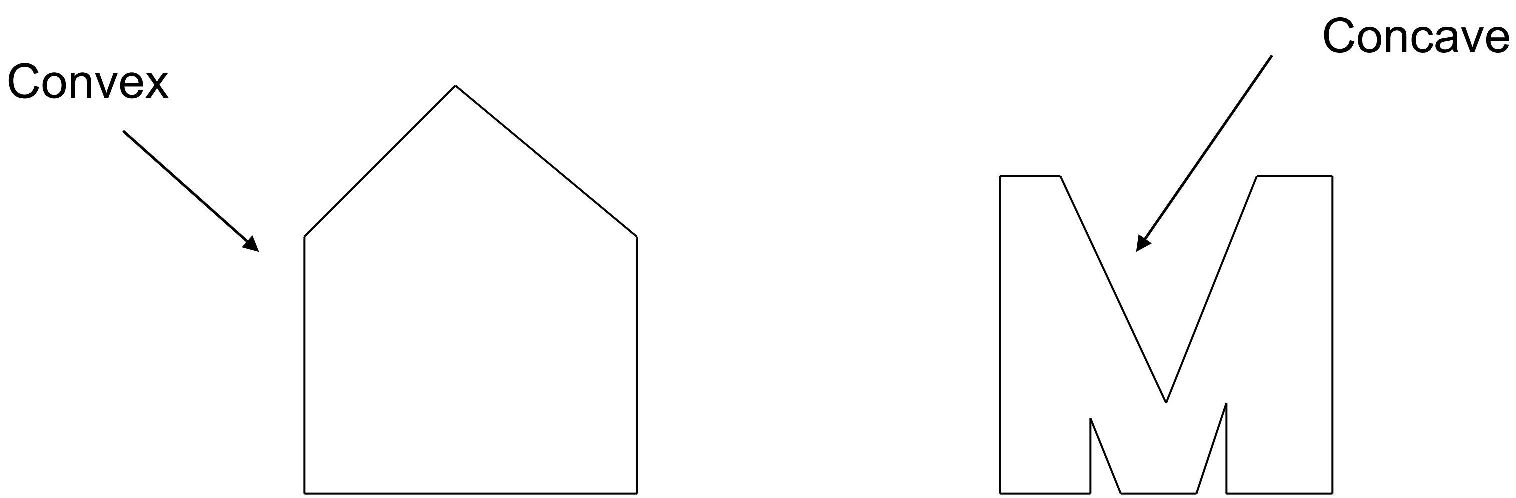
$$f_{ellipse}(x_{k+1}, y_k - 1)$$



$$r_y^{\,2} x^2 + r_x^{\,2} y^2 - r_x^{\,2} r_y^{\,2}$$

Polygons

- A polygon is a sequence $P_0, P_1, \ldots\ldots\ldots, P_{n-1}$ vertices (points) where N>=3 and the associated edges $P_0P_1, P_1P_2, \ldots\ldots\ldots, P_{n-1}P_0$
- Polygons can be classified as concave or convex
- Convex polygons are those where all the interior angles of two edges meeting at a vertex is < 180 degrees
- If two edges meet at a vertex and have an interior angle > 180 degrees the polygon is said to be concave.



Area of a Polygon

- The area of a polygon (convex or concave) can be computed from the equation below – the vertices $P_0 \ldots P_{n-1}$ should be labeled counter-clockwise

$$2A(P_0.....P_{n-1}) = \sum_{i=0}^{n-1}(x_i y_{i+1} - y_i x_{i+1})$$

- We will not go into great detail as to how this equation is derived but suffice to say that the polygon is broken down into a series of triangles and the area of each summed together.

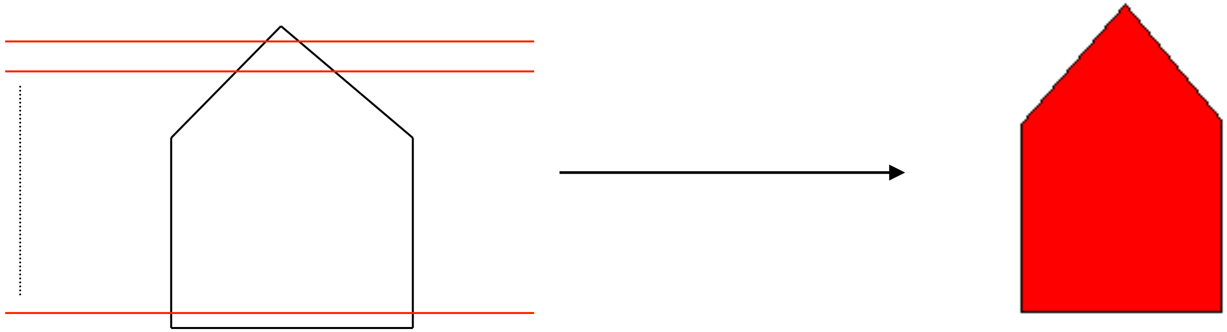
Polygon Filling

• Polygons can be represented as a structured set of points and the boundary can be displayed by simply plotting the lines between the point representation

