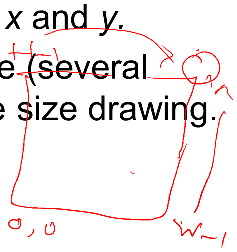
# Coordinate Systems

## Coordinate Systems

- We have been using the coordinate system of the screen window (in pixels).
- The range is:
  - from 0 (left) to some value screenWidth – 1 in *x*,
  - from 0 (usually top) to some value screenHeight –1 in *y*.
  - We can use only positive values of *x* and *y*.
  - The values must have a large range (several hundred pixels) to get a reasonable size drawing.

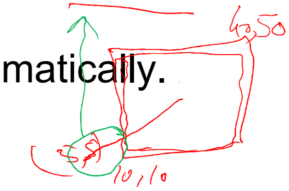


## Coordinate Systems (2)

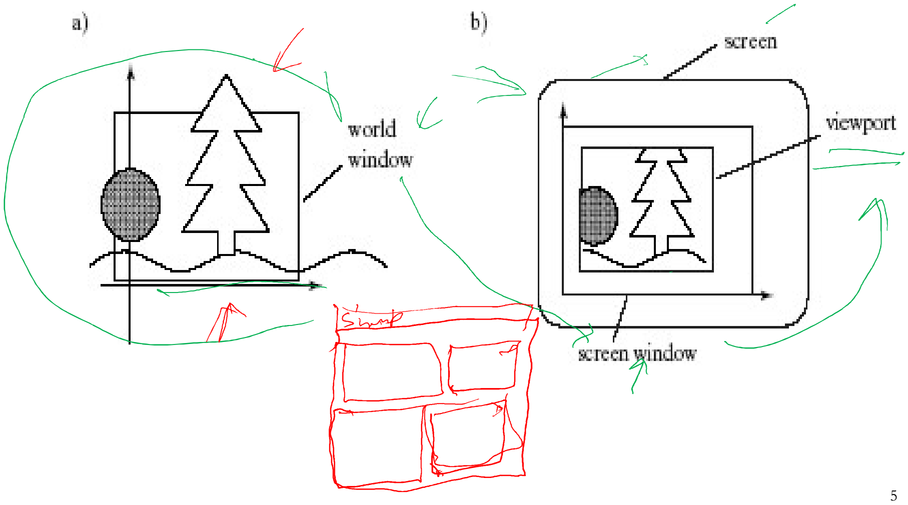
- Description of object is usually referred to as a **modeling** task, and displaying pictures as a **viewing** task.
- The space in which objects are described is called **world coordinates** (the numbers used for x and y are those in the world, where the objects are defined).
- World coordinates use the Cartesian *xy*-coordinate system used in mathematics, based on whatever units are convenient.

## Coordinate Systems (3)

- We define a rectangular **world window** in these world coordinates.
- The world window specifies which part of the world should be drawn:
  - whichever part lies inside the window should be drawn
  - whichever part lies outside should be clipped away and not drawn.
- OpenGL does the clipping automatically.



# Coordinate Systems (4)



# Coordinate Systems (5)

- In addition, we define a rectangular **viewport** in the screen window on the display.
- A mapping between the world window and the viewport is established by OpenGL.
  - scaling [change size]
  - translations [move object]

transformation

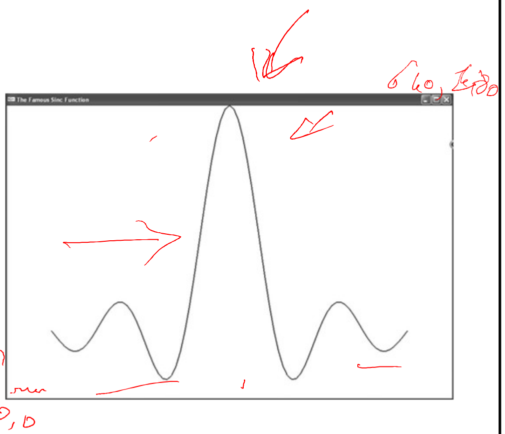
- The objects inside the world window appear automatically at proper sizes and locations inside the viewport (in **screen coordinates,** which are pixel coordinates on the display).

## Coordinate Systems Example

- We want to graph

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

- Sinc(0) = 1 by definition. Interesting parts of the function are in -4.0 ≤ x ≤ 4.0.



