APPENDIX A

Three-dimensional graphics techniques

Three-dimensional graphics programs have many important applications, from computer aided engineering and architectural design to aircraft flight simulators and film animation. This chapter forms a brief introduction and starting point if you want to design your own three-dimensional modelling program....

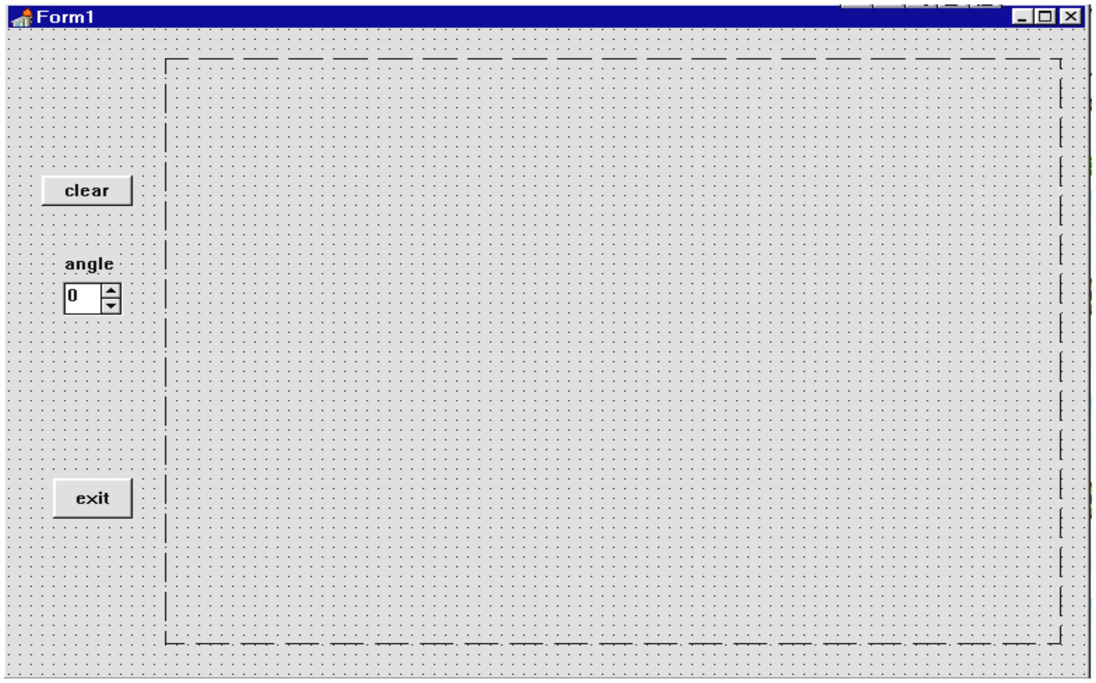
but be warned ...

the mathematical techniques are quite complex and require an understanding of ***trigonometry*** and ***matrices***. If you are not familiar with these topics, it would be useful to read appropriate sections of an A-level mathematics textbook.

Rotation in two dimensions

Before tackling anything as difficult as three-dimensional graphics, we will begin with a simple drawing program to input shapes and rotate them in two dimensions.

Set up a directory ROTATE and save a new Delphi project into it. Use the Object Inspector to **Maximize** the *Form*, and drag the grid to nearly fill the screen. Add an ***Image box***, two ***Buttons***, and a ***Spin Edit*** component:
Click on the ***Image box*** and press ENTER to bring up the Object Inspector.



Click on the Image Box and press ENTER to bring up the Object Inspector. Set the **Width** property to 640, and the **Height** to 480. Drag the Image Box into position on the screen.

Give the ***Buttons*** the captions '**clear**' and '**exit**'. Place a ***Label*** above the Spin Edit with the caption '**angle**'.

Double-click the '**exit**' button to produce an event handler and add the line:

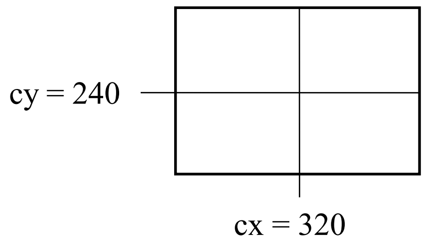
```
procedure TForm1.Button2Click(Sender: TObject);
begin
  halt;
end;
```

Now double-click the *Form* grid to produce an '**OnCreate**' procedure and add the line:

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

Compile and run the program. Check that a white background area is displayed. Click the '**exit**' button to return to the Delphi editing screen.

We are now going to set up a procedure to draw shapes on the Form using the mouse. First, however, it will be useful to define two constants ***cx*** and ***cy*** to give the coordinates of the centre of the image box:



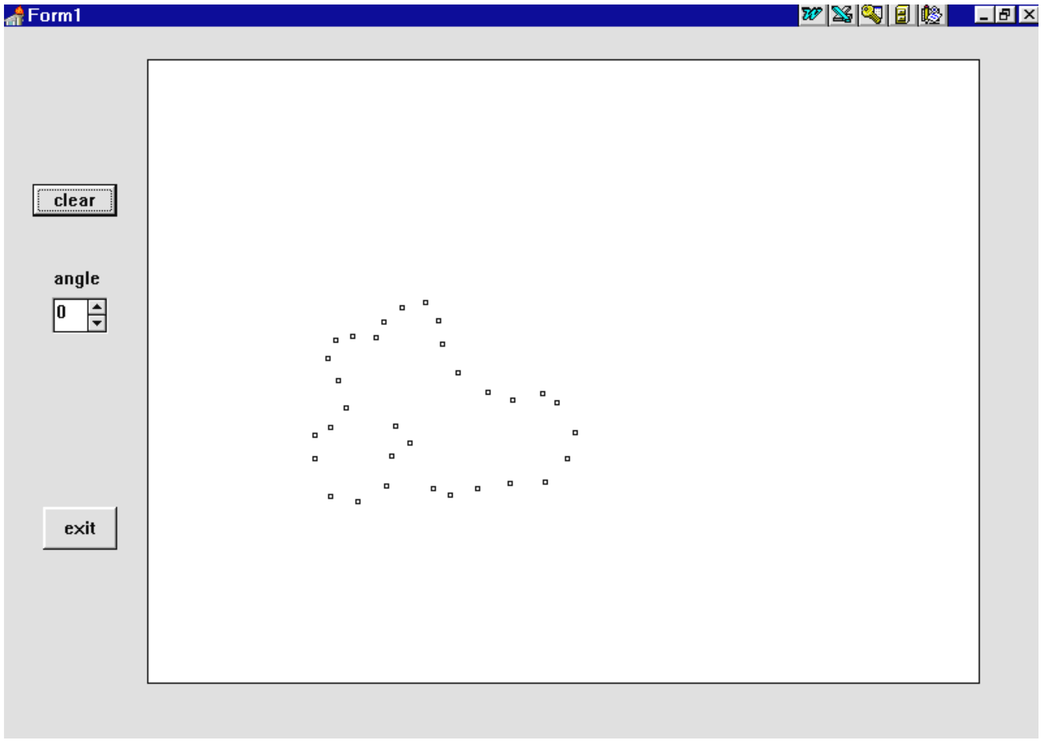
Insert the constants just above the '*implementation*' heading near the top of the program:

```
........
var
  Form1: TForm1;
const
  cx=320;
  cy=240;
implementation
........
```