• But how do we fill the polygon, that is how do we determine what pixel coordinates should be coloured/filled as part of the polygon

• Techniques to achieve this are called Area or Polygon fill algorithms – we will look briefly at one such technique

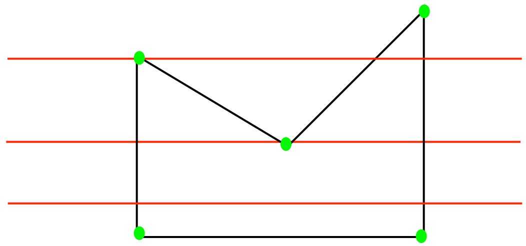
• Consider the following convex polygon:

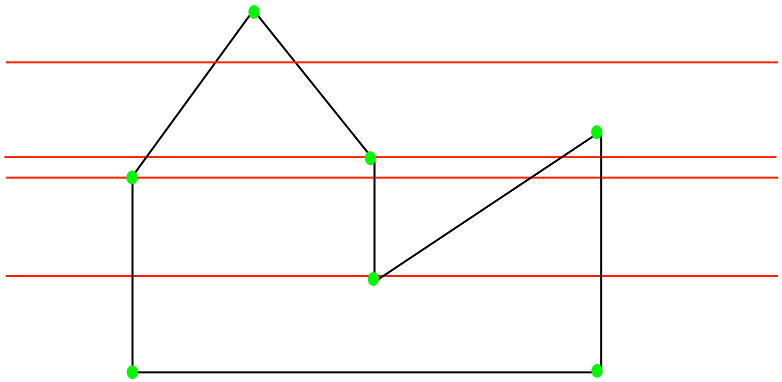


• The red lines are the scan lines on a raster display device. By simply noting where the scan line intersects with a polygon edge (line), we can tell if we are inside or outside of the polygon, i.e. cross first edge brings us inside, cross another brings outside etc.

Polygon Filling

- There are some ‘special’ cases where this simple technique will not work, consider the following polygons and determine whether the technique will work or not?



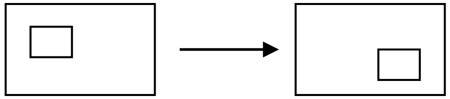


- This simple outside-inside algorithm will work fine for filling the polygon on the left but what about the one on the right
- You should see that it breaks down when the scan line encounters a point connected by two edges where the edges in question, have y values that are monotonically increasing or decreasing
- These points should be treated as a special case and only one edge should be included at these points

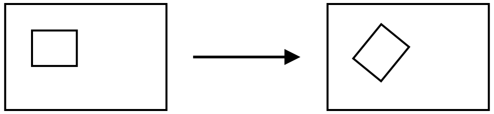
Transformations in 2D

- We looked at some more output primitives in the form of circles and ellipses and efficient scan conversion techniques for them
- Variations of these algorithms can be applied to other curves as well that we have not looked at like parabolas, hyperbolas, splines etc.
- All these curves and their three dimensional equivalent form the set of output primitives in computer graphics and are the **basic building blocks of a scene to be displayed by a graphics device**
- Today we will look at how to apply **transformations to these 2-d primitives** and get them to move around the viewing coordinate system the way we would like
- We will firstly look at **geometric transformations** of objects

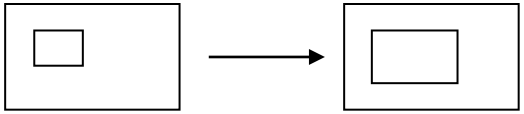
translation

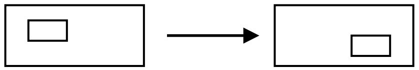


rotation



scaling





Translation

- A translation is applied to an object by **repositioning it along a straight line path** from one location to another

- A two dimensional point is translated by adding translation distances t_x and t_y to the original coordinate positions

$$x' = x + t_x \qquad\qquad y' = y + t_y$$

Or in matrix form as,

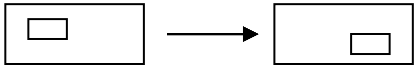
$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