## Coordinate Systems Example (2)

- The program which graphs this function is given in Fig.
- The function setWindow sets the world window size:

```
void setWindow(GLdouble left, GLdouble right,
   GLdouble bottom, GLdouble top)
{          glMatrixMode(GL_PROJECTION);
           glLoadIdentity();
           gluOrtho2D(left, right, bottom, top);}
```

## Coordinate Systems Example (3)

- The function setViewport sets the screen viewport size:

```
void setViewport(GLint left, GLint right, GLint
   bottom, GLint top)
{ glViewport(left, bottom, right - left, top -
   bottom);}
```

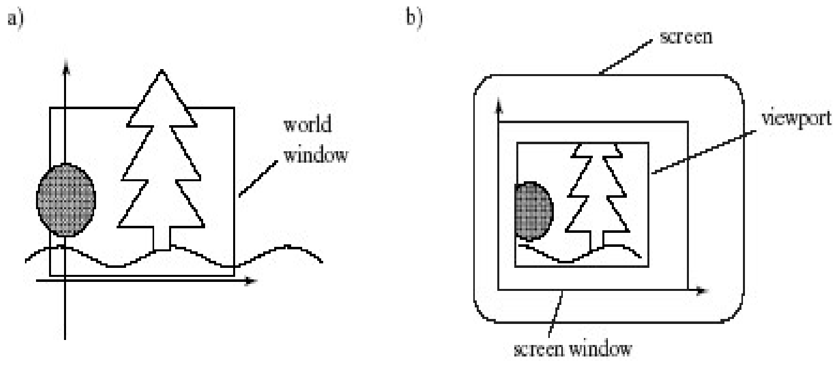
- Calls: setWindow(-5.0, 5.0, -0.3, 1.0);
- setViewport(0, 640, 0, 480);

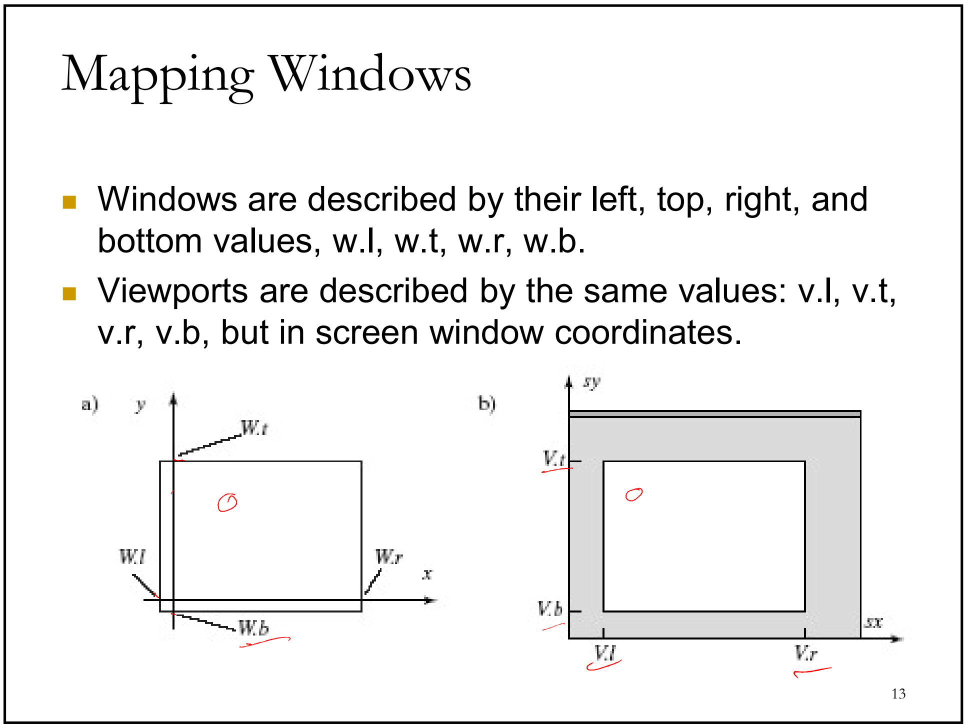
## Window to Viewport Transformation

# Windows and Viewports

- We use natural coordinates for what we are drawing (the world window).
- OpenGL converts our coordinates to screen coordinates when we set up a screen window and a viewport.
- The viewport may be smaller than the screen window. The default viewport is the entire screen window.
- The conversion requires scaling and shifting:
  - mapping the world window to the screen window and the viewport.

