# FIFTEEN

# Geometrical techniques for graphics

As a change from programs handling data records, this chapter looks at two applications which produce graphical output. In both programs we will need to use geometrical techniques:

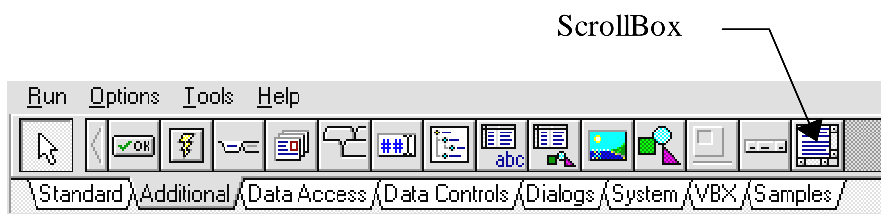
## Journey distance

A program is required which will display a map of the Meirionnydd coast area. It should be possible to mark out a route on the map using the mouse pointer, then the computer should display the distance travelled.

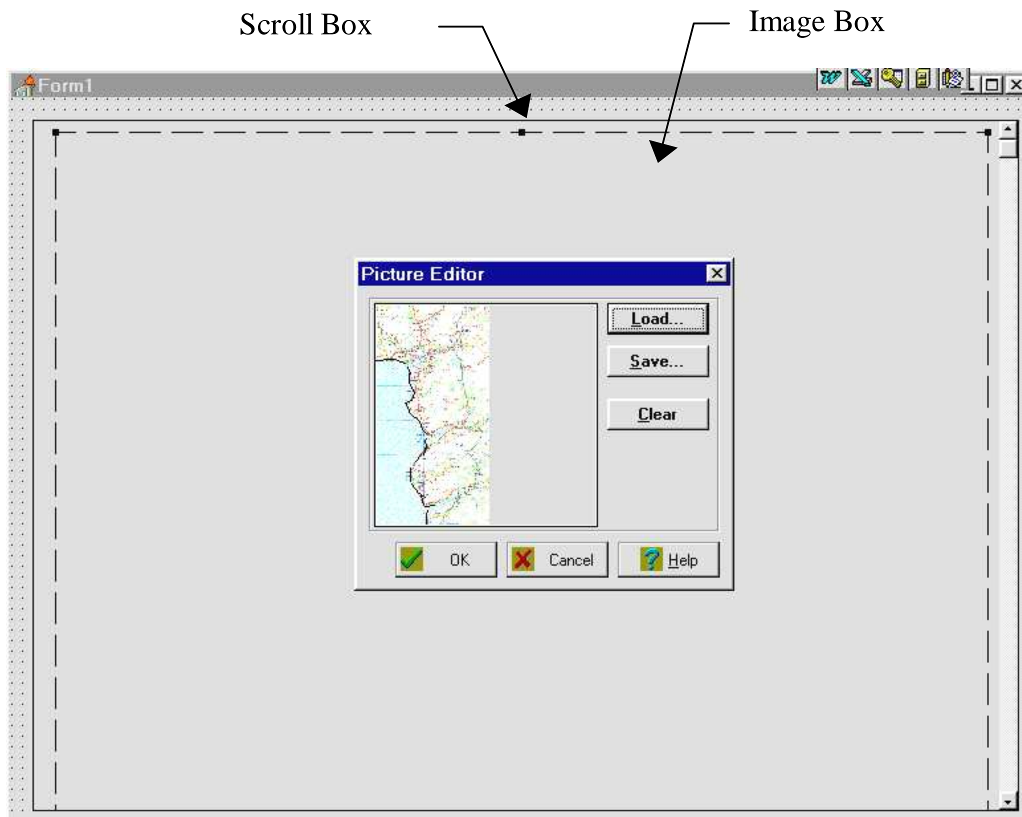
Begin the program by setting up a new directory MAPDIST and saving a Delphi project into it. Maximize the Form and drag the dotted grid to nearly fill the screen.

A map is provided for you as the file COAST.BMP. There is a slight problem because map image is larger than the computer screen, but we can get around this by using a *Scroll Box*:

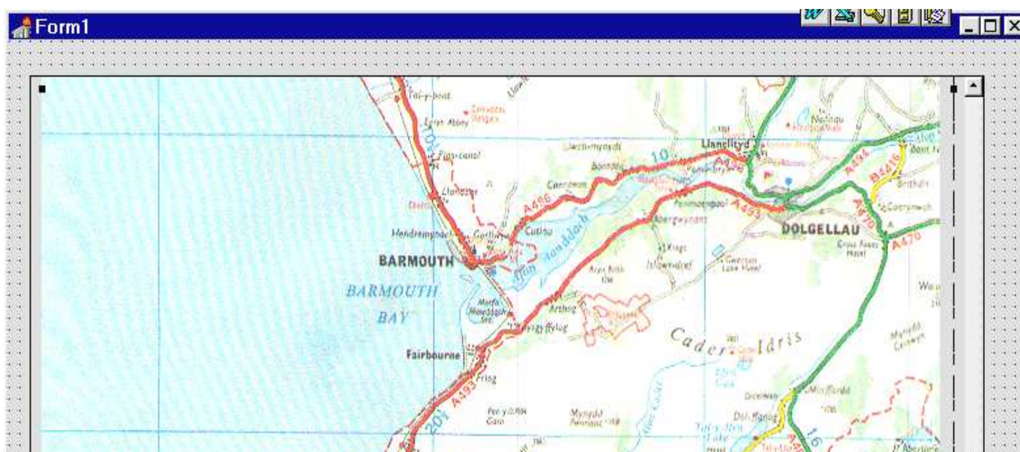
Begin by selecting the **Scroll Box** component from the ADDITIONAL menu:




Drag the *Scroll Box* to nearly fill the dotted grid of the Form as shown below. Press ENTER to bring up the Object Inspector, then double-click **VertScrollBar** to show the properties. Set the **Range** to **1400**.




Now place an **Image Box** inside the *Scroll Box.* You will find that by operating the vertical scroll bar, you will be able to drag the bottom edge of the *Image Box* downwards so that it is much larger than the screen height. Keep extending the *Image Box* so that the dotted outline extends nearly to the edges of all the scrolling screen area.

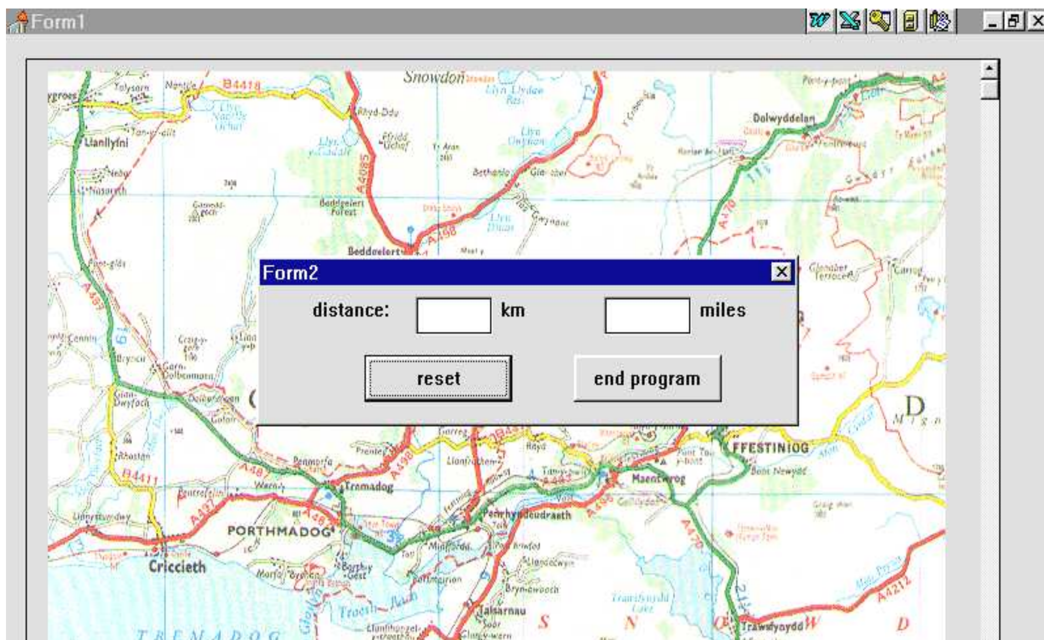



Double-click in the *Image Box* and load the file COAST.BMP. Check that the map fits the Image Box. Drag the *Scroll Box* and *Image Box* wider if necessary.