Translation equation

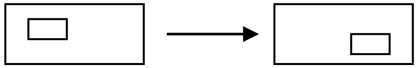
T is called the translation vector

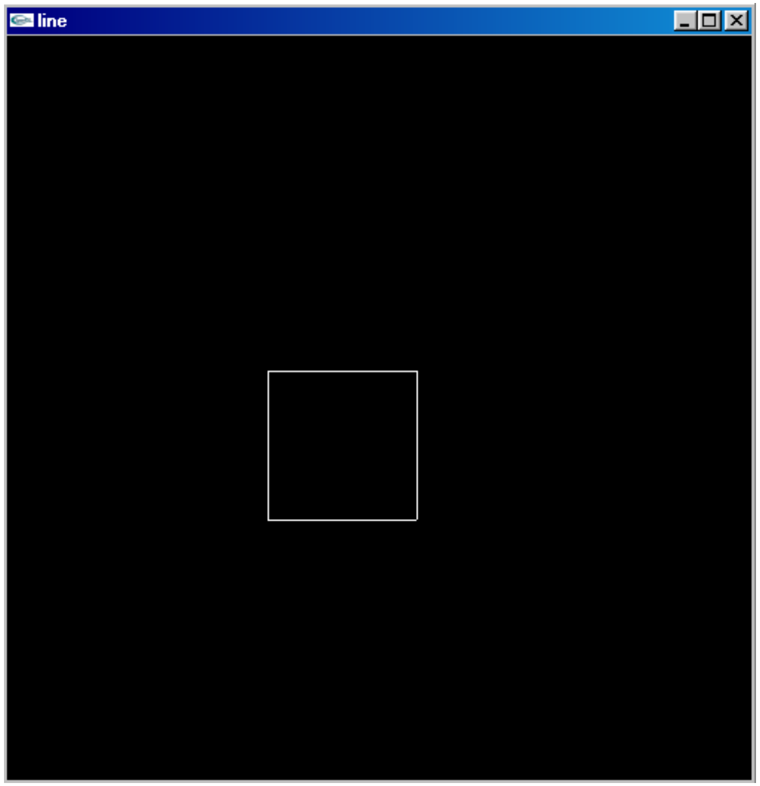
Translation

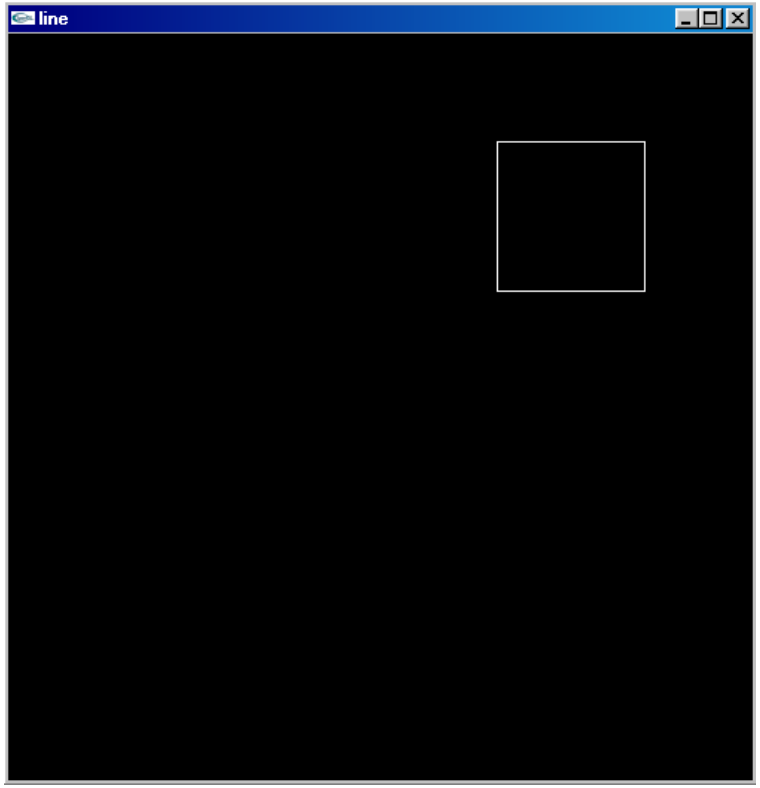


- Translation is known as a **rigid body transformation** and moves objects without deformation
- Lines can be translated by applying the translation equation to **both of the line endpoints and redrawing the line**
- Polygons can be translated by applying the translation equation to each of the vertices and regenerating the polygon using the new vertices
- Circles and ellipses can be translated by applying the translation equation to **the center coordinates** and redrawing the object in the new location
- Other objects can be translated by applying the translation transformation equations to the parameters defining the object

