Panning and Zooming

Zooming

Zooming provides an option to the user to change the view of a document or image. It is applicable in two ways – Zooming in and Zooming out. It usually involves shrinking or stretching of pixels.

Zooming, panning and scaling are related to each other, as all three of them are used in context to a certain view on the screen. Zooming provides an option to the user to change the view of a document or image. It is applicable in two ways Zooming in and Zooming out.

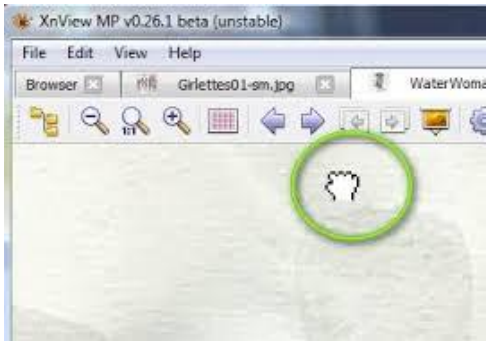
Zooming in allows the user to get the closer look of the particular area of interest on the screen. On the other hand, zooming out allows the user to have a larger view. Zooming in feature is often accompanied by scroll bars which appear when the screen is not wide enough to hold all of the content of the page. It is generally represented by showing a magnifying glass icon. A + sign depicts zoomed in, whereas – sign depicts zoomed out. In modern browsers, zooming is achieved by just stretching pixels. Generally, it involves the techniques in which pixel stretches widthwise and height wise both.

- Zooming that is not simply a change in size or scale like simple magnification
- Objects change fundamental appearance/presence at different zoom levels
- Zooming is like step function with boundaries where

Panning

Panning is another way of looking at the desired view. It is achieved by grabbing the document and then moving it around to focus on the desired or target area.

Panning is another way of looking at the desired view. It is achieved by grabbing the document and then moving it around to focus on the desired or target area. It makes accessibility to those areas which otherwise may remain hidden from the user.



Panning helps to quickly move around the area at the same magnification set by the user. It is usually represented by an icon showing a hand.

Scaling

Scaling resizes the image or text. There are numerous techniques which are used for scaling. Sometimes, scaling and zooming are used interchangeably.

Scaling is a process which aids a user to view a certain text or image. Scaling resizes the image or text. There are numerous techniques which are used for scaling. For example, in scaling method of nearest neighbor interpolation, every pixel is simply replaced by four pixels of the same size. Scaling is also used for performing zooming in and zooming out operation on digital images.

- Many data sets are too large to visualize on one screen
- May simply be too many cases
- May be too many variables
- May only be able to highlight particular cases or particular variables, but viewer’s focus may change from time to time

Zooming affects all the portions of the display image or text in a uniform manner. However, in scaling a coder can also use techniques like - resizing can be done by keeping the original height and altering the width or just vice versa. An image (or regions of an image) can be zoomed either through pixel replication or interpolation.

Scaling is used in context to the increase of decrease the physical size of the items. Unlike just stretching the pixels, it uses techniques like nearest neighbor interpolation for resizing. It is important to mention that the term scaling is also used for resizing of the image that occurs on the paper on which it needs to be printed, specifically without any type of pixel re sampling. It just stretches the image on the paper and do not change or alter actual pixels.

Comparison between Zooming, Panning and Scaling:

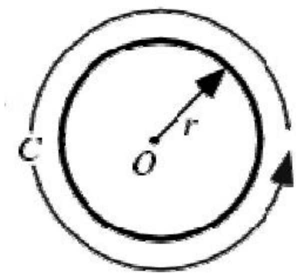
	zooming	**Panning**	**Scaling**
Definition	Zooming provides an option to the user to change the view of a document or image. It is applicable in two ways – Zooming in and Zooming out.	Panning is another way of looking at the desired view. It is achieved by grabbing the document and then moving it around to focus on the desired or target area.	Scaling resizes the image or text. There are numerous techniques which are used for scaling.
Common method	Stretching the pixels	Using the hand tool	Using techniques like nearest neighbor interpolation for resizing
In other contexts	Just the focus of camera is moved further or near the object. No change in the object’s length, width and depth	-A photography technique for capturing an object in motion	Modifies the width, length and depth of the object to make it smaller or larger