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

```
procedure TForm1.Image1MouseDown(Sender:TObject;
  Button:TMouseButton;Shift:TShiftState;X,Y: Integer);
begin
  image1.canvas.rectangle(x-2,y-2,x+2,y+2);
end;
```

Compile and run the program. It should be possible to plot small squares on the image area by clicking the mouse:



Return to the Delphi editing screen. Later in the program we will need to know the positions where the mouse has been clicked on the image area. We will set up arrays to record the across coordinate (**xpos**) and the down coordinate (**ypos**) for each point. A variable will also be needed to keep a count of the number of points entered. Add these to the *Public declarations* section:

```
public
  { Public declarations }
  xpos,ypos:array[1..100] of integer;
  count:integer;
end;
```

The next step in developing the program is to draw a continuous line on the image area rather than a series of separate points. We begin by initialising the count to zero before drawing begins. Double-click on the *Form* grid to bring up the '**OnCreate**' procedure, then add a line:

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

An algorithm we can use to draw a shape is:

```
1.  increase the point count by 1
2.  IF this is the first point entered THEN
3.      put a dot on the screen
4.  ELSE
5.      draw a line from the previous point to the current point
6.  END IF
7.  record the coordinates of the current point in the arrays
```

Go to the '**MouseDown**' procedure for the *Image Box.* Delete the existing line of program and replace it with the following:

```
 procedure TForm1.Image1MouseDown(Sender:
          TObject; Button: TMouseButton;
          Shift: TShiftState;X,Y:Integer);
begin
  count:=count+1;
  if count=1 then
    image1.canvas.pixels[x,y]:=clBlack
  else
  begin
    with image1.canvas do
    begin
      moveto(xpos[count-1]+cx,ypos[count-1]+cy);
      lineto(x,y);
    end;
  end;
  xpos[count]:=x-cx;
  ypos[count]:=y-cy;
end;
```

This starts by increasing the number of points entered:

count:=count+1;

The count was initialised to zero, so count will become 1 when the first point is being entered.

When the first point is entered we just turn the pixel black at the position where the mouse was clicked - this will give a small black dot on the screen:

```
    if count=1 then
        image1.canvas.pixels[x,y]:=clBlack
```

The coordinates of the current point are recorded in the arrays:

```
        xpos[count]:=x-cx;
        ypos[count]:=y-cy;
```

We calculate the position relative to the **centre** of the Image Box by subtracting the constants ***cx*** and ***cy*** from the actual screen position.

As each further point is entered, a section of line will need to be drawn:

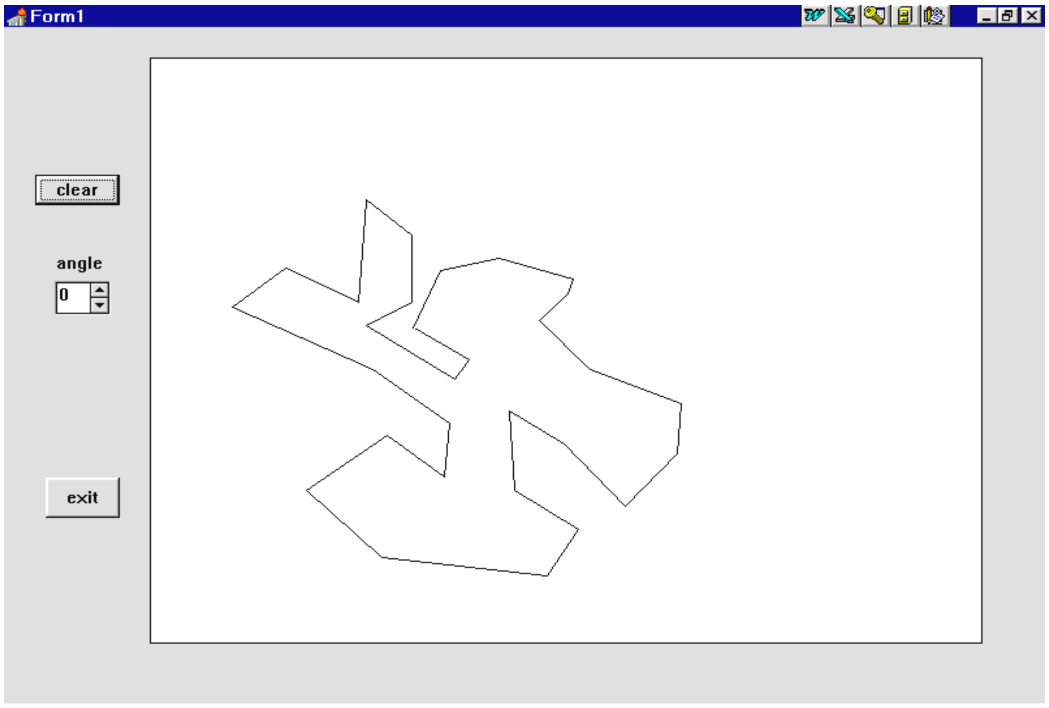
The coordinates of the previous point visited will be stored at array index **[count-1]**. We access the data and move to that point, remembering to add back ***cx*** and ***cy:***

```
        moveto(xpos[count-1] + cx, ypos[count-1] + cy);
```

A line is then drawn to the current mouse position:

```
        lineto(x,y);
```

Compile and run the program. This time when the mouse is clicked on the image area, a continuous line can be drawn:



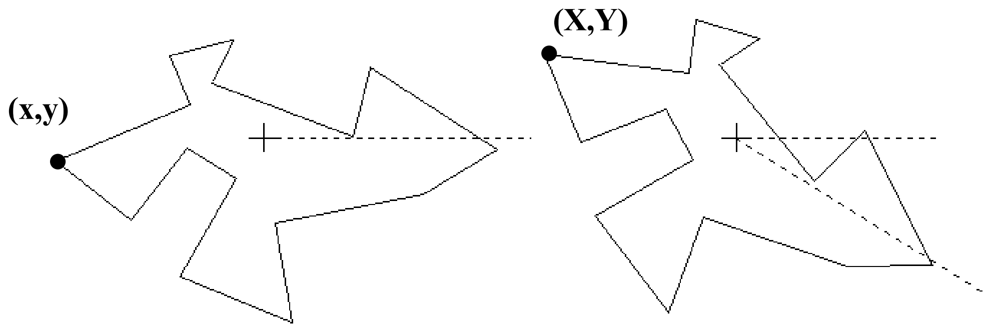
Test the drawing facility, then return to the Delphi editing screen.

It will be useful to be able to blank out the image area so that a new shape can be drawn. Double-click the '**clear**' button to produce an event handler, then add lines to clear the screen and re-initialise the point count to zero:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  image1.canvas.rectangle(0,0,640,480);
  count:=0;
end;
```

Compile and run the program. Enter a shape. Check that the screen can be cleared by clicking the button, then a new shape entered. Return to the Delphi editing screen.

We will now attempt to rotate a shape around the centre point of the *Image Box*. Suppose that a shape is rotated by an angle ❑:



As a result of the rotation, a typical point (**x,y**) on the shape moves to the position (**X,Y**). Using trigonometry, it is possible to calculate the new coordinates **X** and **Y**:

$$X = x.\cos\theta - y.\sin\theta$$

$$Y = x.\sin\theta + y.\cos\theta$$

Proofs of these equations can be found in mathematics textbooks.

Note: The two equations can also be written in matrix form:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The equations and the matrices are just different ways of saying the same thing. From the rules for multiplying matrices, it is always the case that:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{matrix} x.A + y.B \\ x.C + y.D \end{matrix}$$

Try substituting:

A = cos θ, B = - sin θ, C = sin θ, D = cos θ

and show that the matrices multiply out to give the correct equations for **X** and **Y.**

The matrix containing the trigonometric functions:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

is known as the ***two dimensional rotation matrix***. We will be making use of this later to develop more complex formulae for rotations in three dimensions.

We will now use the formulae to rotate the shape drawn in the *Image Box*. Begin by adding two new arrays ***rx*** and ***ry*** to the *Public delarations* section:

```
public
   { Public declarations }
   xpos,ypos,rx,ry:array[1..100] of integer;
   count:integer;
 end;
```

These arrays will be used to store the new coordinates we calculate for each of the points after the shape is rotated.

Bring the Form1 window to the front, then double-click the ***Spin Edit*** component to produce an event handler. Add the lines:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
var
  i:integer;
  angle:real;
begin
  angle:=spinedit1.value*pi/180;
  for i:=1 to count do
  begin
    rx[i]:=round(xpos[i]*cos(angle)-
                             ypos[i]*sin(angle));
    ry[i]:=round(xpos[i]*sin(angle)+
                             ypos[i]*cos(angle));
  end;
  with image1.canvas do
  begin
    rectangle(0,0,640,480);
    moveto(rx[1]+cx,ry[1]+cy);
    for i:=2 to count do
      lineto(rx[i]+cx,ry[i]+cy);
  end;
end;
```

We begin by converting the angle in the Spin Edit box from degrees to radians:

```
     angle:=spinedit1.value*pi/180;
```

A loop then takes each of the points in turn and uses the rotation formulae to calculate the new coordinates after the rotation angle is applied:

```
     for i:=1 to count do
     begin
      rx[i]:=round(xpos[i]*cos(angle) - ypos[i]*sin(angle));
      ry[i]:=round(xpos[i]*sin(angle) + ypos[i]*cos(angle));
```

The results are stored in the **rx** and **ry** arrays. The next section of the program uses these values to replot the shape. We move to the starting point:

```
      moveto(rx[1]+cx,ry[1]+cy);
```

then a loop draws lines for each of the remaining points around the shape:

```
      for i:=2 to count do
         lineto(rx[i]+cx,ry[i]+cy);
```

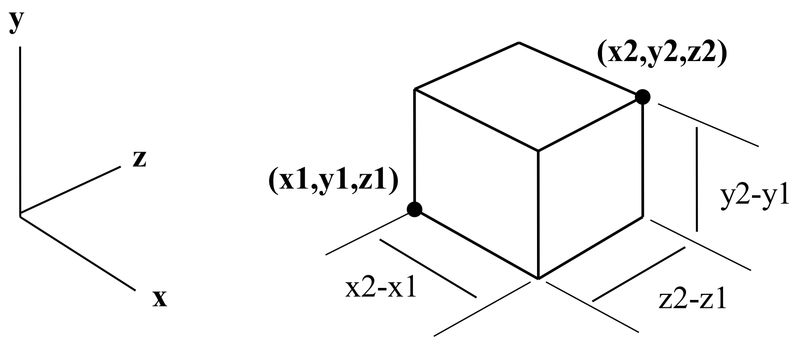
Compile and run the program. Use the mouse to draw a shape, then alter the angle shown by the Spin Edit. The shape should rotate around the centre point of the Image Box. Use the 'clear' button, then check that a different shape can be entered and rotated.