Translation

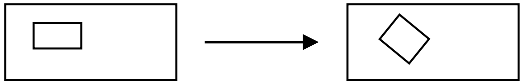




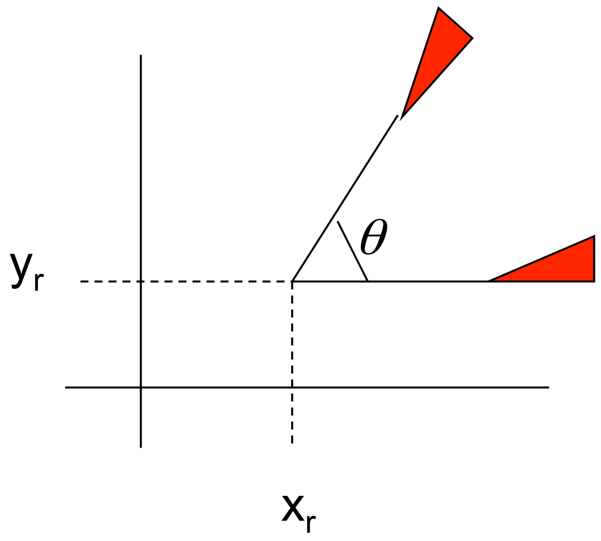


The effect of translating a square in openGL

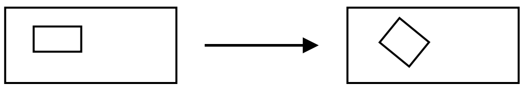
Rotation



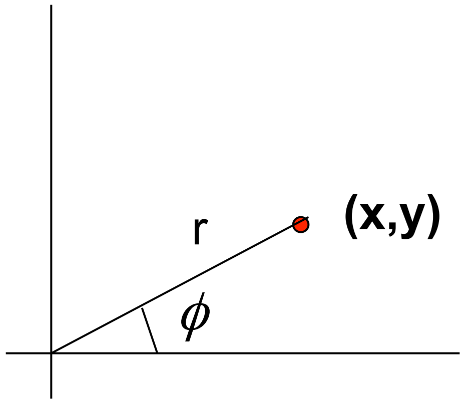
- A two dimensional rotation is applied to an object by **repositioning it along a circular** path in the *x-y* plane
- To generate a rotation we specify a rotation angle θ, which is the **amount by which we wish to rotate the object** and a rotation point (x_r,y_r), about which the object is to be rotated



Rotation

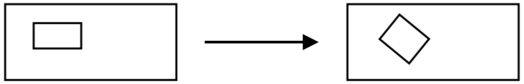


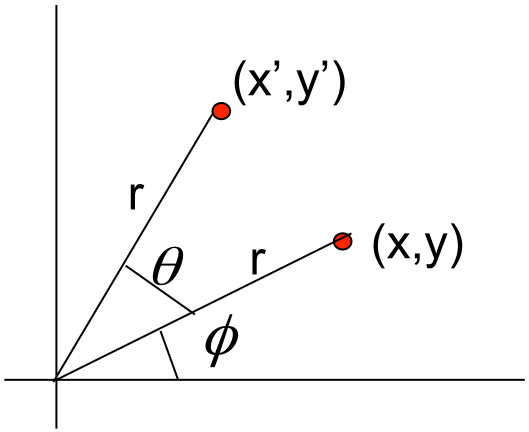
- **Positive values of theta define counter clockwise** rotations and **negative values clockwise rotations**
- The transformation equations are **simplified** somewhat if the **rotation point is at the origin**
- Rotations are specified in polar coordinates



$$x = r\cos\phi \qquad y = r\sin\phi$$

Rotation





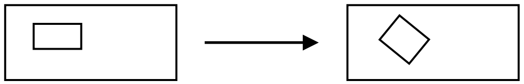
The rotated values (x' ,y') are given by the equations

$$x' = r\cos(\phi + \theta) \qquad y' = r\sin(\phi + \theta)$$

Or,

$$x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta \qquad y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

Rotation



- Substituting the equations for *x* and *y* into the equations for *x'* and *y'* gives the transformation equation for rotating a point at position (x,y) through an angle θ about the origin

$$x' = x\cos\theta - y\sin\theta \qquad y' = x\sin\theta + y\cos\theta$$

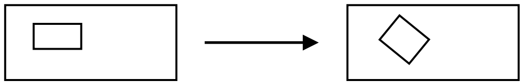
Or in matrix form as,

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P}$$

R is called the rotation matrix

Rotation



- For arbitrary pivot points (x_r, y_r) the transformation equations become

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$

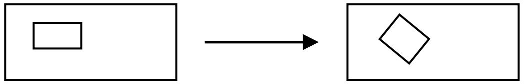
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

- The matrix form can be computed accordingly

$$\mathbf{P} = \begin{bmatrix} x - x_r \\ y - y_r \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

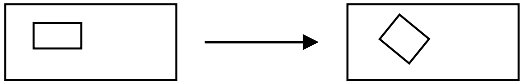
$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} + \begin{bmatrix} x_r \\ y_r \end{bmatrix}$$