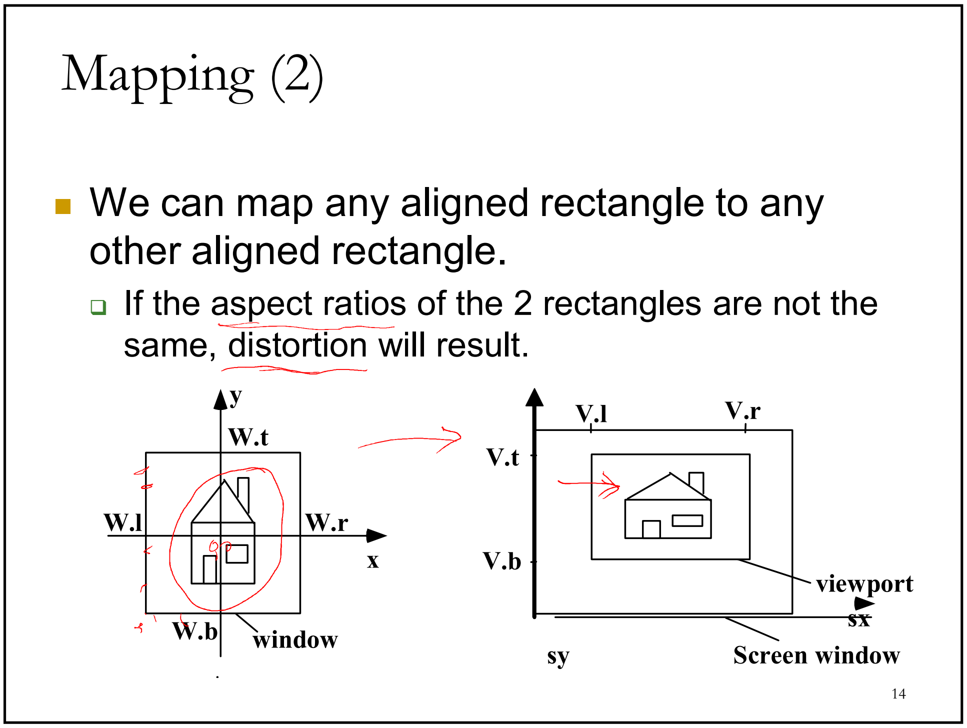
# Windows and Viewport





## Mapping Windows

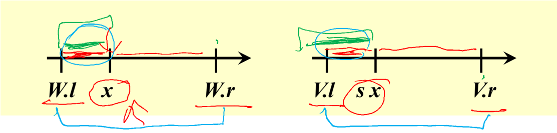
- Windows are described by their left, top, right, and bottom values, w.l, w.t, w.r, w.b.
- Viewports are described by the same values: v.l, v.t, v.r, v.b, but in screen window coordinates.



## Mapping (2)

- We can map any aligned rectangle to any other aligned rectangle.
  - If the aspect ratios of the 2 rectangles are not the same, distortion will result.

## The Mapping from the World Window to the Viewport



Proportionality forces the mapping to have the linear form for x to sx:

$$sx = Ax + C \qquad (1)$$

For some constants *A* and *C*. From the figure above, we can write

$$\frac{sx - V.l}{V.r - V.l} = \frac{x - W.l}{W.r - W.l} \qquad (2)$$

Rearranging

$$sx = V.l + \frac{x - W.l}{W.r - W.l}(V.r - V.l) \qquad (3)$$

Or

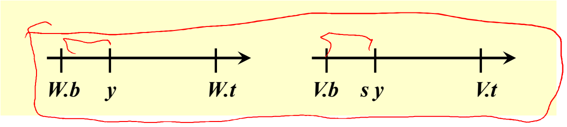
$$sx = \overset{A}{\boxed{\left(\frac{V.r - V.l}{W.r - W.l}\right)}}x + \underbrace{\left\{V.l - \overset{A}{\left(\frac{V.r - V.l}{W.r - W.l}\right)}W.l\right\}}_{C} \qquad (4)$$

x

y

$$A = \left(\frac{V.r - V.l}{W.r - W.l}\right), \quad C = \{V.l - A \times W.l\} \qquad (5)$$

## The Mapping from the World Window to the Viewport



W.t V.t

Similarly proportionality in y demands:

$$sy = By + D \qquad (6)$$

For some constants B and D. From the figure above, we can write

$$\frac{sy - V.b}{V.t - V.b} = \frac{y - W.b}{W.t - W.b} \qquad (7)$$

Rearranging

$$sy = V.b + \frac{y - W.b}{W.t - W.b}(V.t - V.b) \qquad (8)$$

Or

$$sy = \boxed{\left(\frac{V.t - V.b}{W.t - W.b}\right)}y + \left\{V.b - \left(\frac{V.t - V.b}{W.t - W.b}\right)W.b\right\} \qquad (9)$$

sy

$$B = \left(\frac{V.t - V.b}{W.t - W.b}\right), \quad D = \{V.b - B \times W.b\} \qquad (10)$$

## The Mapping from the World Window to the Viewport

Collectively, the transformation equations are:

$$sx = Ax + C$$
$$sy = By + D \qquad (11)$$

Where the constants A, B, C and D are given by:

$$A = \left(\frac{V.r - V.l}{W.r - W.l}\right), \quad C = \{V.l - A \times W.l\} \qquad (12)$$

And

$$B = \left(\frac{V.t - V.b}{W.t - W.b}\right), \quad D = \{V.b - B \times W.b\} \qquad (13)$$

The mapping can be used with any point (x, y) inside or outside of the window. Points inside the window map to points inside the viewport, and points outside the window map to points outside the viewport.