Circle Drawing Techniques

Circle

A circle is the set of points in a plane that are equidistant from a given point O. The distance r from the center is called the radius, and the point O is called the center. Twice the radius is known as the diameter d=2r. The angle a circle subtends from its center is a full angle, equal to 360° or 2πradians. A circle has the maximum possible area for a given perimeter, and the minimum possible perimeter for a given area. The perimeter C of a circle is called the circumference, and is given by

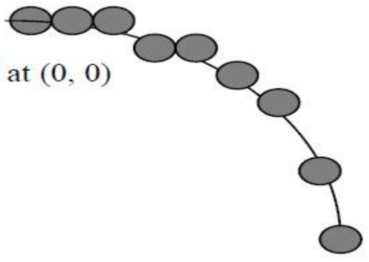
$$C = 2 \pi r$$



Circle Drawing Techniques

First of all for circle drawing we need to know what the input will be. Well the input will be one center point (x, y) and radius r. Therefore, using these two inputs there are a number of ways to draw a circle. They involve understanding level very simple to complex and reversely time complexity inefficient to efficient. We see them one by one giving comparative study.

Circle drawing using Cartesian coordinates



This technique uses the equation for a circle on radius r centered at (0, 0) given as:

$$x^2 + y^2 = r^2,$$

an obvious choice is to plot

$$y = \pm \sqrt{(r^2 - x^2)}$$

Obviously in most of the cases the circle is not centered at (0, 0), rather there is a center point (x_c, y_c); other than (0, 0). Therefore the equation of the circle having center at point (x_c, y_c):

$$(x - x_c)^2 + (y - y_c)^2 = r^2,$$

this implies that ,

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

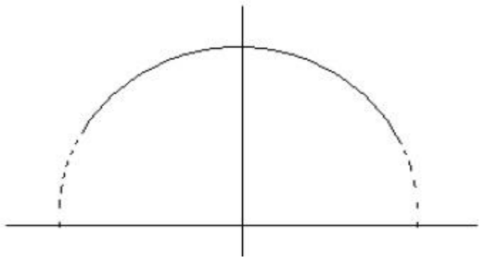
Using above equation a circle can easily be drawn. The value of x varies from r-xc to r+xc. and y will be calculated using above formula. Using this technique a simple algorithm will be:

Circle1 (xcenter, ycenter, radius)

```
for x = radius - xcenter to radius + xcenter

    $y = x_c + \sqrt{r^2 - (x - x_c)^2}$
                                    drawPixel (x, y)
    $y = x_c - \sqrt{r^2 - (x - x_c)^2}$
    drawPixel (x, y)
```

This works, but is inefficient because of the multiplications and square root operations. It also creates large gaps in the circle for values of x close to r as shown in the above figure.



Circle drawing using Polar coordinates

A better approach, to eliminate unequal spacing as shown in above figure is to calculate points along the circular boundary using polar coordinates r and θ. Expressing the circle equation in parametric polar form yields the pair of equations

$$x = x_c + r \cos \theta$$

$$y = y_c + r \sin \theta$$

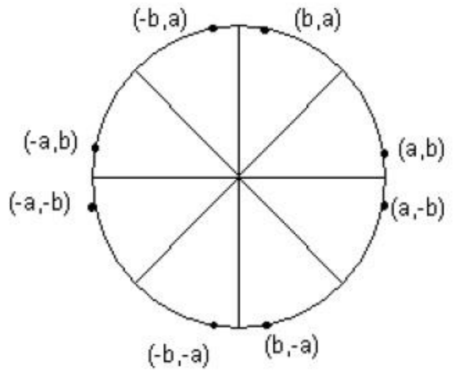
Using above equation circle can be plotted by calculating x and y coordinates as θ takes values from 0 to 360 degrees or 0 to 2π. The step size chosen for θ depends on the application and the display device. Larger angular separations along the circumference can be connected with straight-line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the step size at 1/r. This plots pixel positions that are approximately one unit apart.

Now let us see how this technique can be sum up in algorithmic form.

Circle2 (xcenter, ycenter, radius)

```
for θ = 0 to 2π step 1/r
        $x = x_c + r * \cos θ$
        $y = y_c + r * \sin θ$
        drawPixel (x, y)
```

Again this is very simple technique and also solves problem of unequal space but unfortunately this technique is still inefficient in terms of calculations involves especially floating point calculations.



Calculations can be reduced by considering the symmetry of circles. The shape of circle is similar in each quadrant. We can generate the circle section in the second quadrant of the xy-plane by noting that the two circle sections are symmetric with respect to the y axis and circle sections in the third an fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the x axis. We can take this one step further and note that there is also symmetry between octants. Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45o line dividing the two octants. These symmetry conditions are illustrated in above figure.