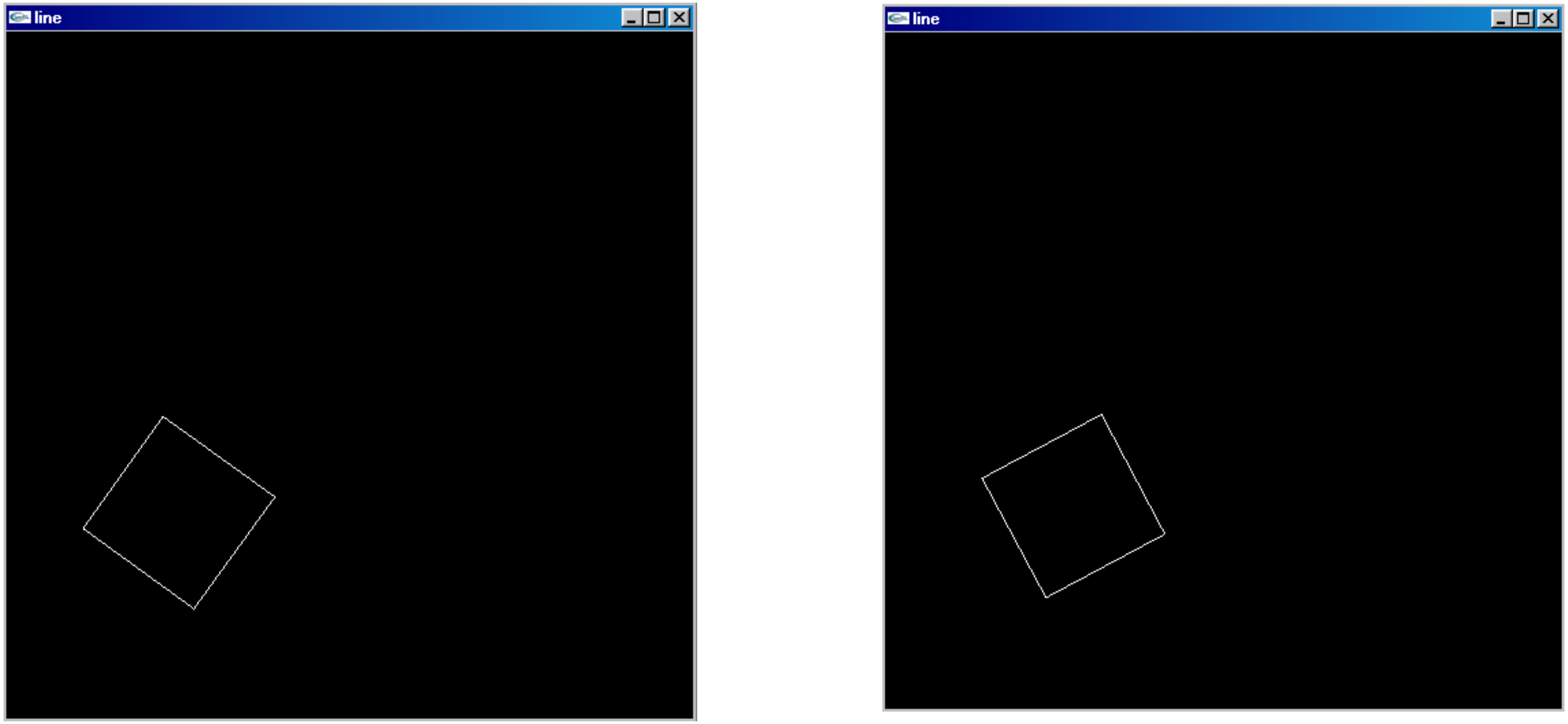
Rotation



- Rotation is known as a **rigid body transformation** and moves objects without deformation
- Lines can be rotated by **applying the rotation equation to both of the line endpoints** and redrawing the line
- Polygons can be rotated by **applying the rotation equation to each of the vertices** and regenerating the polygon using the new vertices
- Ellipses can be rotated by applying the **rotation equation to coordinates defining the major and minor axis**
- Curved lines are rotated by repositioning the defining points and redrawing the curves

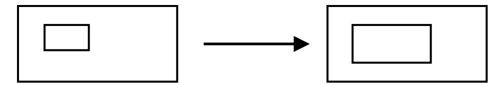
Rotation





The effect of rotating a square in openGL

Scaling



- Scaling is not a rigid body transformation because it changes the size of the object
- Scaling for polygons is possible by multiplying each vertex by scaling factors s_x and s_y to produce the scaled coordinate (x',y')

$$x' = x \cdot s_x \qquad\qquad y' = y \cdot s_y$$

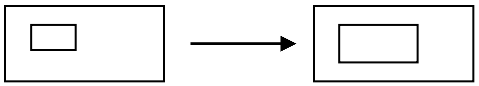
Or in matrix form as,

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

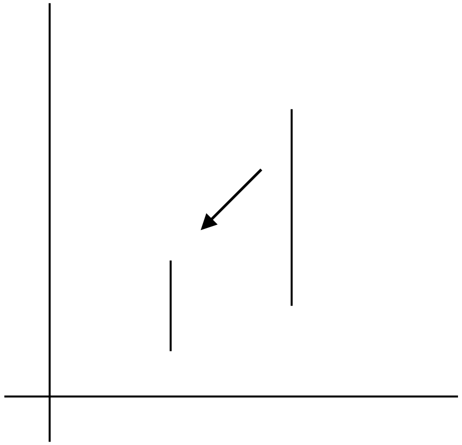
$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