Rotating a wire-frame cube

We can now move on to three dimensional shapes. Our next objective is to set up a program to display a cube. There should be a procedure to rotate the cube into any position as it is being viewed on the screen. {Note: In this section, the term 'cube' is taken to include cuboid shapes where the width, length and height may be different.}

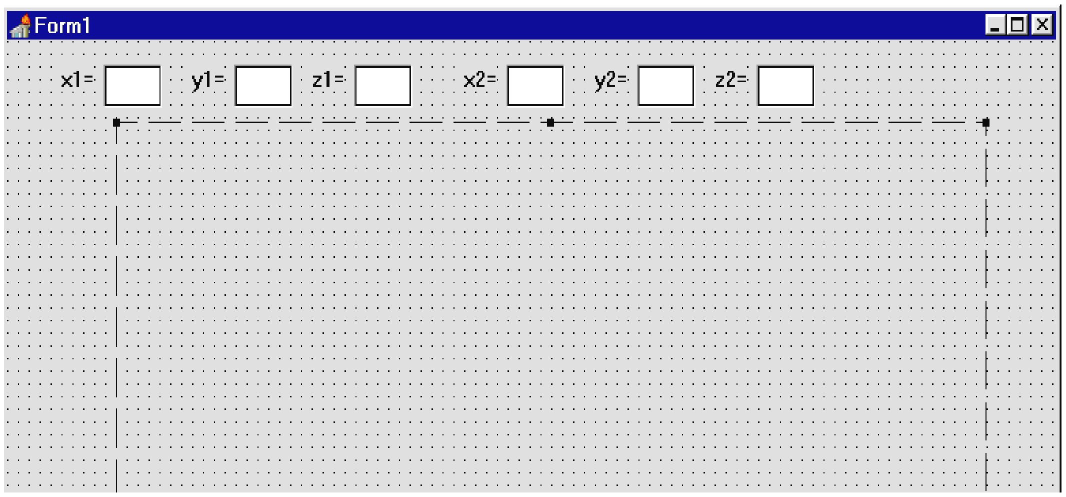
Set up a new directory CUBE and save a Delphi project into it. Use the Object inspector to maximize the form, and drag the grid to nearly fill the screen.

Three-dimensional graphics require a coordinate system with three axes - these are called ***x***, ***y*** and ***z:***



To enter the position of a cube which is oriented parallel to the three axes, we only need to give the x,y,z positions of opposite corners - the other corner positions can then be calculated. We will set up the *Form* to input this information:

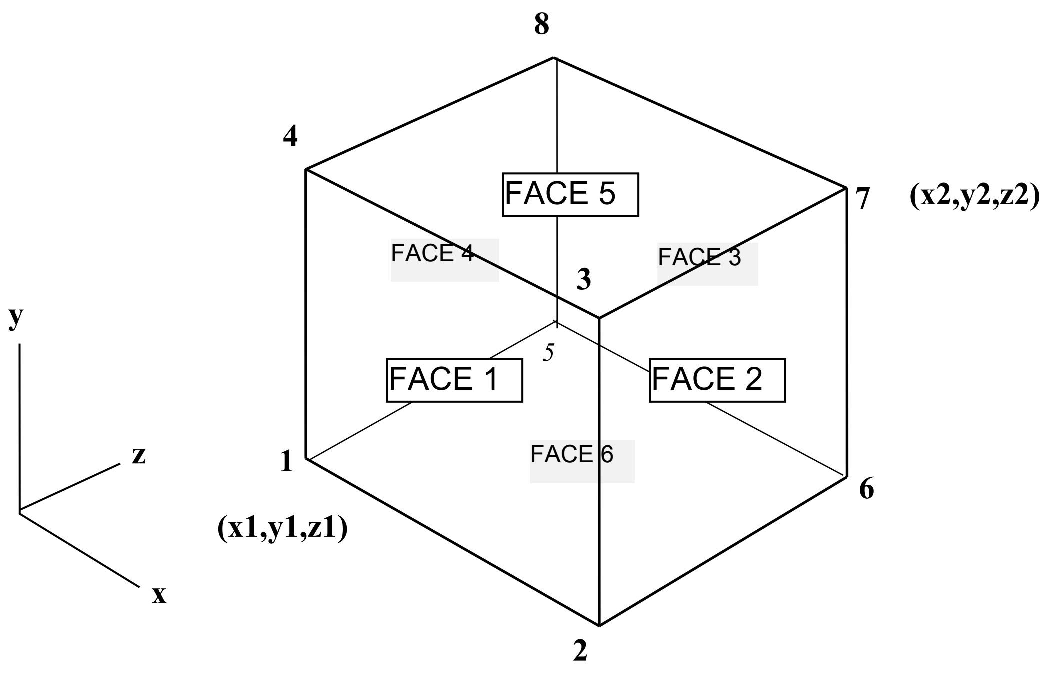
Add six ***Edit Boxes*** to the Form. Put ***Labels*** alongside with the captions: **'x1='**, **'y1='**, **'z1='**, **'x2='**, **'y2='**, and **'z2='**:



Place an ***Image Box*** on the grid, and use the Object Inspector to set its **Width** property to **640** and **Height** to **480**.

Double-click the Form1 grid to produce an 'OnCreate' event handler, then add a line of program to give a white background for the image area:

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



To proceed further with the solid modelling program we will need a numbering system for the **corners** and **faces** of the cube which is to be drawn by the computer.

Using the system above, coordinates will be entered for **corners 1** and **7**. Since the cube faces are initially parallel to the x, y and z axes, it is possible to list the coordinates of the remaining six corners:

corner 2	(x2, y1, z1)
corner 3	(x2, y2, z1)
corner 4	(x1, y2, z1)
corner 5	(x1, y1, z2)
corner 6	(x2, y1, z2)
corner 8	(x1, y2, z2)

We will set up an array to record the coordinates of each corner of the cube:

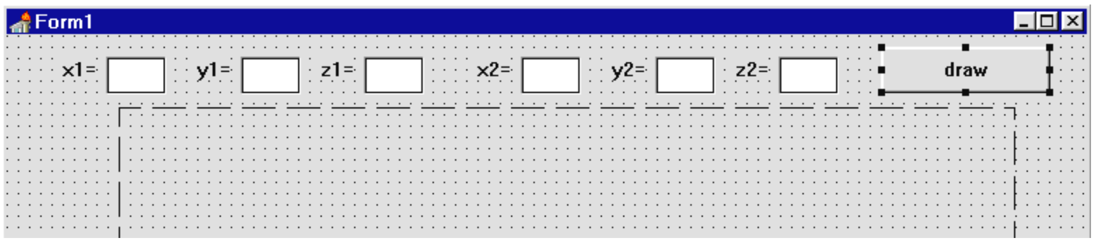
c[1..8, 1..3]

corner number

coordinate:
1 = x
2 = y
3 = z

Add this array to the *Public declarations* section:

```
public
   { Public declarations }
   c:array[1..8,1..3] of real;
```



Add a button to the form and give this the caption '**draw**'. Double-click the button to produce an event handler, then add the line:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  drawcube;
end;
```

We will set up a procedure '**drawcube**' to produce the screen display. First add this to the list of procedures at the top of the program:

```
       ......
  procedure FormCreate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure drawcube;
```

Go to the bottom of the program to insert the procedure:

```
procedure TForm1.drawcube;
var
  x1,y1,z1,x2,y2,z2:real;
begin
  x1:=strtofloat(edit1.text);
  y1:=strtofloat(edit2.text);
  z1:=strtofloat(edit3.text);
  x2:=strtofloat(edit4.text);
  y2:=strtofloat(edit5.text);
  z2:=strtofloat(edit6.text);
  c[1,1]:=x1; c[1,2]:=y1; c[1,3]:=z1;
  c[2,1]:=x2; c[2,2]:=y1; c[2,3]:=z1;
  c[3,1]:=x2; c[3,2]:=y2; c[3,3]:=z1;
  c[4,1]:=x1; c[4,2]:=y2; c[4,3]:=z1;
  c[5,1]:=x1; c[5,2]:=y1; c[5,3]:=z2;
  c[6,1]:=x2; c[6,2]:=y1; c[6,3]:=z2;
  c[7,1]:=x2; c[7,2]:=y2; c[7,3]:=z2;
  c[8,1]:=x1; c[8,2]:=y2; c[8,3]:=z2;
end;
```

This begins by transfering the values from the edit boxes to the x1 .. z2 variables. These are then used to set the coordinates for each of the corners of the cube.

We must now tell the computer the order in which to link the corners to draw the faces of the cube. Refering to the diagram on the previous page, **face 1** can be drawn by starting at corner 1, drawing lines to corners 2, 3, and 4, then back to corner 1:

face 1	**corners 1, 2, 3, 4**

The other corner sequences are:

face 2	**corners 2, 6, 7, 3**
face 3	**corners 6, 7, 8, 5**
face 4	**corners 1, 5, 8, 4**
face 5	**corners 4, 3, 7, 8**
face 6	**corners 1, 2, 6, 5**

An array can be used to record these sequences, with index values specifying the face and corner:

f [3, 2] = **7**

face 3

second corner in the sequence

Add this array to the *Public declarations* section:

```
public
  { Public declarations }
  c:array[1..8,1..3] of real;
  f:array[1..6,1..4] of integer;
```

Return to the end of the **drawcube** procedure. Add the lines to set up the array values:

```
   .......
  c[8,1]:=x1; c[8,2]:=y2; c[8,3]:=z2;
  f[1,1]:=1;  f[1,2]:=2;  f[1,3]:=3;  f[1,4]:=4;
  f[2,1]:=2;  f[2,2]:=6;  f[2,3]:=7;  f[2,4]:=3;
  f[3,1]:=6;  f[3,2]:=7;  f[3,3]:=8;  f[3,4]:=5;
  f[4,1]:=1;  f[4,2]:=5;  f[4,3]:=8;  f[4,4]:=4;
  f[5,1]:=4;  f[5,2]:=3;  f[5,3]:=7;  f[5,4]:=8;
  f[6,1]:=1;  f[6,2]:=2;  f[6,3]:=6;  f[6,4]:=5;
end;
```

As in the two-dimensional drawing program, it will be useful to have constants cx and cy to give the offset for the centre of the image box. Add these just above the '**implementation**' heading:

```
.......
const
  cx=320;
  cy=240;
implementation
```

Go to the bottom of the **drawcube** procedure and insert the lines to draw the cube faces:

```
    .......
    f[6,1]:=1;  f[6,2]:=2;  f[6,3]:=6;  f[6,4]:=5;
    scale:=20;
    with image1.canvas do
    begin
      brush.color:=clWhite;
      rectangle(0,0,640,480);
      for face:=1 to 6 do
      begin
        for i:=1 to 4 do
        begin
          corner:=f[face,i];
          x:=cx+round(c[corner,1]*scale);
          y:=cy+round(c[corner,2]*scale);
          if i=1 then
          begin
            startx:=x;
            starty:=y;
            moveto(x,y);
          end
          else
            lineto(x,y);
        end;
        lineto(startx,starty);
      end;
    end;
  end;
```

We begin by giving a scale factor:

scale:=20;

The coordinates for the corners will be multiplied by this scale factor so that the cube is drawn at a reasonable size on the screen.

A loop draws each of the faces in turn:

for face:=1 to 6 do . . . .

Inside this is another loop which draws each edge of the face:

for i:=1 to 4 do . . . .

The ***f*** array gives the reference number of each corner in the sequence required to draw the face:

corner:=f[face,i];

The ***x position*** of the corner is retrieved from the array of corner coordinates. This value multiplied by the scale factor, then we add the offset for the centre of the image box:

```
x:=cx+round(c[corner,1]*scale);
```

We now have the actual **x position** where the corner is to be plotted on the screen. The screen **y position** of the corner is calculated in a similar way:

```
y:=cy+round(c[corner,2]*scale);
```

If this is the first corner of the face, we just move to the **(x,y)** position, ready to start the line drawing. We also record the start position:

```
if i=1 then
begin
    startx:=x;
    starty:=y;
    moveto(x,y);
```

For the remaining edges, a line is drawn:

```
lineto(x,y);
```

The outline of the face is completed by drawing a final line back to the starting point:

```
lineto(startx,starty);
```

Add variables at the start of the **drawcube** procedure:

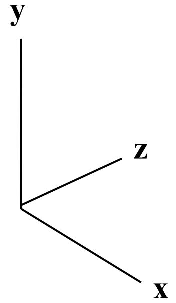
```
procedure TForm1.drawcube;
var
  x1,y1,z1,x2,y2,z2:real;
  face,corner,x,y,startx,starty,i:integer;
```

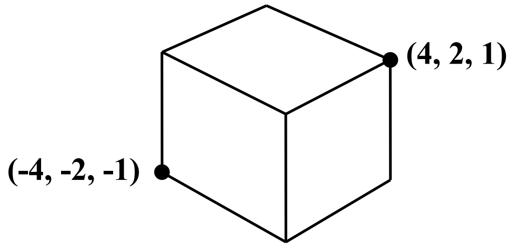
the add the variable '**scale**' to the *Public declarations* section:

```
{ Public declarations }
c:array[1..8,1..3] of real;
f:array[1..6,1..4] of integer;
scale:real;
```

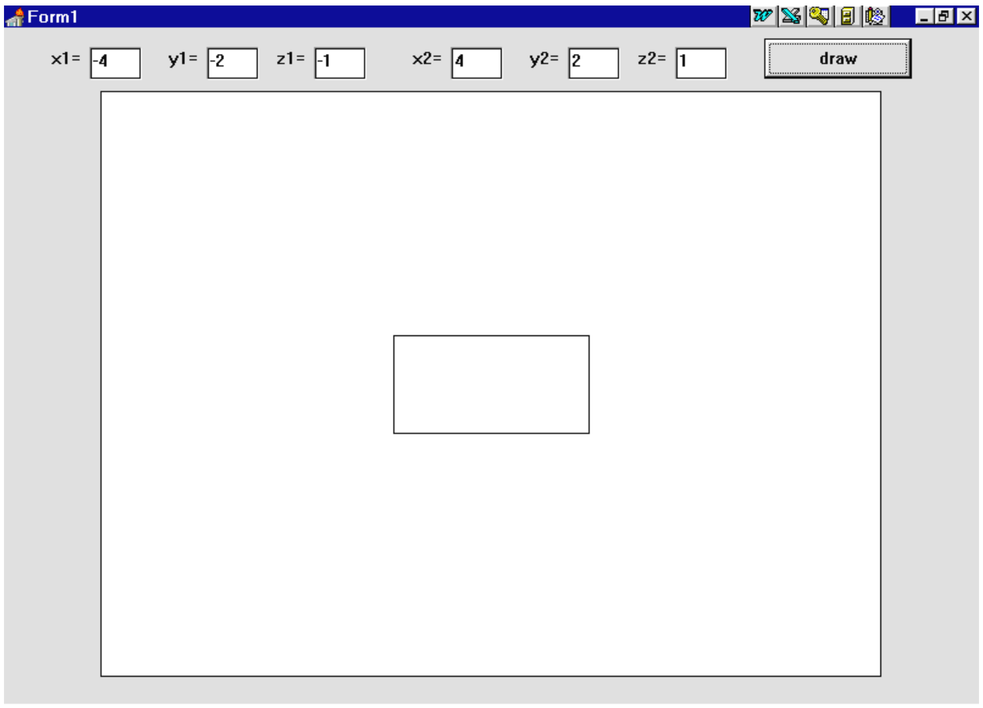
Compile and run the program. Enter test values in the edit boxes:

x1 = - 4	**y1 = -2**	**z1 = -1**
x2 = 4	**y2 = 2**	**z2 = 1**



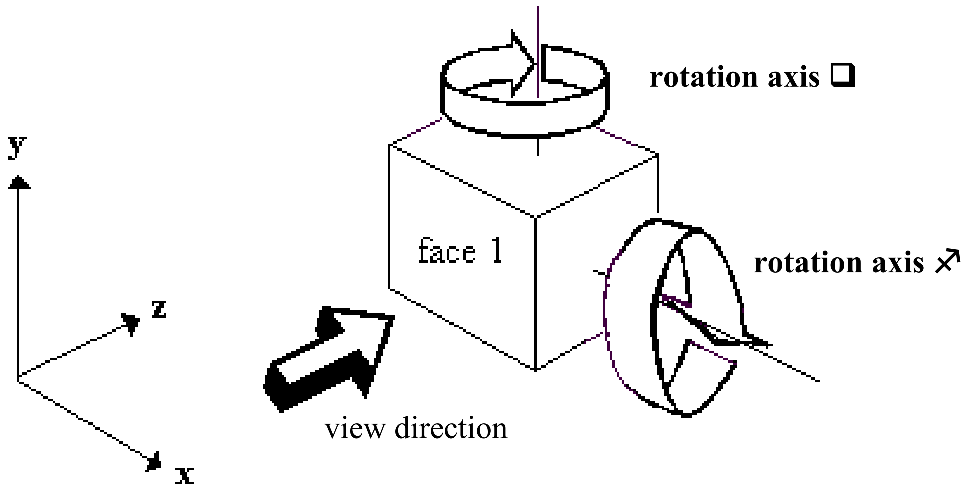


Click the '**draw**' button. A rectangle appears in the image box:


Form1

x1= -4   y1= -2   z1= -1   x2= 4   y2= 2   z2= 1

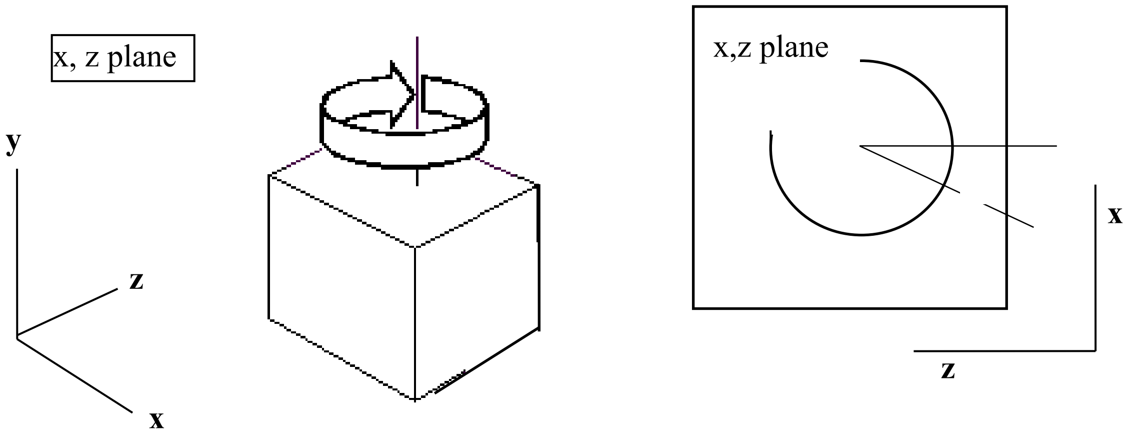
We are viewing the cube in the direction of the **z-axis** and only see **face 1**:



To view the other faces it will be necessary to apply a rotation to the cube. This is similar to the two dimensional rotation we carried out in the previous program. However, moving the cube into every possible position will require a combined rotation around <u>two</u> axes - we can call the two rotation angles ❑ {theta} and ➶ {phi}.

Before stopping the program, try various combinations of x, y, and z coordinates for the two opposite corners of the cube. Notice that because of the view direction, only the x- and y-coordinates affect the shape which is drawn.

We now need to carefully work out the formulae to rotate the cube. Consider first the rotation ❑. This takes place in the **x,z plane**:



Supposing that some point with the coordinates **(x, z)** is rotated by an **angle** ❑ so that it ends up in position **(X, Z)**. The new coordinates of the point are given by the formulae:

$$X = x.\cos\theta - z.\sin\theta$$

$$Z = x.\sin\theta + z.\cos\theta$$

These are the same formulae as for the two-dimensional rotation program - we have just used z instead of y. The two equations can also be written in matrix form:

$$\begin{bmatrix} X \\ Z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix}$$

However, because we are working in three-dimensions, we mustn't ignore the y coordinate. This will be unaffected if the point rotates in the **x,z plane**, so the new coordinate ***Y*** will be the same as the old coordinate ***y***. The full set of equations is therefore:

$$X = x.\cos\theta - z.\sin\theta$$

$$Y = y$$

$$Z = x.\sin\theta + z.\cos\theta$$

These three equations can also be written in matrix form:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