NOTE: It is important to keep the map the same size as the original bitmap image in the file, so that we can make accurate distance measurements. Don't alter the **Stretch** property - leave it as '**False**'.

Compile and run the program. Check that the whole map area from Blaenau Ffestiniog in the north to Tywyn in the south can be seen. Return to the Delphi editing screen.

Use the short-cut button to add a new blank Form. This will be a small window to show distances measured on the map:




Use the Object Inspector to set the properties for *Form2*:

| | |
|---|---|
| **BorderStyle** | **Dialog** |
| **FormStyle** | **StayOnTop** |
| **Visible** | **True** |

Add two *Edit Boxes*, and the *Labels*: '**distance:**', '**km**', and '**miles**'. Complete the Form by including two *Buttons* with the captions '**reset**' and '**end program**'.

Bring **Unit2** to the front and add a '**uses**' instruction under the *implementation* heading:

```
implementation
{$R *.DFM}
uses
  unit1;
```

Go now to **Unit1,** and add *Unit2* to the '**uses**' list near the top of the program:

```
uses
  SysUtils, WinTypes,....ExtCtrls, unit2;
```

This has linked the *Units* so that they are now able to exchange data when the program is running.

Bring Form2 to the front and double-click the 'end program' button to produce an event handler. Add the line:

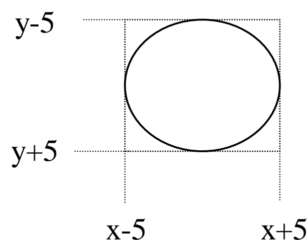
```
procedure TForm2.Button2Click(Sender: TObject);
begin
  halt;
end;
```

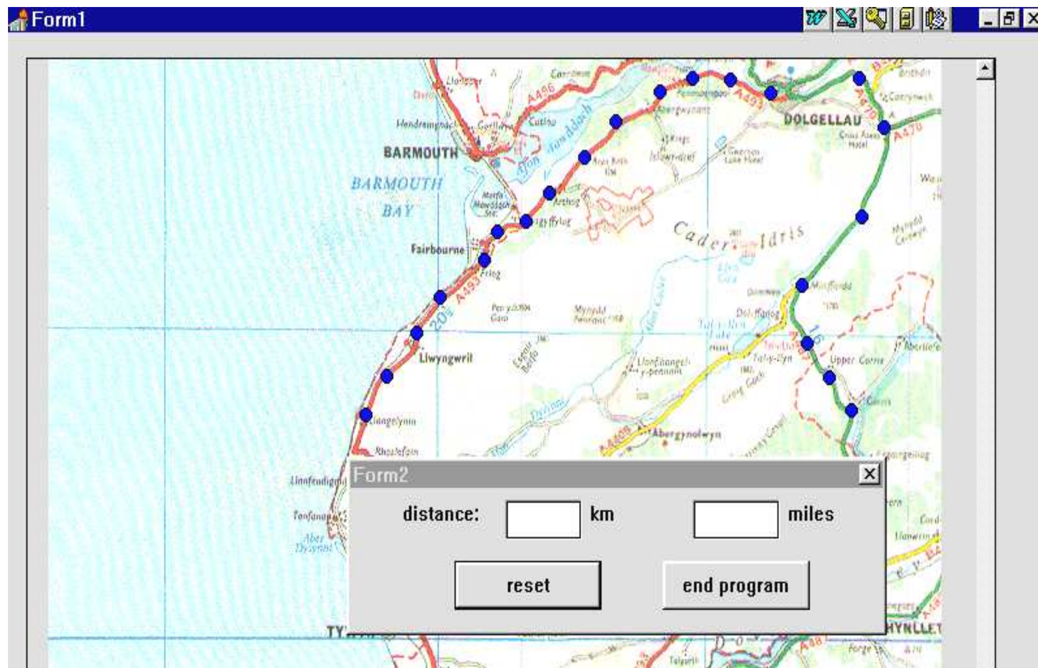
Build and run the program. Check that the *Form2* window appears on top of the map, then press the '**end program**' button to return to the Delphi editing screen.

Click to select the *Image Box* on *Form1*, then press ENTER to bring up the Object Inspector. Click the **Events** tab, then double-click alongside '**OnMouseDown**' to produce an event handler. Add the lines:

```
procedure TForm1.Image1MouseDown
        (Sender: TObject; Button: TMouseButton;
             Shift: TShiftState; X, Y: Integer);
begin
  image1.canvas.brush.color:=clBlue;
  image1.canvas.ellipse(x-5,y-5,x+5,y+5);
end;
```

The purpose of these lines is to produce a blue circle at the point where the mouse is clicked on the map. The computer records the position as ***x*** screen units across and ***y*** screen units down. We make use of these variables to draw the circle to fit within the boundaries:

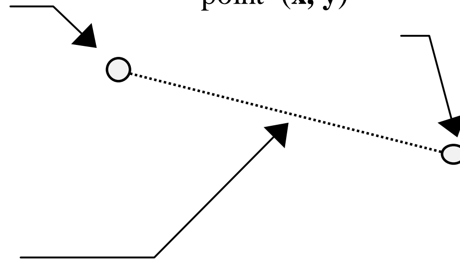




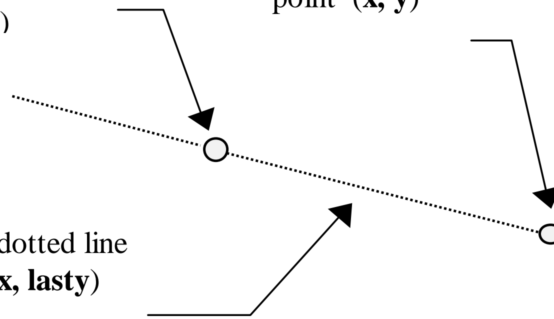

Build and run the program. By clicking the mouse on the map, it should be possible to plot a series of blue circles to mark out a route. Check that this works, then return to the Delphi editing screen.

It would be useful, particularly with routes across open country, to show the circles linked with a dotted line:

Three steps are needed to draw the first leg of the route:




The sequence can then be repeated for each of the remaining stages:




Return to the Unit1 program screen and add lines to the MouseDown event handler to do this:

```
procedure TForm1.Image1MouseDown
       (Sender: TObject; Button: TMouseButton;
            Shift: TShiftState; X, Y: Integer);
begin
  image1.canvas.brush.color:=clBlue;
  if firstpoint=true then
    firstpoint:=false
  else
```

```
    begin
      image1.canvas.pen.style:=psDot;
      image1.canvas.moveto(lastx,lasty);
      image1.canvas.lineto(x,y);
      image1.canvas.pen.style:=psSolid;
    end;
    image1.canvas.ellipse(x-5,y-5,x+5,y+5);
    lastx:=x;
    lasty:=y;
  end;
```

We are using a Boolean variable '**firstpoint**' which is **true** when the first point on the route has just been entered - this must be treated differently because no dotted line is needed. We just set '**firstpoint**' to **false**, ready for the next time a point is entered:

```
if firstpoint=true then
   firstpoint:=false
```

For other points along the route, a dotted line is drawn to link the current point to the previous point. This is done with:

```
image1.canvas.pen.style:=psDot;
image1.canvas.moveto(lastx,lasty);
image1.canvas.lineto(x,y);
```

At the end of the precedure, we record the position of the circle we have just drawn as (**lastx, lasty**), ready for linking to the next one along the route:

```
lastx:=x;
lasty:=y;
```

Add the variables to the *Public declarations* section near the top of the program:

```
public
   { Public declarations }
   lastx,lasty:integer;
   firstpoint:boolean;
```