Consider the window and viewport in Fig. The window has (*W.l*, *W.r*, *W.b*, *W.t*) = (0, 2.0, 0, 1.0) and the viewport has (*V.l*, *V.r*, *V.b*, *V.t*) = (40, 400, 60, 300).

$$A = \left(\frac{V.r - V.l}{W.r - W.l}\right), \quad C = \{V.l - A \times W.l\}$$

$$B = \left(\frac{V.t - V.b}{W.t - W.b}\right), \quad D = \{V.b - B \times W.b\}$$

$$sx = Ax + C \qquad sy = By + D$$

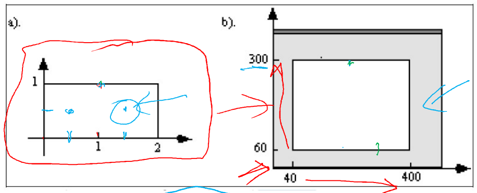
$$A = \frac{400-40}{2-0} = \frac{360}{2} = 180$$

$$C = 40 - 180 \times 0 = 40$$

$$B = \frac{300-60}{1-0} = 240$$

$$D = 60 - 240 \times 0 = 60$$

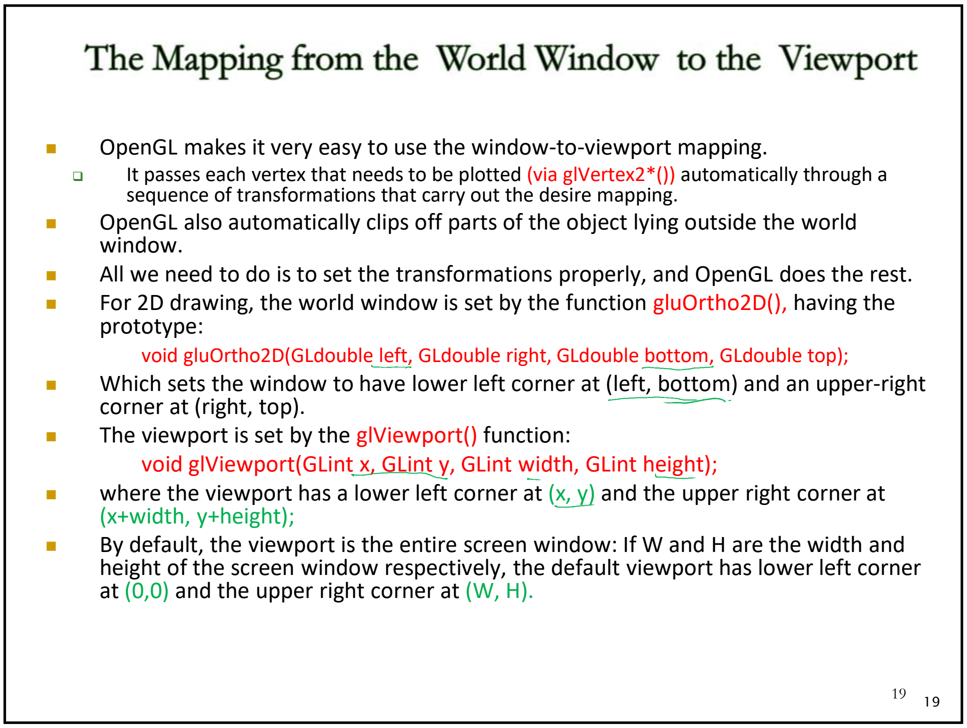
$$Sx = 180 * 1 + 40 = \boxed{220}$$
$$Sy = 240 * 1 + 60 = \boxed{300}$$



P(1 1)