S is called the scaling matrix

Scaling

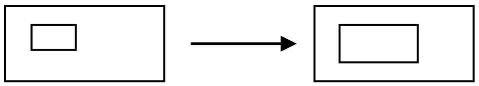


- Values less than 1 for s_x and s_y shrink the object, values greater than 1 enlarge the object
- Unequal values of s_x and s_y results in differential scaling which can produce new shapes
- Objects transformed with the scaling matrix defined previously are actually scaled and translated
- Scaling coefficients **less than one move the object closer to the origin** while **scaling coefficients greater than one move it further away from the origin** than it was previously



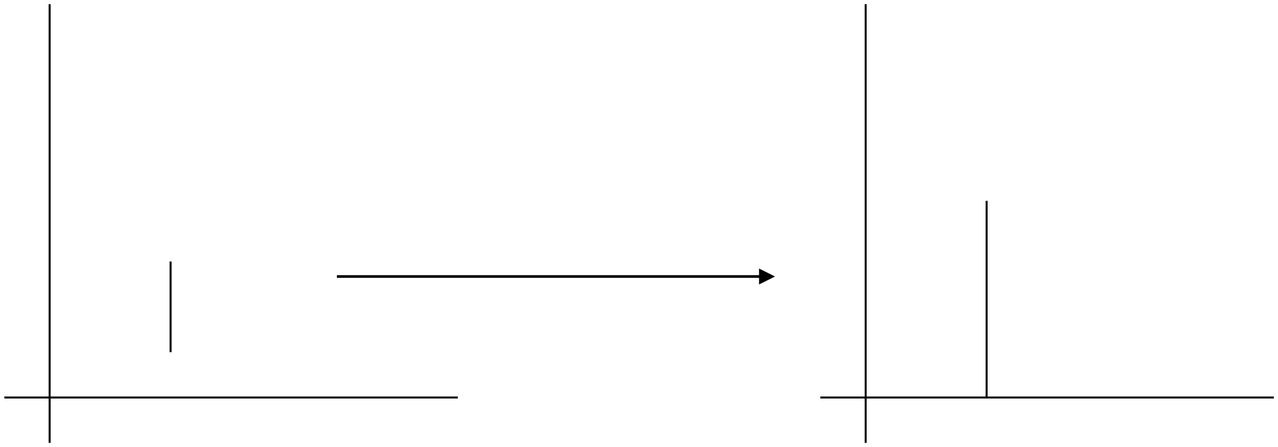
A line scaled with previous equations for s_x and s_y (s_x=s_y=0.5)

Scaling

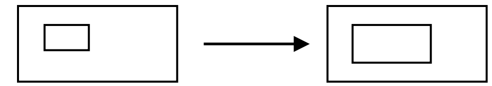


- This problem can be resolved by **choosing a fixed point that is to remain unchanged after the scaling**
- Coordinates for this point (x_f, y_f) can be chosen to be one of the vertices, the object centroid or any other position
- A polygon is then scaled by **applying scaling factors to the distance from each vertex to the fixed point**

A line scaled relative to fixed mid-point ($s_x = s_y = 0.5$)



Scaling



- So our new scaling equations relative to a fixed point are:

$$x' = x_f + (x - x_f)s_x \qquad\qquad y' = y_f + (y - y_f)s_y$$

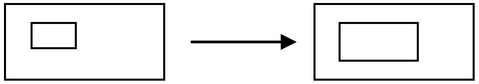
Or

$$x' = x \cdot s_x + x_f(1 - s_x) \qquad\qquad y' = y \cdot s_y + y_f(1 - s_y)$$

Where $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constant for all object vertices

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} + \begin{bmatrix} x_f(1 - s_x) \\ y_f(1 - s_y) \end{bmatrix}$$

Scaling



• Lines can be scaled by applying the **scaling equation to both of the line endpoints** and redrawing the line

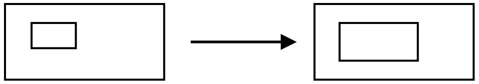
• Polygons can be scaled by applying the **scaling equation to each of the vertices and regenerating the polygon using the new vertices**