One final task is to initialise '**firstpoint**' to **true** when the program starts. Double-click the Form1 dotted grid to produce an '**OnCreate**' procedure, then add the line:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  firstpoint:=true;
end;
```

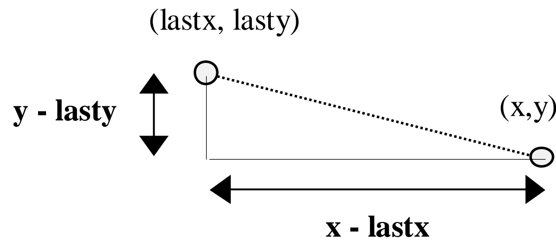
Build and run the program. When a route is entered, the points should now be linked by a dotted line. Return to the Delphi editing screen.

It would be useful to be able to reload a clean copy of the map, ready for a different route to be entered. Bring **Form2** to the front and double-click the '**reset**' button to create an event handler. Add the lines:

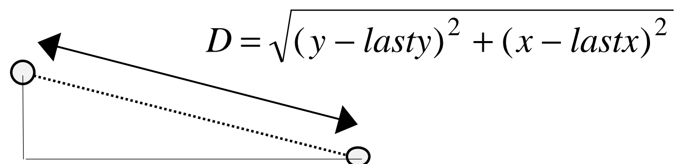
```
procedure TForm2.Button1Click(Sender: TObject);
begin
  form1.firstpoint:=true;
  form1.image1.picture.loadfromfile('coast.bmp');
  edit1.text:='';
end;
```

Build and run the program. Enter a route, then check that the map can be cleared by pressing the '**reset**' button. Return to the Delphi editing screen.

We can now start work on the calculation of journey distance. We know the screen coordinates of each point along the route, so we can work out how far we have moved vertically and horizontally on the map:




The straight line distance '*as the crow flies*' can be found using Pythagoras' theorem. We add the squares of the horizontal and vertical changes in distance, then find the square root:



$$D = \sqrt{(y - lasty)^2 + (x - lastx)^2}$$

One final difficulty is that the distance will be measured in screen units. We will need a conversion factor to convert this to kilometres or miles on the map. The map scale is :

**20 screen units = 1 kilometre**

Go to the bottom of **Unit1**, just above the final '**end.**' command, and add a new procedure to carry out the distance calculation:

```
procedure TForm1.calculate
                      (x,y,lastx,lasty:real);
var
  d:real;
begin
  scale:=1/20;
  d:=sqrt(sqr(x-lastx)+sqr(y-lasty));
  distance:=distance+d*scale;
  form2.edit1.text:=
          floattostrf(distance,fffixed,8,1);
end;
```

The first line of the procedure:

**procedure TForm1.calculate(lastx, lasty, x, y :real);**

is showing that the four coordinate values **lastx, lasty, x, y** need to be input to the procedure for use in the calculation.

The line:

**d:=sqrt(sqr(x-lastx)+sqr(y-lasty));**

uses the Pythagoras formula to calculate the distance in screen units between the current pair of points, then the line:

**distance:=distance+d*scale;**

converts this to kilometres and adds it to the journey distance so far. The total distance is then displayed in the *Edit Box* on *Form2*:

**form2.edit1.text := floattostrf(distance,fffixed,8,1);**

Add the procedure to the list near the top of Unit1:

```
type
   ...........
   procedure FormCreate(Sender: TObject);
   procedure calculate(x,y,lastx,lasty:real);
```

and add the variables to the Public declarations:

```
  { Public declarations }
    lastx,lasty:integer;
    firstpoint:boolean;
    distance,scale:real;
```

It is necessary to add lines to the '**MouseDown**' procedure to initialise the distance to zero when the first point is entered, and to call the calculate procedure each time another point along the route is entered. The procedure becomes:

```
procedure TForm1.Image1MouseDown
         (Sender: TObject; Button: TMouseButton;
             Shift: TShiftState; X, Y: Integer);
begin
  image1.canvas.brush.color:=clBlue;
  if firstpoint=true then
  begin
    distance:=0;
    firstpoint:=false;
  end
  else
  begin
    image1.canvas.pen.style:=psDot;
    image1.canvas.moveto(lastx,lasty);
    image1.canvas.lineto(x,y);
    image1.canvas.pen.style:=psSolid;
    calculate(x,y,lastx,lasty);
  end;
     .......
```

Build and run the program. Test this by entering the route from Dolgellau to Porthmadog along the main road via Trawsfynydd. Press '**reset**', then find the distance along the alternative route via Barmouth and the toll bridge at Penrhyndeudraeth. Return to the Delphi editing screen.

It just remains to show the distance in miles as well as kilometres. To convert from kilometres to miles it is necessary to multiply by 0.62137. Add lines to the calculate procedure to do this:

```
procedure TForm1.calculate(x,y,lastx,lasty:real);
var
  d,miles:real;
begin
  scale:=1/20;
  d:=sqrt(sqr(x-lastx)+sqr(y-lasty));
  distance:=distance+d*scale;
  miles:=distance*0.62137;
  form2.edit1.text:=floattostrf(distance,fffixed,8,1);
  form2.edit2.text:=floattostrf(miles,fffixed,8,1);
end;
```

Build an run the finished program. Check that distances are now displayed in **miles** as well as kilometres.

For the next project we will look at graphical techniques for animation, taking an example from engineering.

Now that fast computers with good graphics are available, engineers are making more and more use of animation when designing machinery. Computer animations can be used to check that moving parts will not come into collision with each other while the machine is running, and that valves and switches will operate at the correct moments - this is known as '***kinematic analysis***'.

## Diesel engine animation

You are asked to produce a computer animation of a diesel engine, showing the motion of the piston and valves.

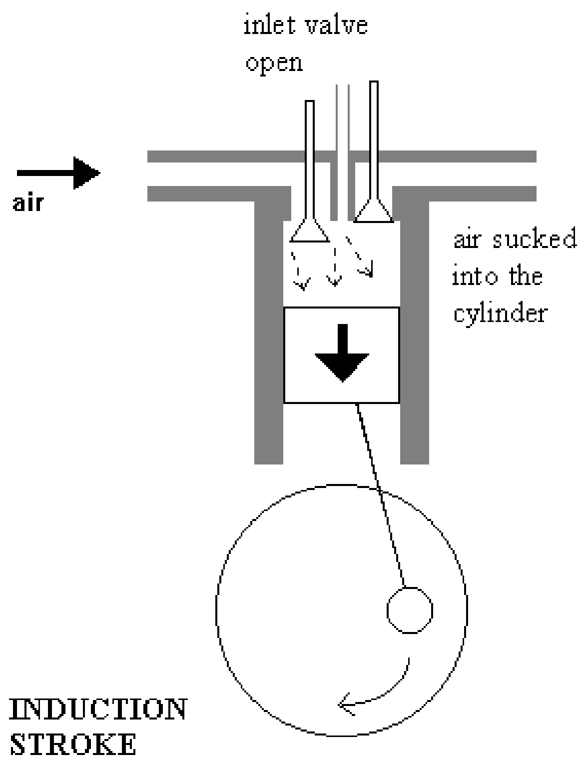
The stages of the diesel engine cycle are shown on the following page. The complete cycle involves two rotations of the flywheel:

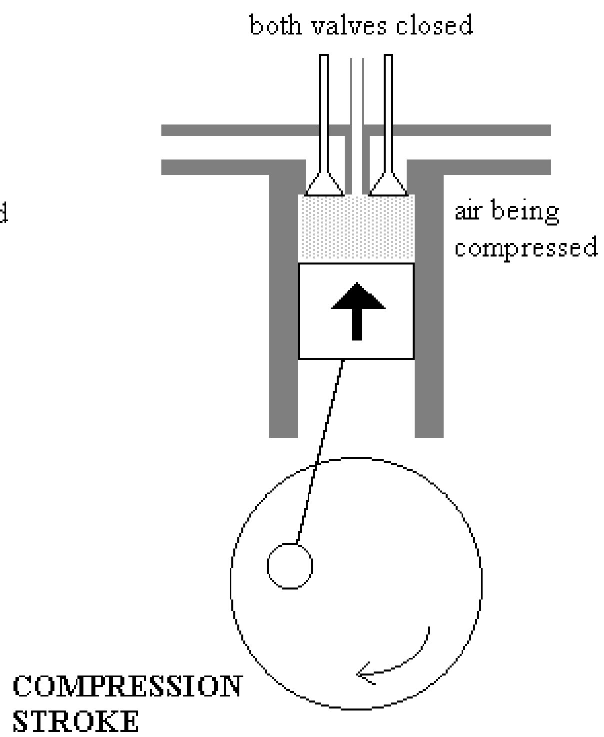
1. The **induction stroke.** The inlet valve is open, and air is sucked into the cylinder as the piston moves downwards.
2. The **compression stroke**. The inlet valve closes, and the air in the cylinder is compressed as the piston moves upwards.
3. The **power stroke**. Fuel is sprayed into the cylinder. This ignites, and the explosion drives the piston downwards.
4. The **exhaust stroke**. The exhaust valve opens, and the exhaust gases are pushed out of the cylinder as the piston moves upwards again.

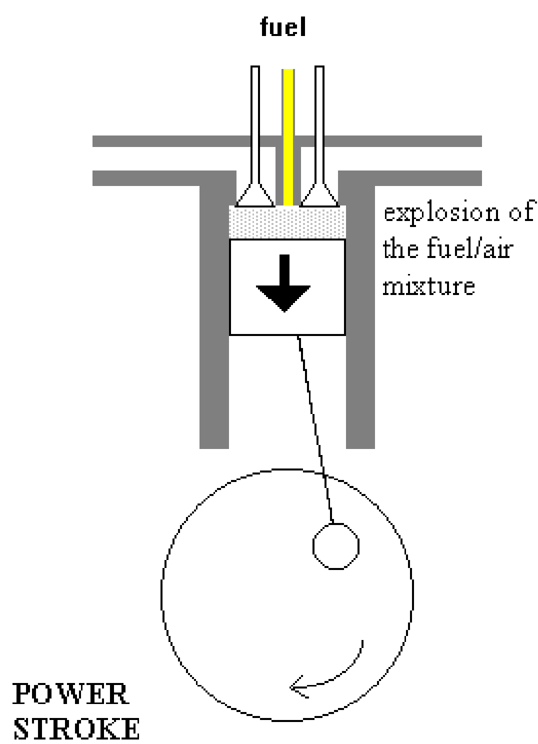
Begin the program by setting up a new directory DIESEL and saving a Delphi project into it. Use the Object Inspector to **Maximize** the screen, and drag the dotted grid to nearly fill the screen. Add an **Image Box** to the Form. Set the **Width** property to **640** and the **Height** property to **480**.

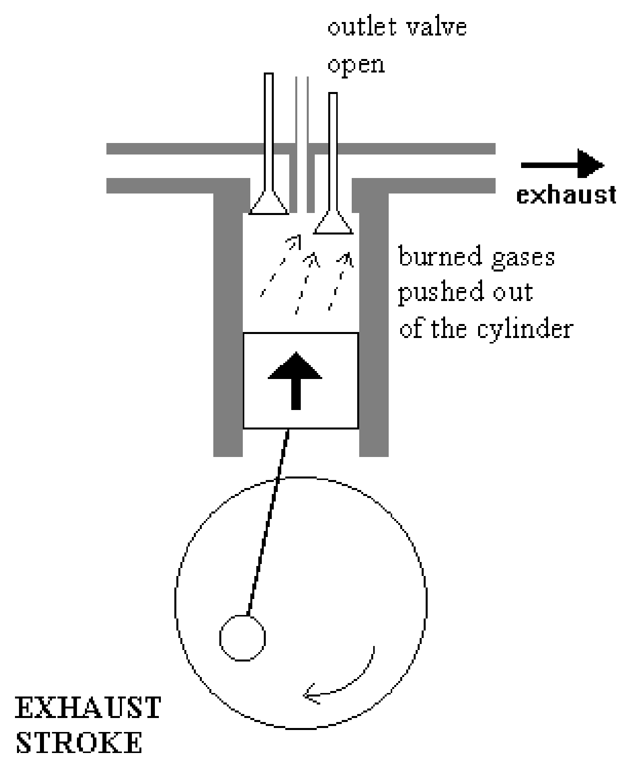
Double-click the dotted grid of the form outside the Image Box to produce an '**OnCreate**' event handler. Add the line:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  image1.canvas.rectangle(0,0,640,480);
end;
```