$P_2(1.5, 0.5)$

$P_2'(310, 180)$

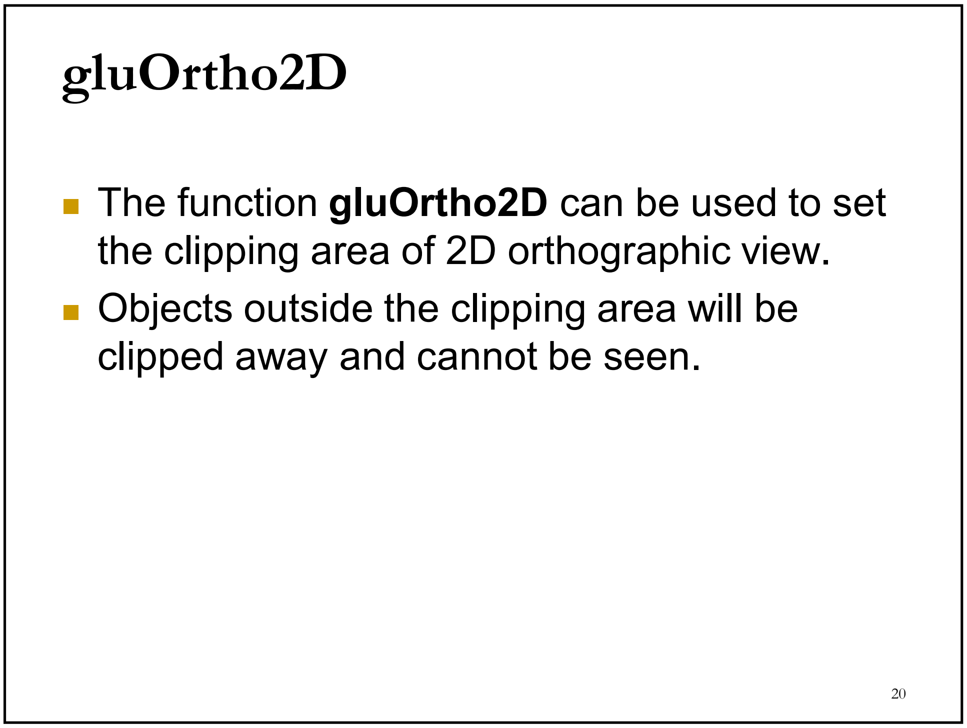


## The Mapping from the World Window to the Viewport

- OpenGL makes it very easy to use the window-to-viewport mapping.
  - It passes each vertex that needs to be plotted (via glVertex2*()) automatically through a sequence of transformations that carry out the desire mapping.
- OpenGL also automatically clips off parts of the object lying outside the world window.
- All we need to do is to set the transformations properly, and OpenGL does the rest.
- For 2D drawing, the world window is set by the function gluOrtho2D(), having the prototype:

  `void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);`
- Which sets the window to have lower left corner at (left, bottom) and an upper-right corner at (right, top).
- The viewport is set by the glViewport() function:

  `void glViewport(GLint x, GLint y, GLint width, GLint height);`
- where the viewport has a lower left corner at (x, y) and the upper right corner at (x+width, y+height);
- By default, the viewport is the entire screen window: If W and H are the width and height of the screen window respectively, the default viewport has lower left corner at (0,0) and the upper right corner at (W, H).



## gluOrtho2D

- The function **gluOrtho2D** can be used to set the clipping area of 2D orthographic view.
- Objects outside the clipping area will be clipped away and cannot be seen.

# glLoadIdentity

modeling & viewing

- As for the modelview matrix, it is used to make various transformations to the models (objects) in your world.
- Like this you only have to define your object once and then translate it or rotate it or scale it.
- You would **use** the projection matrix before drawing the objects in your scene to set the view volume