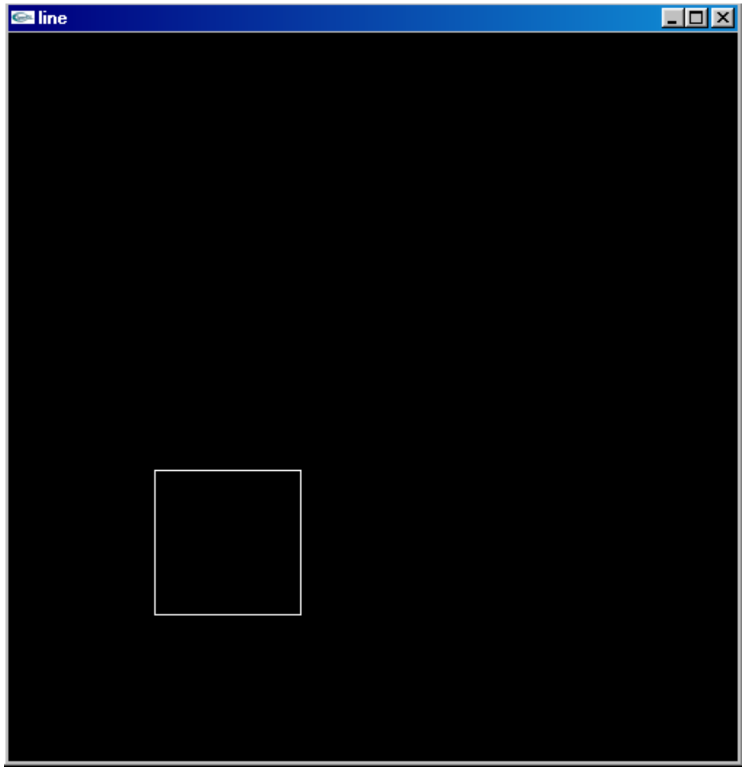
• Ellipses can be scaled by **applying the scaling equation to the coordinates defining the major and minor axis**

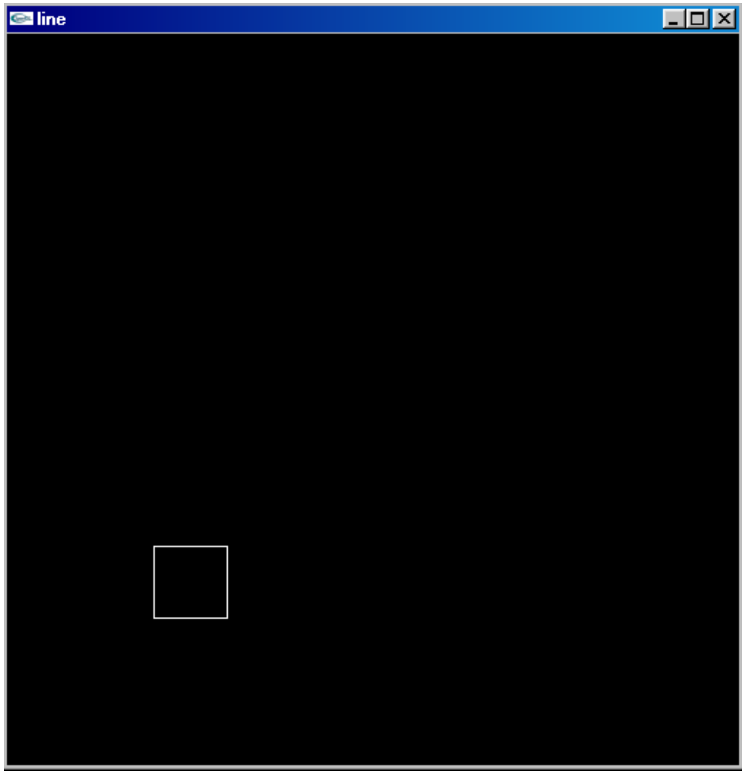
•Circles can be scaled by simply **adjusting the radius**

• Other objects can be scaled by applying the scaling transformation equations to the parameters defining the object

Scaling

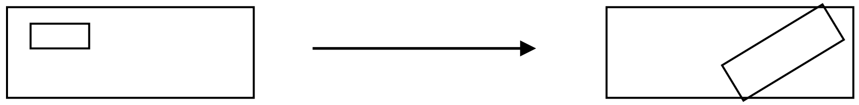






The effect of scaling a square in openGL

Combining transformations



• Many graphics application involves **sequences of transformations**

• Imagine an object flying or translating through the air, most of the time it will **be spinning or rotating as well**

• For these applications we have to **apply a number of transformation, one followed by another**

• The form of the transformations we have looked at so far is

$$\mathbf{P}' = \mathbf{M} \cdot \mathbf{P} + \mathbf{T}$$

• Applying sequences of transformations in this form can involve **calculating the transformed coordinates one step at a time or introducing big memory overheads to record previous transformations**

• A more efficient approach would be to combine the transformations so the final coordinates are computed directly from the initial coordinates

Homogenous coordinates

- We can achieve this by reformulating the matrix form of our transformations with multiplications only

$$\mathbf{P}' = \mathbf{M} \cdot \mathbf{P} + \mathbf{T} \longrightarrow \mathbf{P}' = \mathbf{M_h} \cdot \mathbf{P}$$

- Where M above is a transformation matrix

- If we could do this then computing the new coordinates would involve simply multiplying the transformation matrices by one another and the result by the coordinate we want the transformations applied to

$$\mathbf{P}' = \mathbf{M_{h1}} \cdot \mathbf{P} \qquad \mathbf{P}'' = \mathbf{M_{h2}} \cdot \mathbf{P}'$$

$$\mathbf{P}'' = \mathbf{M_{h2}}(\mathbf{M_{h1}} \cdot \mathbf{P}) \longrightarrow \mathbf{P}'' = (\mathbf{M_{h2}}\mathbf{M_{h1}}) \cdot \mathbf{P}$$