inlet valve
open
air
air sucked
into the
cylinder
INDUCTION
STROKE

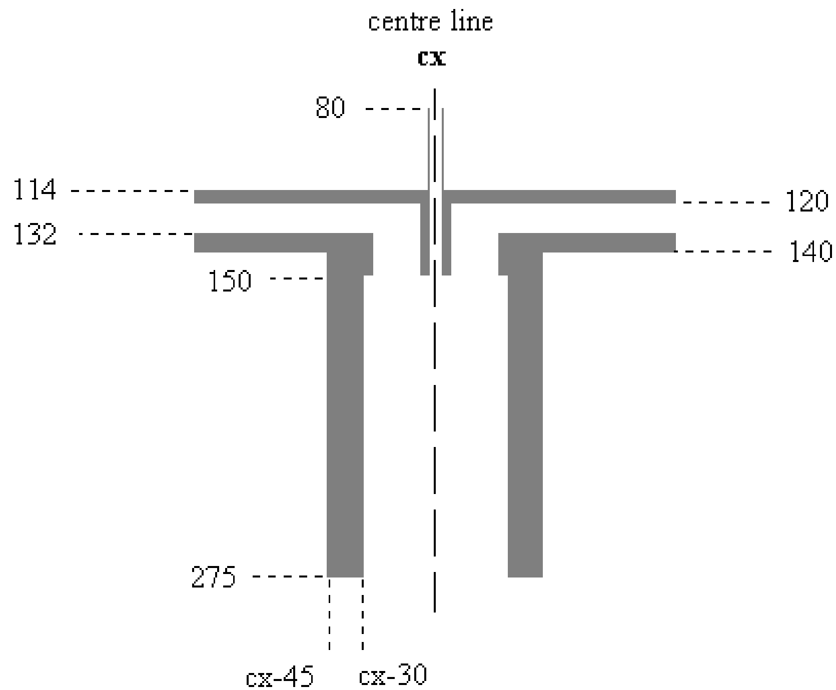

both valves closed
air being
compressed
COMPRESSION
STROKE


fuel
explosion of
the fuel/air
mixture
POWER
STROKE


outlet valve
open
exhaust
burned gases
pushed out
of the cylinder
EXHAUST
STROKE

Compile and run the program to check that a white background area is displayed, then return to the Delphi editing screen.

Looking at the drawings on the previous page, it is possible to divide the engine into *moving* and *non-moving* parts. It is simplest to begin work on the non-moving components:




There is a mirror image symmetry through the centre of the design, so it will be easiest to make measurements relative to this. For example, the inside edges of the cylinder can be drawn 30 units to the **left** and **right** of the centre line. Other suitable screen coordinates are shown on the diagram.

Go to the *Unit1* program screen and add a **constant** to represent the middle of the drawing area. The *image box* has a width of 640 screen units, so the mid line will be at 320:

```
    { Public declarations }
end;

var
  Form1: TForm1;
const
  cx=320;
```

Go to the bottom of the program and add a new procedure '**engine**' to draw the non-moving parts:

```
procedure TForm1.engine;
begin
  with image1.canvas do
  begin
    brush.color:=clGray;
    pen.color:=clGray;
    rectangle(cx-30,140,cx-45,275);
    rectangle(cx+30,140,cx+45,275);
    rectangle(cx-100,140,cx-30,132);
    rectangle(cx+100,140,cx+30,132);
    moveto(cx-3,80);
    lineto(cx-3,150);
    moveto(cx+3,80);
    lineto(cx+3,150);
    rectangle(cx-3,150,cx-6,114);
    rectangle(cx+3,150,cx+7,114);
    rectangle(cx-100,114,cx-6,120);
    rectangle(cx+100,114,cx+6,120);
    rectangle(cx-30,132,cx-26,150);
    rectangle(cx+30,132,cx+26,150);
  end;
end;
```

Notice that we have used a line:

**with image1.canvas do . . . .**

This avoids having to write the '**image1.canvas**' prefix in front of each individual graphics command.