Therefore above algorithm can be optimized by using symmetric octants. Let’s see:

Circle2 (xcenter, ycenter, radius)

```
for θ = 0 to π / 4 step 1/r
      $x = x_c + r * \cos θ$
      $y = y_c + r * \sin θ$
      DrawSymmetricPoints (xcenter, ycenter, x, y)
```

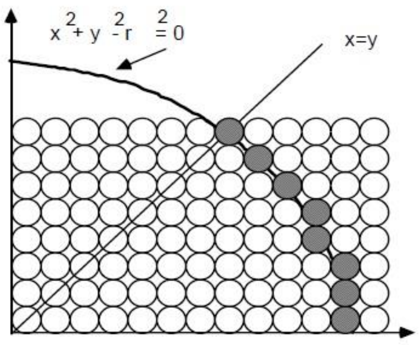
DrawSymmeticPoints (xcenter, ycenter, x, y)

```
Plot ( x + xcenter, y + ycenter )
Plot ( y + xcenter, x + ycenter )
Plot ( y + xcenter, -x + ycenter )
Plot ( x + xcenter, -y + ycenter )
Plot ( -x + xcenter, -y + ycenter)
Plot ( -y + xcenter, -x + ycenter)
Plot ( -y + xcenter, x + ycenter)
Plot ( -x + xcenter, y + ycenter)
```

Hence we have reduced half the calculations by considering symmetric octants of the circle but as we discussed earlier inefficiency is still there and that is due to the use of floating point calculations. In next algorithm we will try to remove this problem.

Midpoint circle Algorithm

As in the Bresenham line drawing algorithm we derive a decision parameter that helps us to determine whether or not to increment in the y coordinate against increment of x coordinate or vice versa for slope > 1. Similarly here we will try to derive decision parameter which can give us closest pixel position.



Let us consider only the first octant of a circle of radius r centred on the origin. We begin by plotting point (r, 0) and end when x < y. The decision at each step is whether to choose the pixel directly above the current pixel or the pixel; which is above and to the left (8-way stepping).

Assume:

Pi = (xi, yi) is the current pixel.

Ti = (xi, yi +1) is the pixel directly above

Si = (xi -1, yi +1) is the pixel above and to the left.

To apply the midpoint method, we define a circle function:

fcircle(x, y) = $x^2 + y^2 - r^2$

Therefore following relations can be observed:

$$f_{circle}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases}$$

The circle function tests given above are performed for the midpoints between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

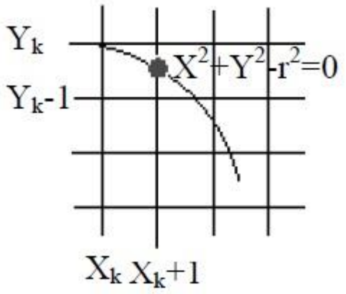


Figure given above shows the midpoint between the two candidate pixels at sampling position x_k+1. Assuming we have just plotted the pixel at (x_k, y_k), we next need to determine whether the pixel at position (x_k + 1, y_k), we next need to determine whether the pixel at position (x_k+1, y_k) or the one at position (x_k+1, y_k-1) is closer to the circle. Our decision parameter is the circle function evaluated at the midpoint between these two pixels:

$$P_k = f_{circle}(x_k + 1, y_k - ½)$$
$$P_k = (x_k + 1)^2 + (y_k - ½)^2 - r^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise, the mid position is outside or on the circle boundary, and we select the pixel on scan-line y_k-1.

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+2} = x_{k+1}+1 = x_k+1+1 = x_k+2$:

$$P_{k+1} = f_{circle}(x_{k+1} + 1, y_{k+1} - ½)$$
$$P_{k+1} = [(x_k + 1) + 1]^2 + (y_{k+1} - ½)^2 - r^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Subtracting (1) from (2), we get

$$P_{k+1} - P_k = [(x_k + 1) + 1]^2 + (y_{k+1} - ½)^2 - r^2 - (x_k + 1)^2 - (y_k - ½)^2 + r^2$$

or

$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_{k+1} - y^2_k) - (y_{k+1} - y_k) + 1$$

Where y_{k+1} is either y_k or y_k-1, depending on the sign of P_k. Therefore, if $P_k < 0$ or negative then y_{k+1} will be y_k and the formula to calculate P_{k+1} will be:

$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_k - y^2_k) - (y_k - y_k) + 1$$
$$P_{k+1} = P_k + 2(x_k + 1) + 1$$

Otherwise, if $P_k > 0$ or positive then y_{k+1} will be y_k-1and the formula to calculate P_{k+1} will be:

$$P_{k+1} = P_k + 2(x_k + 1) + [(y_k -1)^2 - y^2_k] - (y_k -1 - y_k) + 1$$
$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_k - 2 y_k +1 - y^2_k] - (y_k -1 - y_k) + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2 y_k + 1 + 1 +1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2 y_k + 2 +1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2 (y_k - 1) + 1$$

Now a similar case that we observe in line algorithm is that how would starting P_k be evaluated. For this at the start pixel position will be (0, r). Therefore, putting this value is equation, we get

$$P_0 = (0 + 1)^2 + (r - ½)^2 - r^2$$
$$P_0 = 1 + r^2 - r + ¼ - r^2$$
$$P_0 = 5/4 - r$$

If radius r is specified as an integer, we can simply round p_0 to:

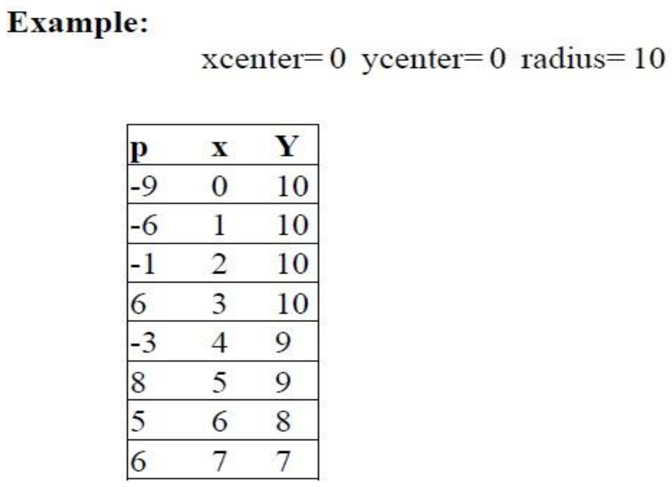
$$P_0 = 1 - r$$

MidpointCircle (xcenter, ycenter, radius)

```
y = r;
x = 0;
p = 1 - r;
do
    DrawSymmetricPoints (xcenter, ycenter, x, y)
    x = x + 1
    If p < 0 Then
    p = p + 2 * ( x + 1 ) + 1
    else
    y = y - 1
```

```
        p = p + 2 * ( x + 1 ) – 2 * ( y - 1 ) + 1
while ( x < y )
```

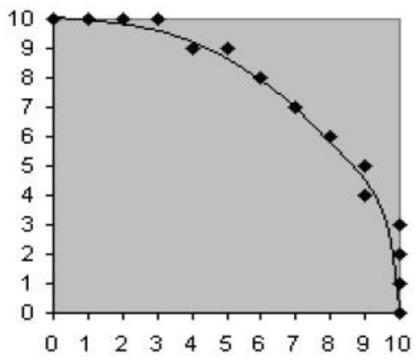
Now let us consider an example to calculate first octant of the circle using above algorithm; while one quarter is displayed where you can observe that exact circle is passing between the points calculated in a raster circle.



Example:

xcenter= 0 ycenter= 0 radius= 10

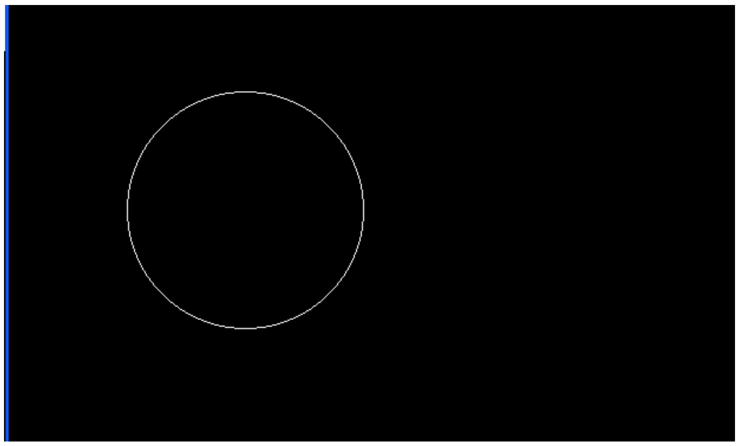
p	x	Y
-9	0	10
-6	1	10
-1	2	10
6	3	10
-3	4	9
8	5	9
5	6	8
6	7	7