- We can reformulate our equations to use matrix multiplication only by using **homogenous coordinates**

Homogenous coordinates

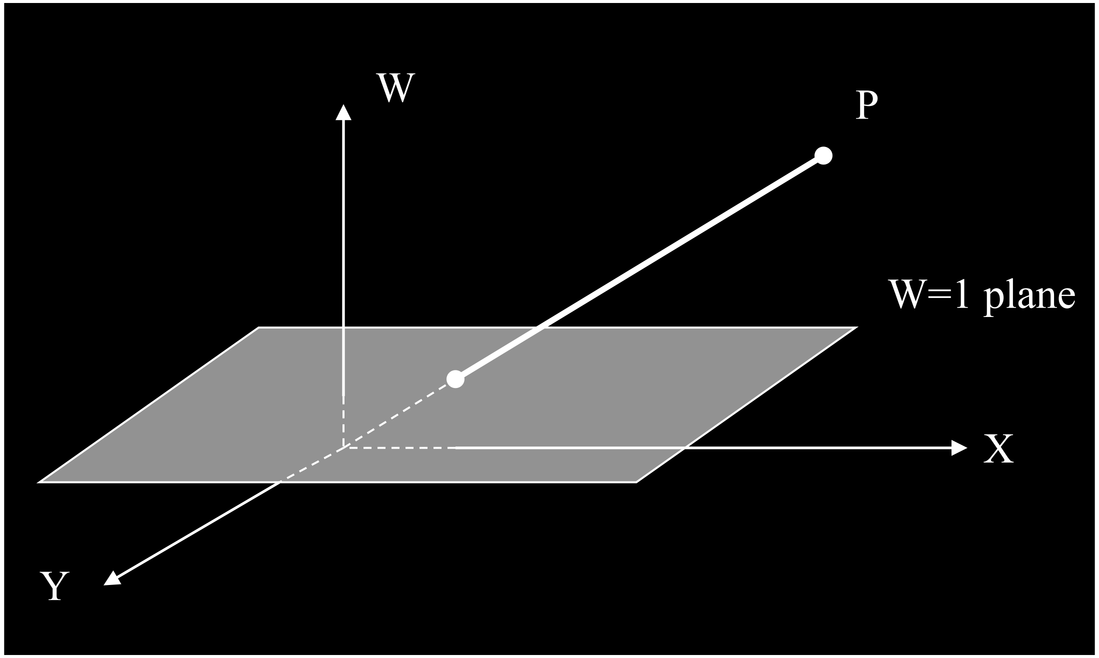
- Add an extra coordinate, W, to a point
- P(x,y,W)
- Two sets of homogeneous coordinates represent the same point if they are a multiple of each other
- (2,5,3) and (4,10,6) represent the same point
- At least one component must be non-zero, (0,0,0) is not defined
- If $W \neq 0$, divide by it to get Cartesian coordinates of point (x/W,y/W,1)
- If W=0, point is said to be at infinity.

Homogenous coordinates

- If we represent (x,y,W) in 3-space, all triples representing the same point describe a line passing through the origin

- If we homogenize the point, we get a point of form (x,y,1)

    - Homogenised points form a plane at W=1.



Homogenous coordinate representation of Translation

- Now we can represent translations as matrix multiplications
- Coordinate points are now a (x,y,w) triple
- Transformation equations are now 3 x 3

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

$$x' = x + t_x$$
$$y' = y + t_y$$
$$1 = 1$$

- Or as:

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P}$$

Homogenous coordinate representation of Rotation

- We can represent rotation about a fixed pivot as matrix multiplications
- For **fixed pivot equal to the coordinate origin** we have

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Or as:

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P}$$

Homogenous coordinate representation of Rotation

- If the pivot point is not at the origin we can still use the rotation matrix defined for an origin pivot

- We achieve this by **firstly translating the object so the pivot point is at the origin**

- **Applying the rotation about the origin**

- **Translating the object so the pivot point is returned to it's original position**

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Homogenous coordinate representation of Scale

- Note in the previous slide that we multiplied transformation matrices

- Forming products of transformations like this is called **concatenation**

- Finally the **scaling transformation relative to the coordinate** origin is now expressed as matrix multiplication

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

- Or as

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P}$$

Scaling matrix (arbitrary fixed point)

General fixed point scaling can be achieved the same way general pivot point rotation can be, that is:

1. Translate object so fixed point is at the coordinate origin
2. Scale the object with respect to the coordinate origin
3. Use the inverse translation of step 1 to return the object to its original position

Concatenating the matrices for these three operations produces the required scaling matrix

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$