Add the procedure to the list near the top of the program:

```
type
  TForm1 = class(TForm)
    Image1: TImage;
    procedure FormCreate(Sender: TObject);
    procedure engine;
```

Go to the **FormCreate** event handler and include a line to call the '**engine**' procedure:

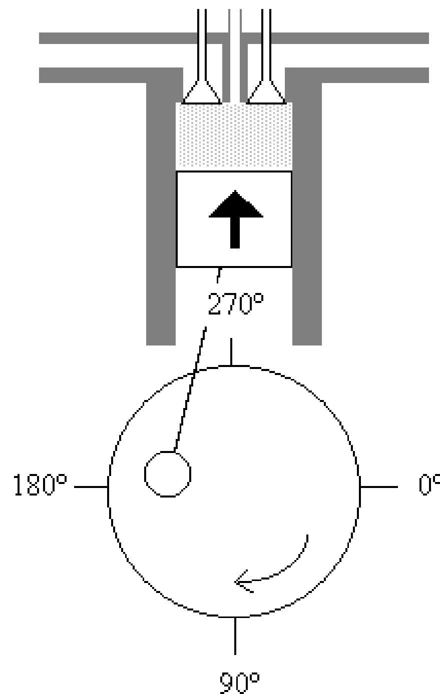
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  image1.canvas.rectangle(0,0,640,480);
  engine;
end;
```

Compile and run the program to check that the graphics are drawn correctly. The cylinder, inlet and outlet pipes and the fuel inlet should be shown. Return to the Delphi editing screen.

The next step is to draw the *moving parts* - the flywheel, piston and connecting rod assembly. This can be done in another procedure called '**piston**'. Add this to the procedure list near the top of the program:

```
type
TForm1 = class(TForm)
  Image1: TImage;
  procedure FormCreate(Sender: TObject);
  procedure engine;
  procedure piston(angle:integer);
```

It will be convenient to use a parameter **'angle'** to specify how far the flywheel has rotated. We can measure the angle clockwise from the horizontal position:




Go to the bottom of the program and insert the '**piston**' procedure shown on the next page. This draws the circle of the flywheel with a radius of 65 screen units, centred at a point 350 units down the screen.

We will be making a number of measurements relative to the centre of the flywheel, so it is convenient to use a variable '**cy**' to record the vertical screen coordinate for the centre of the circle:

```
procedure TForm1.piston(angle:integer);
var
  cy:longint;
begin
  cy:=350;
  with image1.canvas do
  begin
    brush.color:=clWhite;
    pen.color:=clBlack;
    ellipse(cx-65,cy-65,cx+65,cy+65);
  end;
end;
```

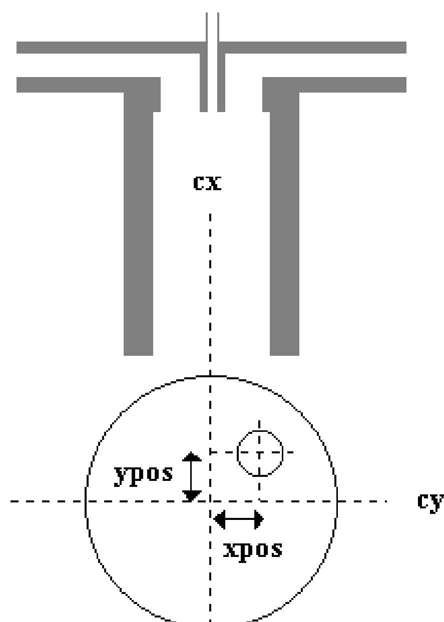
Insert lines in the '**FormCreate**' event handler to initialise the **angle** to zero, then call the '**piston**' procedure:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  image1.canvas.rectangle(0,0,640,480);
  engine;
  angle:=0;
  piston(angle);
end;
```

Add '**angle**' to the *Public declarations* section:

```
public
   { Public declarations }
   angle:integer;
```

Compile and run the program to check that a circle is drawn for the flywheel, then return to the Delphi editing screen.




We must now draw the small circle representing the end of the connecting rod attached to the flywheel. Its position will depend on the rotation angle of the flywheel.

It will be useful to set up two new variables ***xpos*** and ***ypos*** to give the position of the small circle relative to the centre of the flywheel.

Add lines to the **piston** procedure to calculate **xpos** and **ypos**, and draw the small circle:

```
procedure TForm1.piston(angle:integer);
var
  cy,xpos,ypos:longint;
  rad:real;
begin
  cy:=350;
  with image1.canvas do
  begin
    brush.color:=clWhite;
    pen.color:=clBlack;
    ellipse(cx-65,cy-65,cx+65,cy+65);
    rad:=angle*pi/180;
    xpos:=round(35*cos(rad));
    ypos:=round(35*sin(rad));
    ellipse(cx+xpos-12,cy+ypos-12,
                              cx+xpos+12,cy+ypos+12);
  end;
end;
```

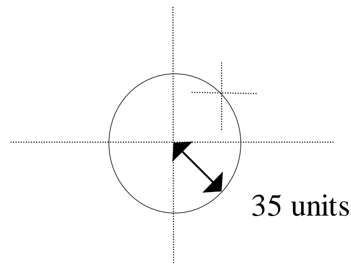
It is necessary to convert the angle from degrees to radians before carrying out the calculation. This is done in the line:

**rad:=angle*pi/180;**

Values for **xpos** and **ypos** are calculated using COSINE and SINE functions.

**xpos:=round(35*cos(rad));**
**ypos:=round(35*sin(rad));**

We have made the small circle follow a path which is 35 screen units out from the centre of the flywheel:




The small circle is then drawn with a radius of 12 units:

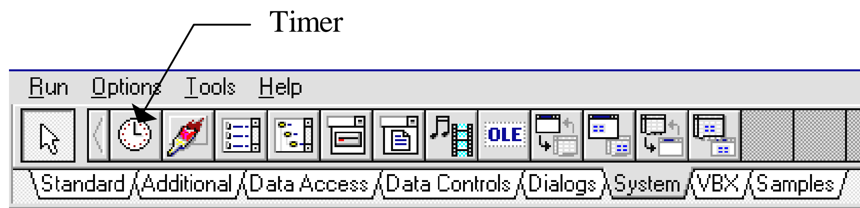
**ellipse(cx+xpos-12,cy+ypos-12, cx+xpos+12,cy+ypos+12);**

The extra distances **xpos** and **ypos** are added to the flywheel centre position (**cx**,**cy**) to give the centre for the small circle.

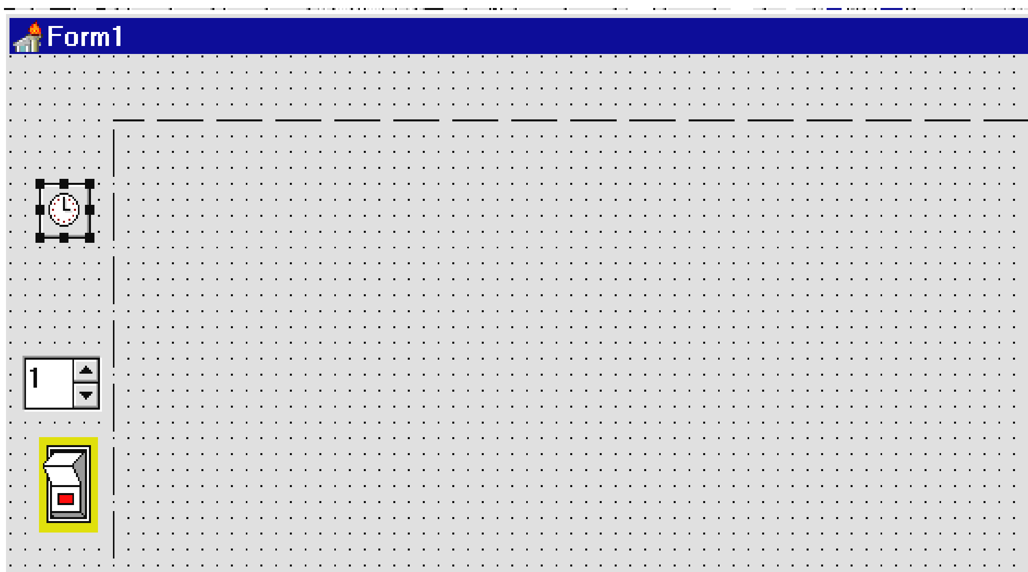
Compile and run the program to check that the small circle is drawn, then return to the Delphi editing screen.

We have been able to produce the basic shape of the engine, so now we can begin work on the animation. This will require some extra components to be added to the form.

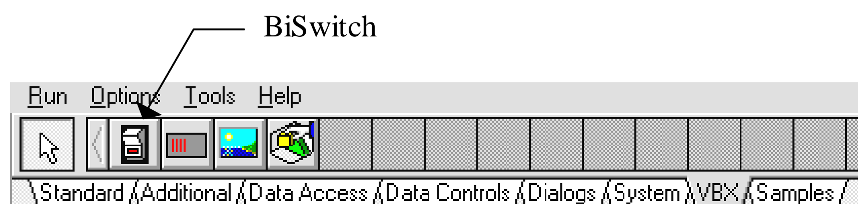
Go first to the SYSTEM menu and select the **Timer** component:




As in the case of the *Main Menu* which we used in a previous chapter, the **Timer** component appears as a fixed size icon. Place the *Timer icon* near the edge of the *Form* grid:




Now add a Spin Edit component at the edge of the Form. Finally, go to the VBX component menu and select a Bi Switch:




Place this below the Spin Edit as shown.