THIS PROGRAM DISPLAY A CIRCLE

```
#include <graphics.h>#include <stdlib.h>
#include <stdio.h>      #include <conio.h>
int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;
  int midx, midy;
  int radius = 100;
  /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
  if (errorcode != grOk)  /* an error occurred */
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
    exit(1); /* terminate with an error code */
  }
  midx = getmaxx() / 2;
```

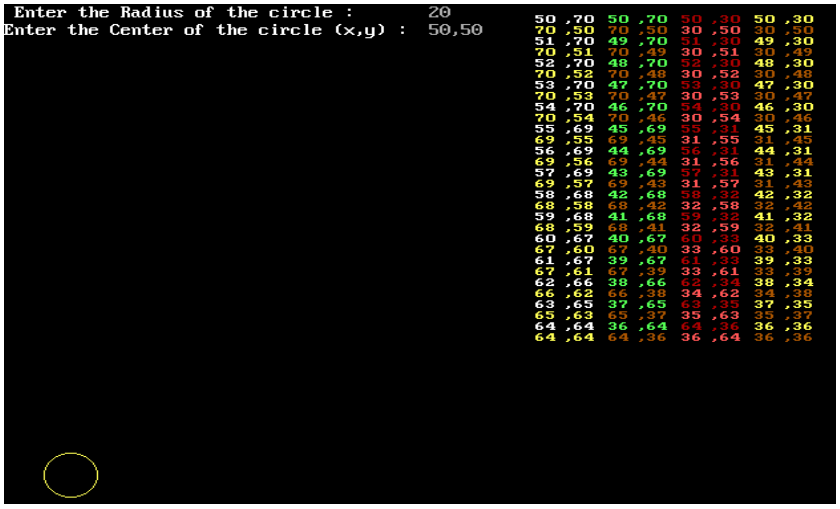
```
   midy = getmaxy() / 2;
   setcolor(getmaxcolor());
   /* draw the circle */
   circle(200,200, 100);
   /* clean up */
   getch();
   closegraph();
   return 0;
}
```