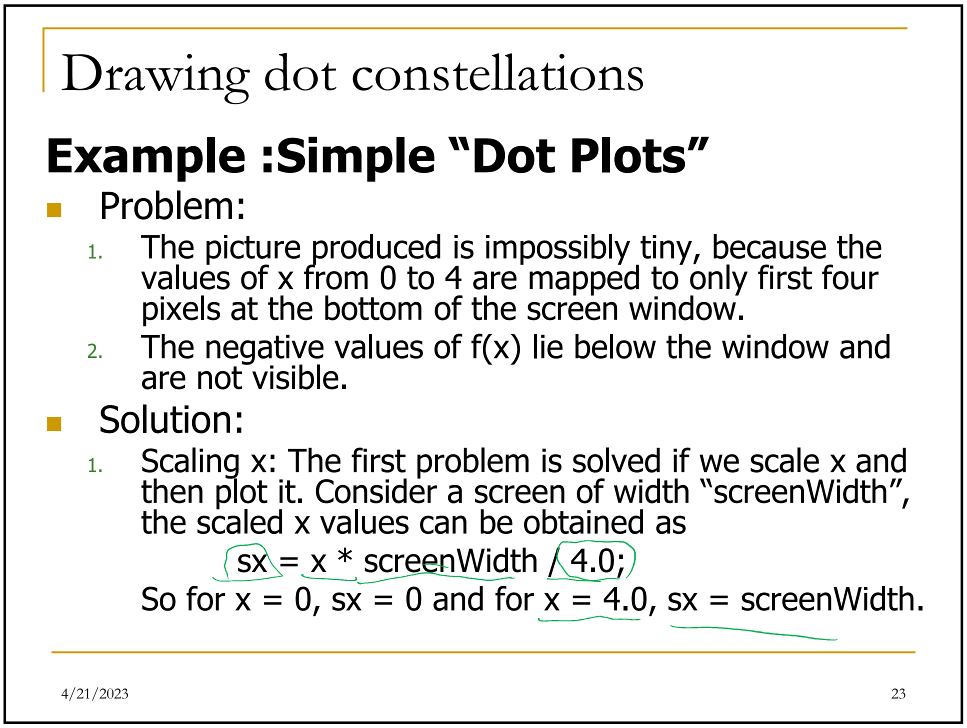
# Drawing dot constellations

## Example: Simple "Dot Plots"

- This example deals with learning the behaviour of some mathematical function f(x) as x varies.
- Suppose we have $f(x) = e^{-x}\cos(2\pi x)$

  where x varies from x=0 to x=4.
- To plot this function we "sample" it at a collection of equispaced x-values and plot a dot at each coordinate pair (x, f(x)).
- Choosing a suitable increment, say 0.005, between consecutive x-values, the basic process will run as follows:

```
glBegin(GL_POINTS);
        for (GLdouble x=0; x<4.0; x += 0.005)
                glVertex2d(x, f(x));
glEnd();
glFlush
```



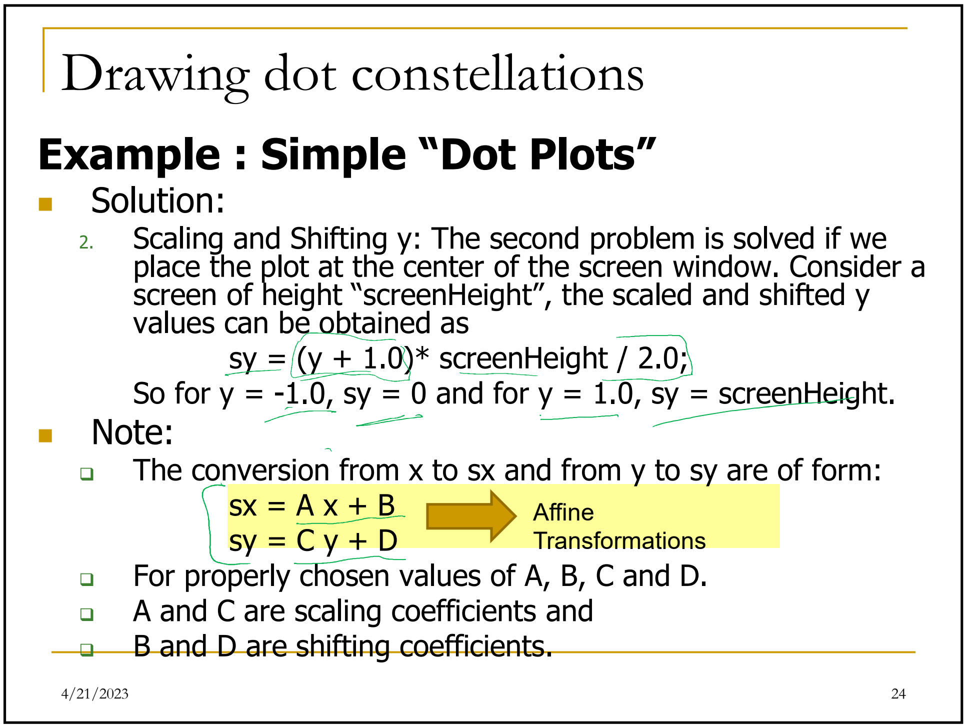
# Drawing dot constellations

## Example :Simple “Dot Plots”

- Problem:
  1. The picture produced is impossibly tiny, because the values of x from 0 to 4 are mapped to only first four pixels at the bottom of the screen window.
  2. The negative values of f(x) lie below the window and are not visible.
- Solution:
  1. Scaling x: The first problem is solved if we scale x and then plot it. Consider a screen of width “screenWidth”, the scaled x values can be obtained as

     sx = x * screenWidth / 4.0;

     So for x = 0, sx = 0 and for x = 4.0, sx = screenWidth.