Click on the Timer icon and press ENTER to bring up the Object Inspector. Set the **properties:**

| | |
|---|---|
| **Enabled** | **True** |
| **Interval** | **200** |

The timer will be used to measure the intervals between redrawing the engine in each rotation position. The interval time is given in thousandths of a second, so we have set a time of 200/1000 - this is one fifth of a second.

Close the Object Inspector, then double-click the **Timer** icon to produce an event handler procedure. Add the lines:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  engine;
  angle:=angle+15;
  if angle>720 then
    angle:=angle-720;
  piston(angle);
end;
```

This procedure will be activated every fifth of a second - each time the interval set on the **Timer** is completed. The non-moving parts will be redrawn with:

**engine;**

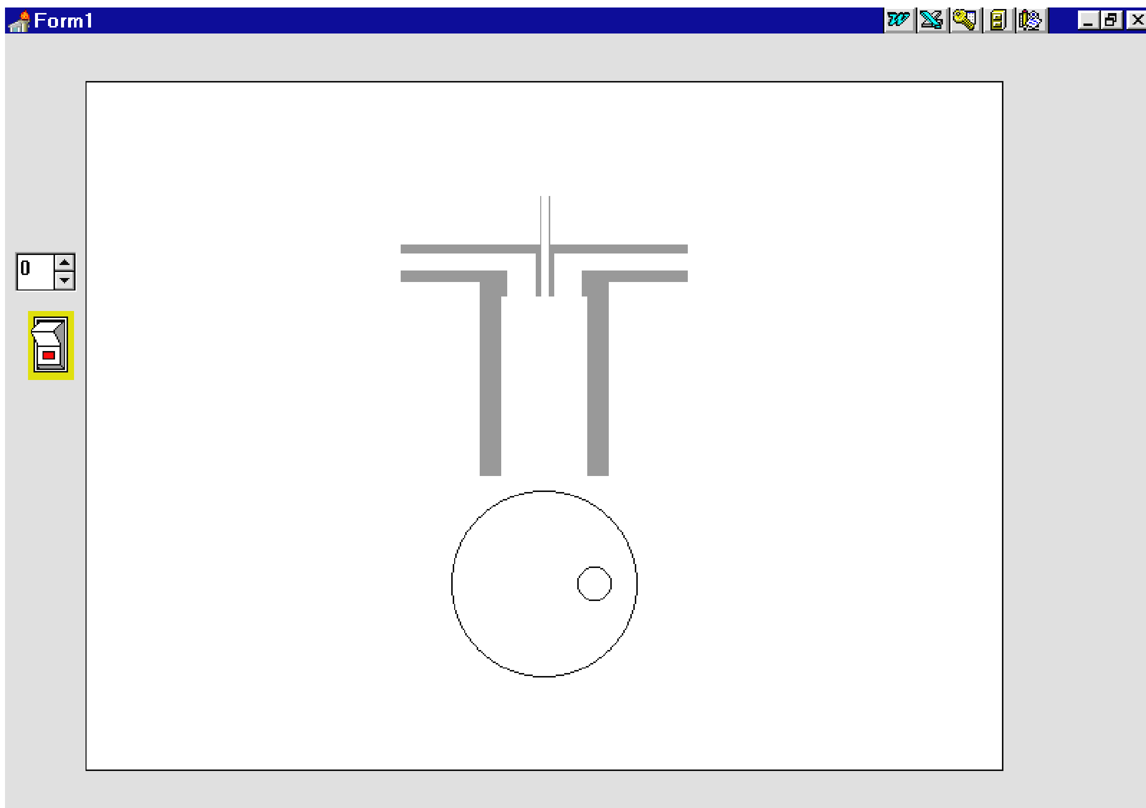
Fifteen degrees are then added to the **angle**. The diesel engine cycle involves two complete turns of the flywheel before repeating. We can record the position in the cycle by allowing the angle to increase up to 720°, but it must then be reset to zero:

```
angle:=angle+15;
if angle>720 then
    angle:=angle-720;
```

We finally use the **piston** procedure to draw the small circle in its correct rotation position on the flywheel:

**piston(angle);**

Compile and run the program. The engine should be shown with the small circle rotating around the flywheel. Check also that the *Spin Edit* value can be changed, and that the *Bi Switch* can be clicked to the **on** or **off** position. Return to the Delphi editing screen.




The **Bi Switch** is to allow the animation to be switched on and off. Click on the BiSwitch component and press ENTER to bring up the Object Inspector. Set the '**pOn**' property to **True**. This initialises the switch to be in the 'on' position when the program starts.

Close the Object Inspector then double-click the BiSwitch to produce an event handler. Add the line:

```
procedure TForm1.BiSwitch1On(Sender: TObject);
begin
  timer1.enabled:=true;
end;
```

This starts the *Timer* to run the animation whenever the *BiSwitch* is in the '**on**' position. We also need an event handler to stop the timer when the BiSwitch is '**off**'. Go back to the Form window and click the switch component. Press ENTER to bring up the Object Inspector, then click the **Events** tab. Double-click alongside '**OnOff**' to produce an event handler, then add the line:

```
procedure TForm1.BiSwitch1Off(Sender: TObject);
begin
  timer1.enabled:=false;
end;
```

Compile and run the program. The *BiSwitch* should begin in the '**on**' position, with the red indicator showing. Click the switch and the animation should pause. Click again and it should re-start. When you have checked that this is working correctly, return to the Delphi editing screen.

The purpose of the **SpinEdit** is to control the speed of the animation. Click on the SpinEdit and press ENTER to bring up the Object Inspector. Set the properties:

| | |
|---|---|
| **MaxValue** | **10** |
| **MinValue** | **1** |
| **Hint** | **speed control** |
| **ShowHint** | **True** |
| **EditorEnabled** | **False** |

Compile and run the program. You should find that the number in the *SpinEdit* window can only be changed by clicking the small arrows alongside - we have disabled the window so that values cannot be typed directly from the keyboard.

If you leave the mouse pointer stationary on the SpinEdit box for a couple of seconds, a yellow hint label with the text '***speed control***' will appear. Hint labels can be added to most components by setting their **Hint** and **ShowHint** properties with the Object Inspector.

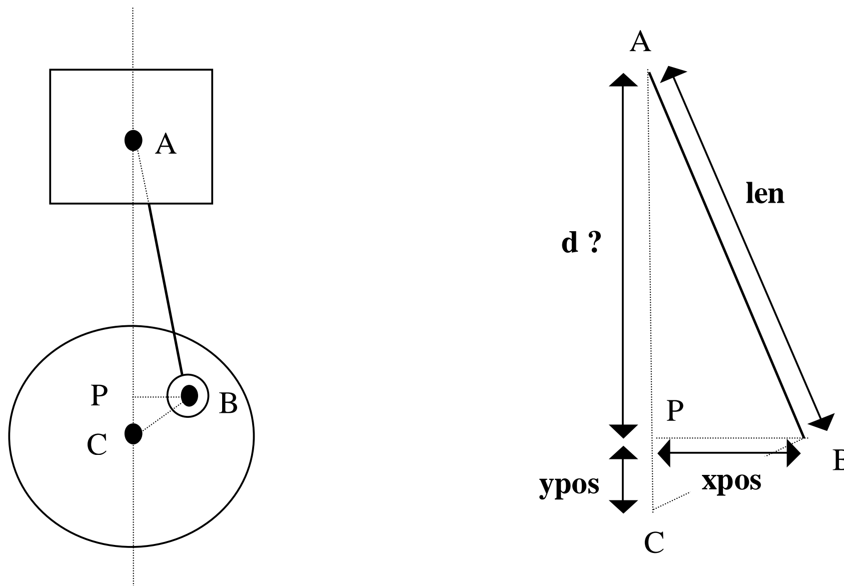
Return to the Delphi editing screen and double-click the SpinEdit to produce an event handler. Add the line:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
begin
  timer1.interval:=(11-spinedit1.value)*20;
end;
```

This takes the value in the *SpinEdit* box and calculates a time interval for the *Timer*. The formula has been written in such a way that the larger the number in the *SpinEdit* box, the shorter the time intervals between redrawing the pictures. This means that the simulation will run faster as the *SpinEdit* value increases.

Compile and run the simulation. Check that the animation can be speeded up or slowed down by changing the *SpinEdit*. Return to the Delphi editing screen.