THIS PROGRAM SHOW A CIRCLE THROUGH MIDPOINT ALGORITHM).

```
#include <stdlib.h>     #include <graphics.h>
#include <math.h>       #include <conio.h>
#include <iostream.h>
int xc, yc, showat=10;
char ruf[10];
void tellpoint(int *num1, int *num2){
  int color; color=getcolor();
  int a,b;
  a = xc+*num1; b = getmaxy()-yc+*num2;
  outtextxy(400,showat,itoa(a,ruf,10));  outtextxy(422,showat,",");
outtextxy(429,showat,itoa(b,ruf,10));
  a=xc-*num1; b=getmaxy()-yc+*num2; setcolor(10);
  outtextxy(455,showat,itoa(a,ruf,10));  outtextxy(477,showat,",");
outtextxy(484,showat,itoa(b,ruf,10));
  a=xc+*num1; b=getmaxy()-yc-*num2; setcolor(20);
  outtextxy(510,showat,itoa(a,ruf,10));  outtextxy(532,showat,",");
outtextxy(539,showat,itoa(b,ruf,10));
  a=xc-*num1; b=getmaxy()-yc-*num2; setcolor(30);
  outtextxy(565,showat,itoa(a,ruf,10));  outtextxy(587,showat,",");
outtextxy(594,showat,itoa(b,ruf,10));
  showat+=10;
  a = xc+*num2; b = getmaxy()-yc+*num1; setcolor(110);
  outtextxy(400,showat,itoa(a,ruf,10));  outtextxy(422,showat,",");
outtextxy(429,showat,itoa(b,ruf,10));
  a=xc+*num2; b=getmaxy()-yc-*num1; setcolor(70);
  outtextxy(455,showat,itoa(a,ruf,10));  outtextxy(477,showat,",");
outtextxy(484,showat,itoa(b,ruf,10));
  a=xc-*num2; b=getmaxy()-yc+*num1; setcolor(140);
  outtextxy(510,showat,itoa(a,ruf,10));  outtextxy(532,showat,",");
outtextxy(539,showat,itoa(b,ruf,10));
  a=xc-*num2; b=getmaxy()-yc-*num1; setcolor(150);
  outtextxy(565,showat,itoa(a,ruf,10));  outtextxy(587,showat,",");
outtextxy(594,showat,itoa(b,ruf,10));
  setcolor(color); showat+=10;
}
void drawpoint(int x, int y){
  putpixel (xc+x, yc+y, 30);      putpixel (xc-x, yc+y, 30);     putpixel (xc+x, yc-y, 30);
  putpixel (xc-x, yc-y, 30);      putpixel (xc+y, yc+x, 30);     putpixel (xc-y, yc+x, 30);
  putpixel (xc+y, yc-x, 30)       putpixel (xc-y, yc-x, 30); }
void main(){
        int gdriver=DETECT, gmode, ecode;
        initgraph(&gdriver, &gmode, "d:\TC3");
        ecode = graphresult();
        if (ecode != grOk) {
                cout << "Graphic error ...";
                cout << "Press any key ...";
                getch();
                exit(1);}
        int x,y,r, Pk;
        char comma;
        cout <<" Enter the Radius of the circle :\t";
        cin >> r;
        cout << "Enter the Center of the circle (x,y) :\t";
        cin >>xc >>comma >>yc;
        yc = getmaxy()-yc;
        x=0;    y=r;
        drawpoint(x,y);
        tellpoint(&x, &y);
        Pk = 1 - r;
        while (x<y) {
```

```
 if (Pk<0)
x +=1;
 else { x +=1;
           y -=1;}
 drawpoint(x,y);
 tellpoint(&x, &y);
if (Pk<0)
  Pk = Pk + 2*x +1;
else
  Pk = Pk + 2*(x-y) +1;}
getch();}
```



```
 Enter the Radius of the circle :        20
Enter the Center of the circle (x,y) :  50,50
```

```
50 ,70 50 ,70 50 ,30 50 ,30
70 ,50 70 ,50 30 ,50 30 ,50
51 ,70 49 ,70 51 ,30 49 ,30
70 ,51 70 ,49 30 ,51 30 ,49
52 ,70 48 ,70 52 ,30 48 ,30
70 ,52 70 ,48 30 ,52 30 ,48
53 ,70 47 ,70 53 ,30 47 ,30
70 ,53 70 ,47 30 ,53 30 ,47
54 ,70 46 ,70 54 ,30 46 ,30
70 ,54 70 ,46 30 ,54 30 ,46
55 ,69 45 ,69 55 ,31 45 ,31
69 ,55 69 ,45 31 ,55 31 ,45
56 ,69 44 ,69 56 ,31 44 ,31
69 ,56 69 ,44 31 ,56 31 ,44
57 ,69 43 ,69 57 ,31 43 ,31
69 ,57 69 ,43 31 ,57 31 ,43
58 ,68 42 ,68 58 ,32 42 ,32
68 ,58 68 ,42 32 ,58 32 ,42
59 ,68 41 ,68 59 ,32 41 ,32
68 ,59 68 ,41 32 ,59 32 ,41
60 ,67 40 ,67 60 ,33 40 ,33
67 ,60 67 ,40 33 ,60 33 ,40
61 ,67 39 ,67 61 ,33 39 ,33
67 ,61 67 ,39 33 ,61 33 ,39
62 ,66 38 ,66 62 ,34 38 ,34
66 ,62 66 ,38 34 ,62 34 ,38
63 ,65 37 ,65 63 ,35 37 ,35
65 ,63 65 ,37 35 ,63 35 ,37
64 ,64 36 ,64 64 ,36 36 ,36
64 ,64 64 ,36 36 ,64 36 ,36
```

THIS PROGRAM MOVES A CIRCLE ALONG THE SCREEN

```
# include<iostream.h>          #include <graphics.h>
#include <stdlib.h>    #include <stdio.h>
#include <conio.h>     # include<dos.h>
int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;
  int xmax, ymax;
  /* initialize graphics and local variables */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode = graphresult();
  /* an error occurred */
  if (errorcode != grOk)
  {
     printf("Graphics error: %s\n", grapherrormsg(errorcode));
     printf("Press any key to halt:");
```

```
      getch();
      exit(1);
   }
   xmax = getmaxx();
   ymax = getmaxy();
  {        while(!kbhit())
{          for(int i=0;i<=500;i+=10)       {
                          delay(100);
                         cleardevice();
                         circle(i,50,50);}
for(i=550;i>=20;i-=10){
                         delay(100);
clear device();
                         circle(i-50,50,50);}
}
  /* clean up */
   getch();
   closegraph();
   return 0;
}
```