# Drawing dot constellations

## Example : Simple “Dot Plots”

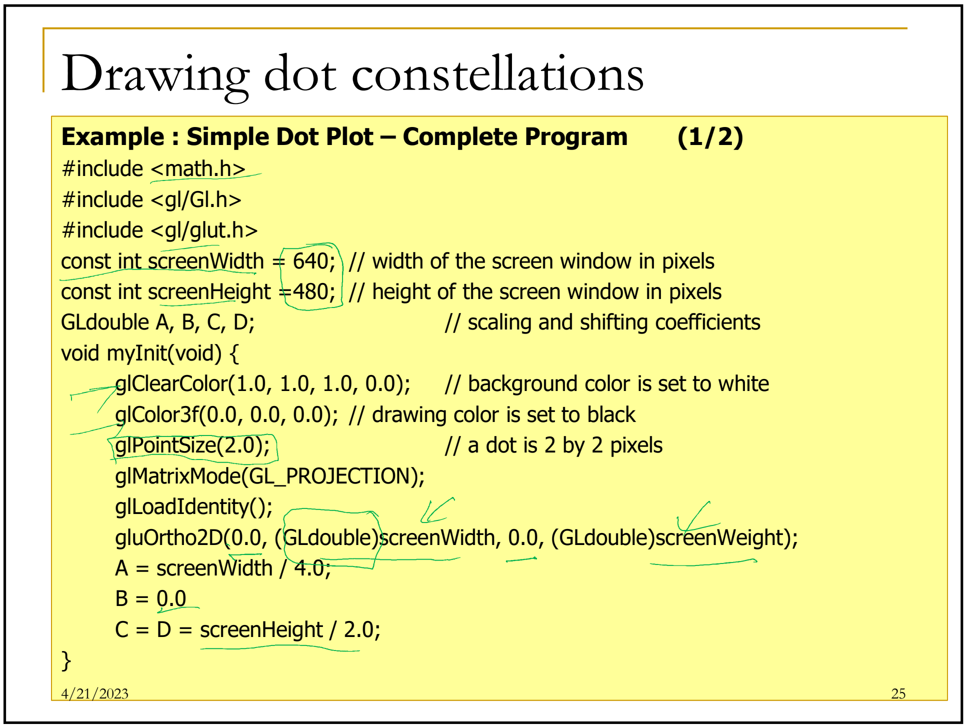
- Solution:
  2. Scaling and Shifting y: The second problem is solved if we place the plot at the center of the screen window. Consider a screen of height “screenHeight”, the scaled and shifted y values can be obtained as

     sy = (y + 1.0)* screenHeight / 2.0;

     So for y = -1.0, sy = 0 and for y = 1.0, sy = screenHeight.
- Note:
  - The conversion from x to sx and from y to sy are of form:

    sx = A x + B
    sy = C y + D

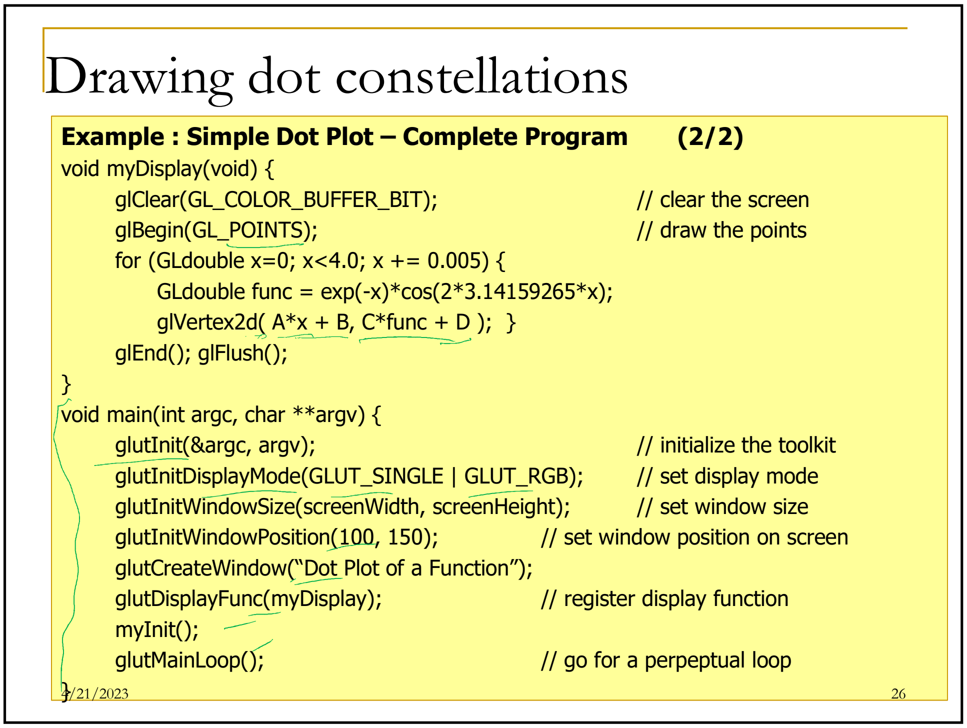
    Affine Transformations
  - For properly chosen values of A, B, C and D.
  - A and C are scaling coefficients and
  - B and D are shifting coefficients.