The next step is to draw the piston and connecting rod in their correct positions as the flywheel rotates. The mathematics for this will require some careful planning!




When the animation is running we will need to draw a rectangle to represent the piston and a line to represent the connecting rod, as shown in the left hand diagram. These must be positioned correctly on the screen.

For any rotation angle, we already know the position of point B where the connecting rod is joined to the flywheel - this is given by the variables **xpos** and **ypos**, relative to the centre of the flywheel at C. To complete the graphics we need to know the distance **d -** the extra distance up the screen to the top of the connecting rod.

We can choose the length of the connecting rod; 140 screen units would be suitable. It is then possible to calculate **d** using Pythagoras' formula on the right-angled triangle ABP in the diagram:

$$d = \sqrt{len^2 - xpos^2}$$

Add lines to the **piston** procedure to carry out this calculation and draw the rectangle and line. The procedure becomes:

```
procedure TForm1.piston(angle:integer);
var
  cy,xpos,ypos,len,py:longint;
  rad,d:real;
begin
  cy:=350;
  len:=140;
  with image1.canvas do
  begin
```

```
        brush.color:=clWhite;
        pen.color:=clWhite;
        rectangle(cx-30,150,cx+30,320);
        pen.color:=clBlack;
        ellipse(cx-65,cy-65,cx+65,cy+65);
        rad:=angle*pi/180;
        xpos:=round(35*cos(rad));
        ypos:=round(35*sin(rad));
        d:=sqrt(len*len-xpos*xpos);
        py:=cy+ypos-round(d);
        moveto(cx+xpos,cy+ypos);
        lineto(cx,py);
        rectangle(cx-30,py-20,cx+30,py+30);
        ellipse(cx+xpos-12,cy+ypos-12,
                           cx+xpos+12,cy+ypos+12);
      end;
    end;
```

The line:

**len:=140;**

sets the length of the connecting rod to be 140 screen units.

We calculate the distance d using Pythagoras' formula:

**d:=sqrt(len*len-xpos*xpos);**

The next step is to draw the connecting rod; '**py**' is the vertical coordinate for the top end of the rod:

**py:=cy+ypos-round(d);**
**moveto(cx+xpos,cy+ypos);**
**lineto(cx,py);**

The piston is then drawn as a rectangle filling the width of the cylinder, and extending 20 units above and 30 units below the end of the connecting rod:

**rectangle(cx-30,py-20,cx+30,py+30);**

NOTE:
To avoid previous positions of the piston and connecting rod showing on the diagram, instructions have been included to blank out the previous drawing with a white rectangle before the new piston is drawn:

**brush.color:=clWhite;**
**pen.color:=clWhite;**
**rectangle(cx-30,150,cx+30,320);**

Compile and run the program to test the animation of the piston and connecting rod assembly, then return to the Delphi editing screen.

We now need to draw the inlet valve which opens to allow air into the cylinder, and the outlet valve which opens to let the exhaust gases escape. The valves can be drawn with a single procedure, provided we specify:

- which **side** of the engine - *left* or *right*
- which **position** for the valve - *up* or *down*.

This information can be included as parameters when the procedure is called.

Begin by adding a valve procedure to the list near the top of the program:

```
type
  TForm1 = class(TForm)
       ....
    procedure engine;
    procedure piston(angle:integer);
    procedure valve(side,position:string);
       ....
```

Go to the bottom of the program and add the procedure:

```
procedure TForm1.valve(side,position:string);
var
  cx,cy:integer;
begin
  if side='left' then
    cx:=303;
  if position='up' then
    cy:=78;
  with image1.canvas do
  begin
    brush.color:=clWhite;
    pen.color:=clBlack;
    moveto(cx-2,cy);
    lineto(cx-2,cy+60);
    lineto(cx-10,cy+72);
    lineto(cx+10,cy+72);
    lineto(cx+2,cy+60);
    lineto(cx+2,cy);
    lineto(cx-2,cy);
  end;
end;
```

When the procedure is called, the parameters '**side**' and '**position**' will be given as text strings, and will have the values '*left*' or '*right*', '*up*' or '*down*':

**procedure TForm1.valve(side,position:string);**

We then make use of '**side**' and '**position**' to set the coordinates **cx** and **cy.** These will determine where on the screen the valve is drawn:

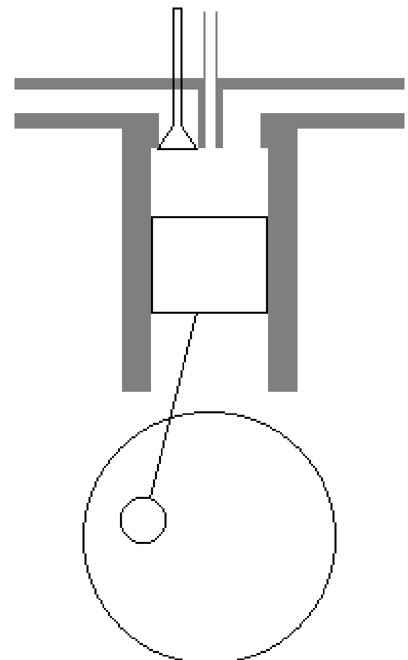
```
if side='left' then
   cx:=303;
if position='up' then
   cy:=78;
```

A series of graphics commands then draw the valve in the required position:

```
with image1.canvas do
begin
   brush.color:=clWhite;
   pen.color:=clBlack;
   moveto(cx-2,cy);
   lineto(cx-2,cy+60);
       .......
```

Go to the Form1 screen and double-click the *Timer* component to bring up the event handler. Add a line of program to call the **valve** procedure with the parameters '**left**' and '**up**':

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  engine;
  angle:=angle+15;
  if angle>720 then
    angle:=angle-720;
  piston(angle);
  valve('left','up');
end;
```



Compile and run the program. A valve should be shown in the '*left - up'* position.

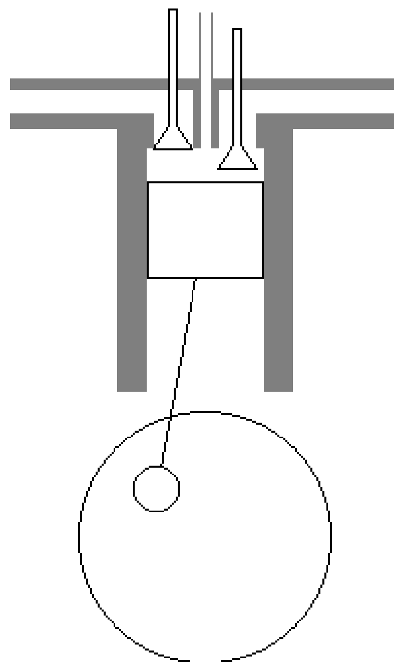
The shape of the valve is correct, but we need to make a gap in the top of the engine assembly for the valve stem to pass through. This can be done by drawing a white rectangle.

We also need to allow for the valve being on the '*right*' side, or in the '*down*' position. Add lines to the **valve** procedure to complete these tasks:

```
procedure TForm1.valve(side,position:string);
var
  cx,cy:integer;
begin
  if side='left' then
    cx:=303
  else
    cx:=336;
  if position='up' then
    cy:=78
  else
    cy:=88;
  with image1.canvas do
  begin
    brush.color:=clWhite;
    pen.color:=clWhite;
    rectangle(cx-2,cy-20,cx+3,cy+60);
    rectangle(cx-9,cy+50,cx+10,cy+72);
    pen.color:=clBlack;
    moveto(cx-2,cy);
    lineto(cx-2,cy+60);
    lineto(cx-10,cy+72);
           .....
```

NOTE:
There is never a semi-colon at the end of the line before an ***else*** command. Be careful to delete the semi-colons when you are altering this procedure.

Add another line to the **Timer** event handling procedure to draw a valve in the '*right - down*' position:

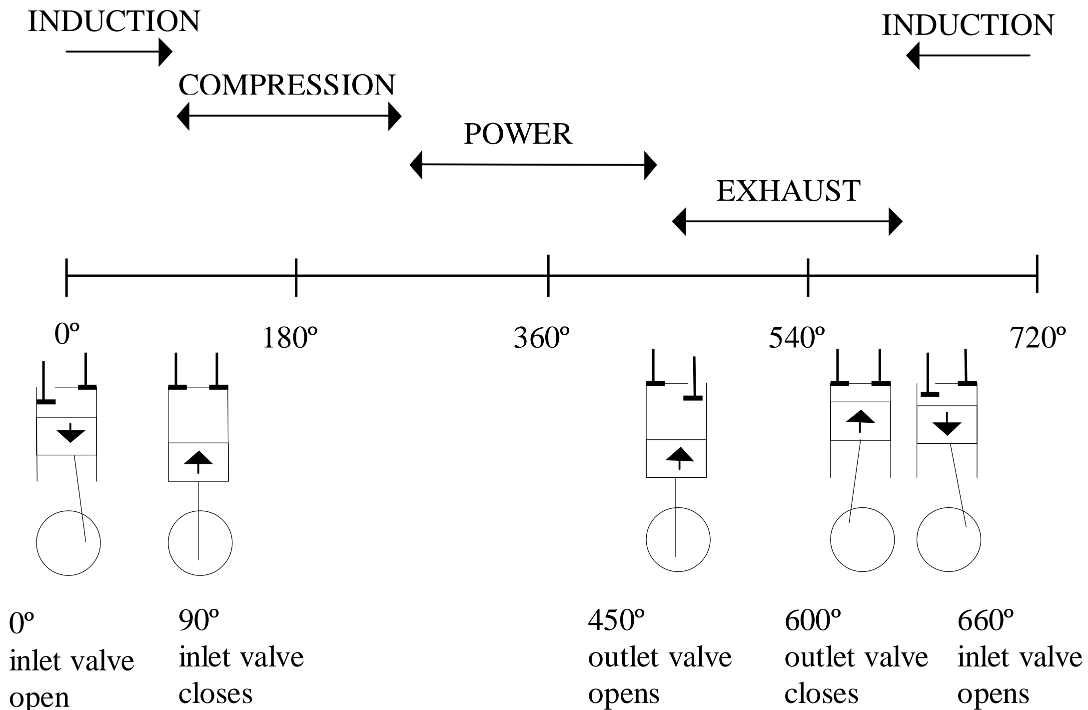


```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  engine;
  angle:=angle+15;
  if angle>720 then
    angle:=angle-720;
  piston(angle);
  valve('left','up');
  valve('right','down');
end;
```

Comile and run the program to check that both valves are now drawn correctly, then return to the Delphi editing screen.

It just remains to animate the valves so that they open and close at the correct times during the engine cycle.

The diagrams on page 308 show the four stages of the Diesel engine cycle: the ***induction, compression, power*** and ***exhaust*** strokes. These stages can be related to the rotation angle of the flywheel:




It is necessary for the **inlet** valve to be open during the ***induction*** stroke, and the **outlet** valve must be open during the ***exhaust*** stroke. The rotation angles at which the valve positions change are shown in the diagram above.

Go to the ***Timer*** event handling procedure and change the lines which display the valves:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  engine;
  angle:=angle+15;
  if angle>720 then
    angle:=angle-720;
  piston(angle);
  if (angle<90) or (angle>660) then
    valve('left','down')
  else
     valve('left','up');
  if (angle>450) and (angle<600) then
    valve('right','down')
  else
    valve('right','up');
end;
```

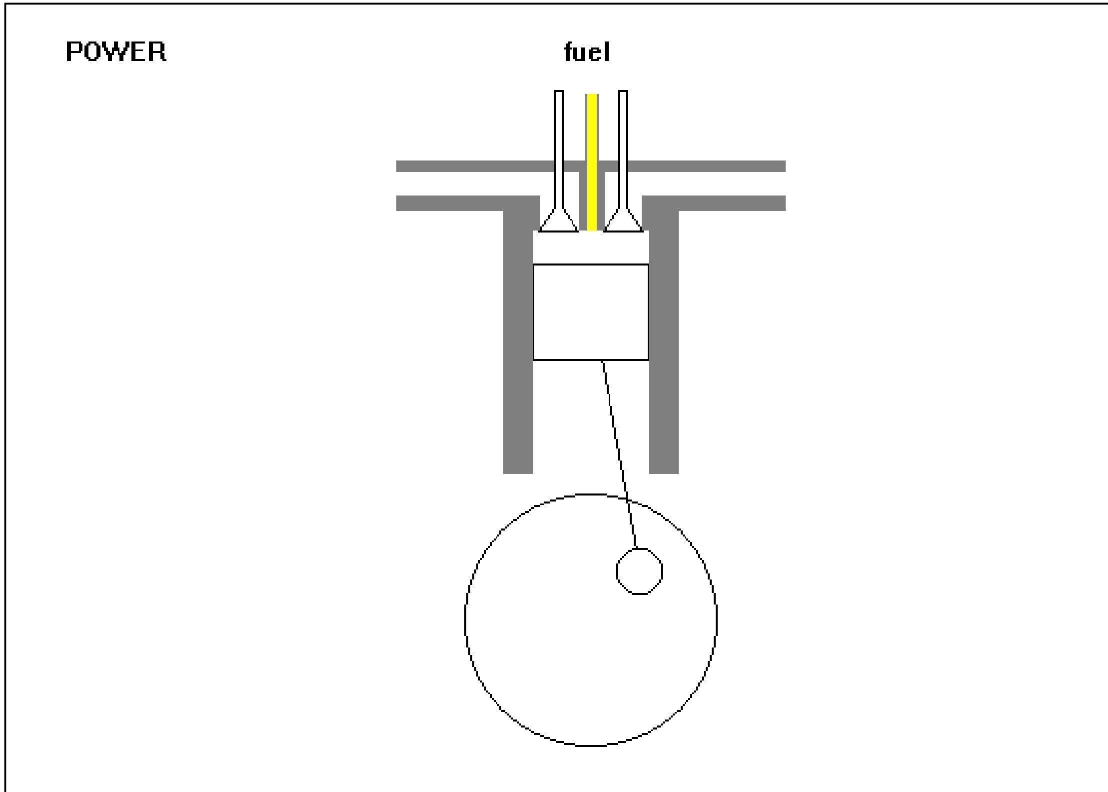
Compile and run the program. The valves should now open and close at the correct times during the engine cycle. Exit and return to the Delphi editing screen.

It would be helpful to display captions to show the stages of the engine cycle as the animation is running. Add lines to the ***Timer*** procedure to do this:

```
      ......
   if (angle>450) and (angle<600) then
      valve('right','down')
    else
      valve('right','up');
    with image1.canvas do
    begin
      pen.color:=clBlack;
      case angle of
        90:  textout(50,50,'COMPRESSION       ');
        270: textout(50,50,'POWER             ');
        450: textout(50,50,'EXHAUST           ');
        630: textout(50,50,'INDUCTION         ');
      end;
    end;
  end;
```

Compile and run the program. The caption should change as each stage of the engine cycle begins. Check this against the diagram on page 307, then return to the Delphi editing screen.

One further improvement we can make is to show the fuel being injected into the cylinder at the start of the power stroke:




Go back to the ***Timer*** procedure and add a section of program to do this:

```
       ........
   with image1.canvas do
    begin
      pen.color:=clBlack;
      case angle of
        90: textout(50,50,'COMPRESSION      ');
        270: textout(50,50,'POWER              ');
        450: textout(50,50,'EXHAUST      ');
        630: textout(50,50,'INDUCTION      ');
      end;
      if angle=270 then
      begin
        brush.color:=clYellow;
        pen.color:=clYellow;
        rectangle(cx-2,80,cx+3,150);
        pen.color:=clBlack;
        brush.color:=clWhite;
        textout(306,50,'fuel');
      end;
```

```
        if angle=360 then
        begin
          brush.color:=clWhite;
          pen.color:=clWhite;
          rectangle(306,50,360,64);
          rectangle(cx-2,80,cx+3,150);
        end;
      end;
    end;
```

A yellow rectangle fill is shown for the fuel when the angle reaches 270º, and this is blanked out with a white rectangle at 360º when the injection of fuel stops.

Compile and run the finished program.

SUMMARY

In this chapter you have:

- Used a *Scroll Box* to display a bitmap image larger than the screen
- Set the **Range** on the Scroll Bar to match the size of the image
- Found the position of the mouse on the screen by using the **Mouse Down** event handler of the *Image Box*
- Used the **canvas.pen.style** property to produce a dotted line
- Calculated a distance on the screen by means of Pythagoras' theorem
- Produced animation using a *Timer* component
- Displayed a **hint** for a component
- Made use of variables (e.g. **cx**, **cy**) to simplify the drawing of graphics