## Drawing dot constellations

**Example : Simple Dot Plot – Complete Program (1/2)**

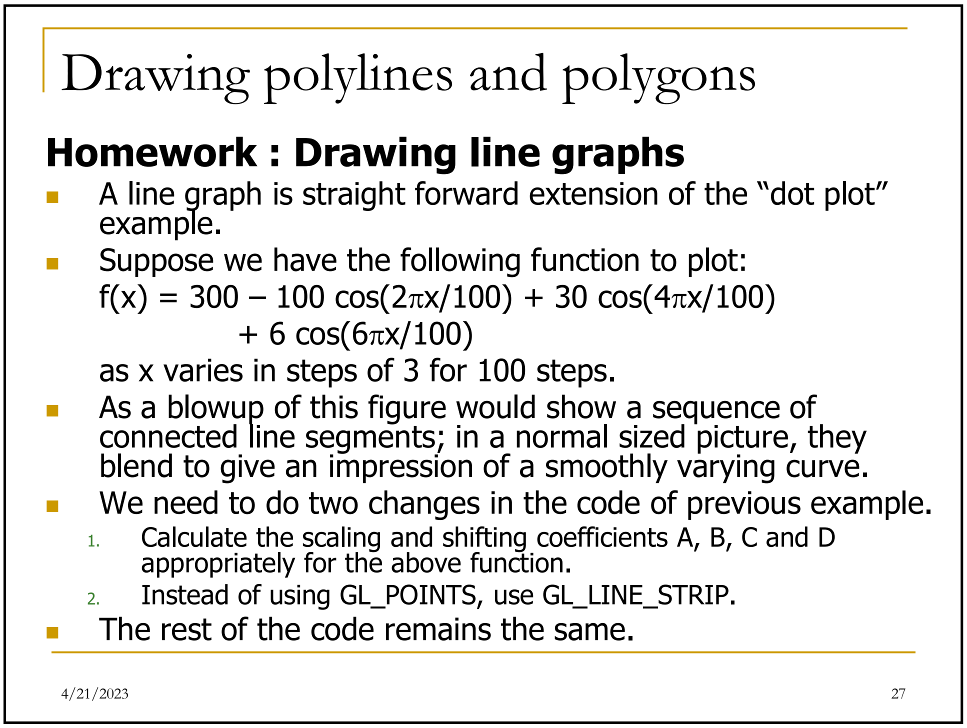
```
#include <math.h>
#include <gl/Gl.h>
#include <gl/glut.h>
const int screenWidth = 640; // width of the screen window in pixels
const int screenHeight =480; // height of the screen window in pixels
GLdouble A, B, C, D;                    // scaling and shifting coefficients
void myInit(void) {
        glClearColor(1.0, 1.0, 1.0, 0.0);    // background color is set to white
        glColor3f(0.0, 0.0, 0.0); // drawing color is set to black
        glPointSize(2.0);                 // a dot is 2 by 2 pixels
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, (GLdouble)screenWidth, 0.0, (GLdouble)screenWeight);
        A = screenWidth / 4.0;
        B = 0.0
        C = D = screenHeight / 2.0;
}
```



## Drawing dot constellations

**Example : Simple Dot Plot – Complete Program (2/2)**

```
void myDisplay(void) {
        glClear(GL_COLOR_BUFFER_BIT);                  // clear the screen
        glBegin(GL_POINTS);                            // draw the points
        for (GLdouble x=0; x<4.0; x += 0.005) {
              GLdouble func = exp(-x)*cos(2*3.14159265*x);
              glVertex2d( A*x + B, C*func + D );  }
        glEnd(); glFlush();
}
void main(int argc, char **argv) {
        glutInit(&argc, argv);                         // initialize the toolkit
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);   // set display mode
        glutInitWindowSize(screenWidth, screenHeight); // set window size
        glutInitWindowPosition(100, 150);         // set window position on screen
        glutCreateWindow(“Dot Plot of a Function”);
        glutDisplayFunc(myDisplay);               // register display function
        myInit();
        glutMainLoop();                           // go for a perpeptual loop
}
```



# Drawing polylines and polygons

## Homework : Drawing line graphs

- A line graph is straight forward extension of the “dot plot” example.
- Suppose we have the following function to plot:

  $f(x) = 300 – 100 \cos(2\pi x/100) + 30 \cos(4\pi x/100)$
  $+ 6 \cos(6\pi x/100)$

  as x varies in steps of 3 for 100 steps.
- As a blowup of this figure would show a sequence of connected line segments; in a normal sized picture, they blend to give an impression of a smoothly varying curve.
- We need to do two changes in the code of previous example.
  1. Calculate the scaling and shifting coefficients A, B, C and D appropriately for the above function.
  2. Instead of using GL_POINTS, use GL_LINE_STRIP.
- The rest of the code remains the same.