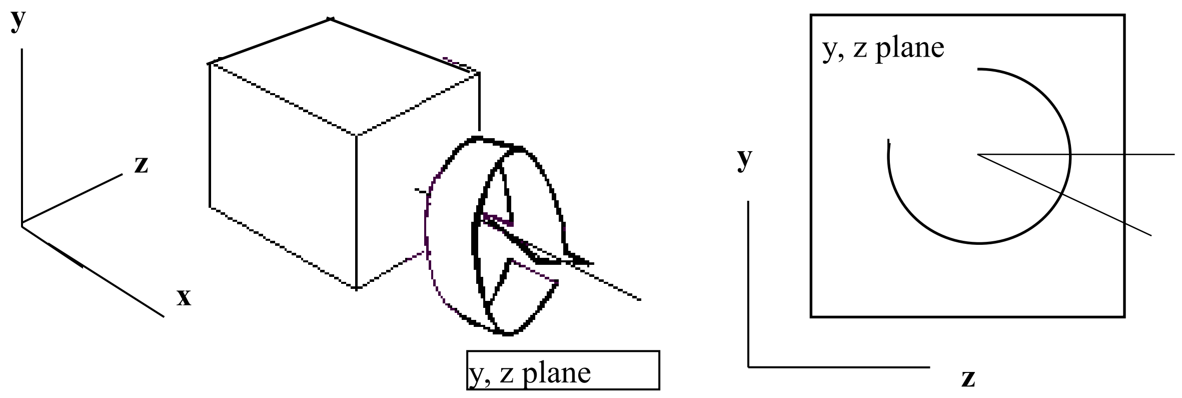
Note: You can check that the equations and the matrices are equivalent by applying the rule for multiplication:

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{matrix} x.A + y.B + z.C \\ x.D + y.E + z.F \\ x.G + y.H + z.I \end{matrix}$$

Substitute:

A = cos ❑ B = 0 C = -sin ❑ D = 0 E = 1
F = 0 G = sin ❑ H = 0 I = cos ❑

The other rotation ϕ takes place in the **y, z plane:**



For rotations in the **y, z plane**, the x coordinate will be unaffected. The set set of equations for a rotation are therefore:

$$X = x$$

$$Y = y.\cos\phi - z.\sin\phi$$

$$Z = y.\sin\phi + z.\cos\phi$$

These three equations can be written in matrix form:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

We now arrive at the general case where the cube is rotated by both an **angle** ❑ and an **angle ϕ** - together these allow us to turn the cube into any position we wish.

We can derive the three-dimensional rotation matrix by multiplying together the matrices for rotation in the **x, z** and **y, z** planes:

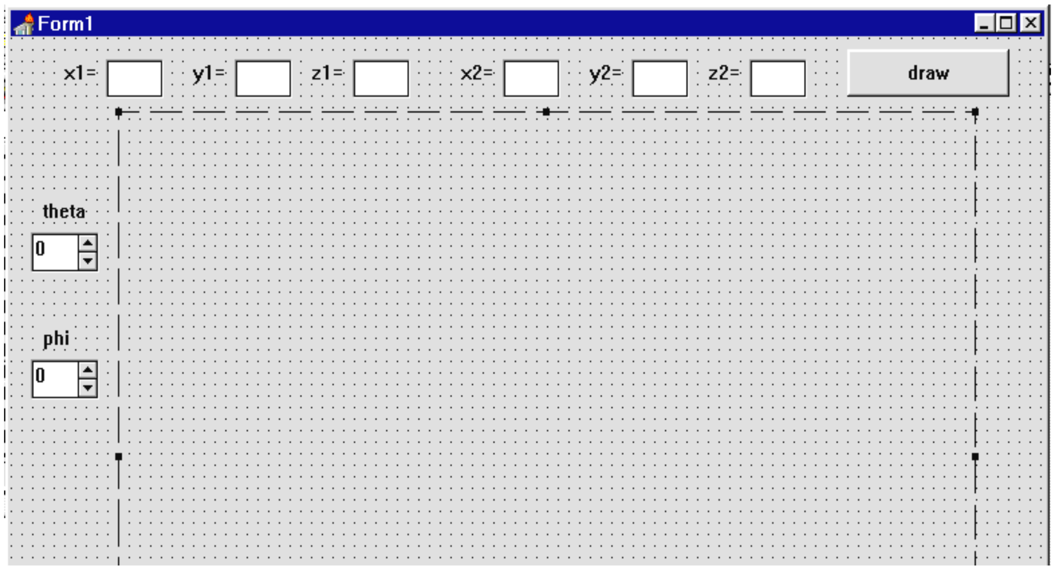
$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} =$$

$$\begin{bmatrix} \cos\theta & -\sin\theta.\sin\phi & -\sin\theta.\cos\phi \\ 0 & \cos\phi & -\sin\phi \\ \sin\theta & \cos\theta.\sin\phi & \cos\theta.\cos\phi \end{bmatrix}$$

This leads to the equations:

$$X = x.\cos\theta - y.\sin\theta.\sin\phi - z.\sin\theta.\cos\phi$$
$$Y = y.\cos\phi - z.\sin\phi$$
$$Z = x.\sin\theta + y.\cos\theta.\sin\phi + z.\cos\theta.\cos\phi$$

We can use these in the program to produce a fully rotating cube, but first we must have a way of changing the rotation angles **θ** and **ϕ**. Go to the *Form1* screen and add two ***Spin Edit*** components. Place ***Labels*** above them with the captions '**theta**' and '**phi**':



Find the position in the **drawcube** procedure where the corner sequences for each face have just been entered. Inset lines at this point to calculated the new coordinates of each corner after rotation by the angles **θ** and **ϕ.** Also alter the lines which set values for **x** and **y** so that they use the calculated values in the ***r*** array:

```
      ..............
f[6,1]:=1;  f[6,2]:=2;  f[6,3]:=6;  f[6,4]:=5;
for corner:=1 to 8 do
begin
  r[corner,1]:=c[corner,1]*cos(theta)
              -c[corner,2]*sin(theta)*sin(phi)
              -c[corner,3]*sin(theta)*cos(phi);
  r[corner,2]:=c[corner,2]*cos(phi)
              -c[corner,3]*sin(phi);
  r[corner,3]:=c[corner,1]*sin(theta)
              +c[corner,2]*cos(theta)*sin(phi)
              +c[corner,3]*cos(theta)*cos(phi);
end;
scale:=20;
with image1.canvas do
begin
  brush.color:=clWhite;
  rectangle(0,0,640,480);
  for face:=1 to 6 do
  begin
    for i:=1 to 4 do
    begin
      corner:=f[face,i];
      x:=cx+round(r[corner,1]*scale);
      y:=cy+round(r[corner,2]*scale);
             ...............
```

Add the variables ***theta***, ***phi***, and the ***r*** **array** to the Public declarations list:

```
public
   { Public declarations }
   c,r:array[1..8,1..3] of real;
   f:array[1..6,1..4] of integer;
   scale,theta,phi:real;
```

Go to the Form1 screen and double-click the '**theta**' ***Spin Edit*** to produce an event handler. Add the lines:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
begin
  theta:=spinedit1.value*pi/180;
  drawcube;
end;
```

This converts the value in the *Spin Edit* box from degrees to radians, then calls the procedure to redraw the cube in the new rotated position.

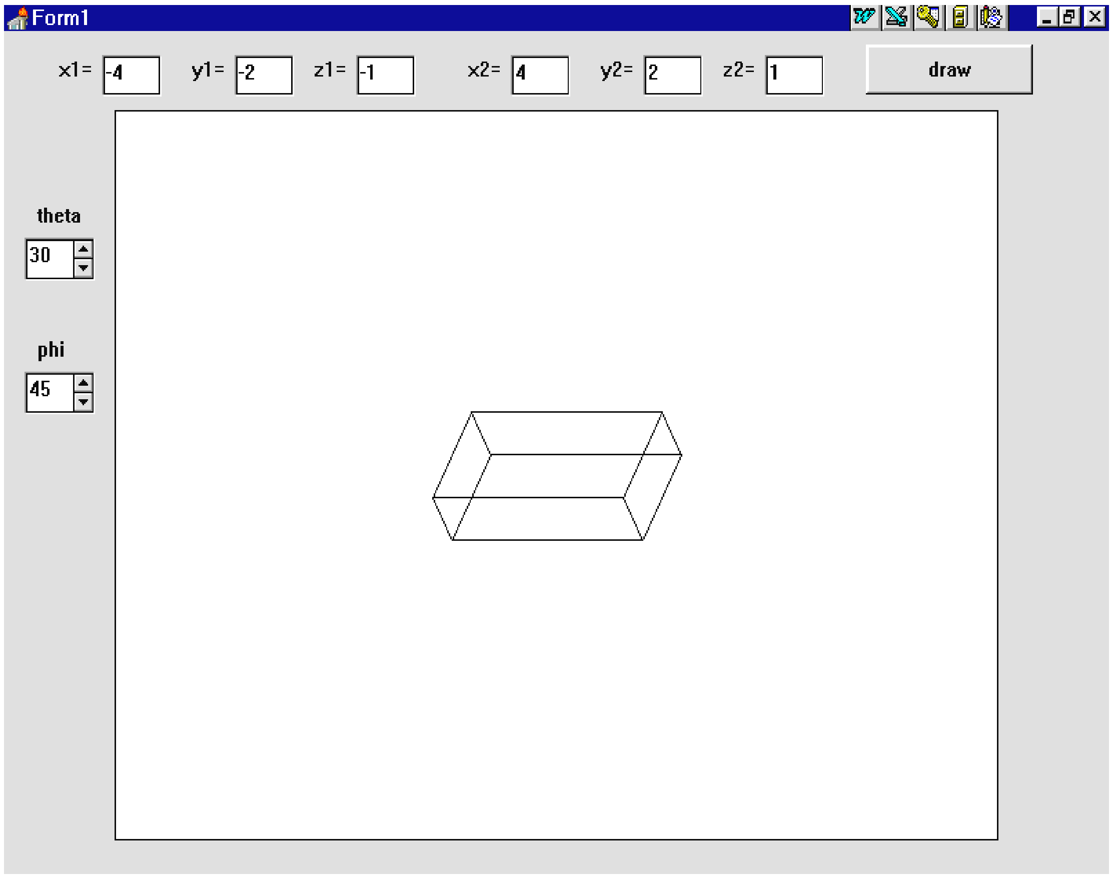
Produce a similar event handler for the '**phi**' ***Spin Edit***:

```
procedure TForm1.SpinEdit2Change(Sender: TObject);
begin
  phi:=spinedit2.value*pi/180;
  drawcube;
end;
```

Compile and run the program. Enter the test coordinates:

x1 = - 4	**y1 = -2**	**z1 = -1**
x2 = 4	**y2 = 2**	**z2 = 1**

The initial side view of the cube appears as a rectangle. It should now be possible to change the angles in the SpinEdit boxes and watch the cube rotate three-dimensionally:



At present the cube is being drawn in '***wire frame***' view, as if it were transparent. In the next section we will explore the techniques for displaying it as a solid opaque object...

Solid modelling using a depth sort algorithm

To produce a more realistic rotating cube, we need to show the faces in solid colour. Go to the **drawcube** procedure and replace the section of program which drew outlines for the faces. The procedure becomes:

```
   .....
  scale:=20;
  with image1.canvas do
  begin
    brush.color:=clWhite;
    rectangle(0,0,640,480);
    for face:=1 to 6 do
    begin
      for i:=1 to 4 do
      begin
        corner:=f[face,i];
        x[i]:=cx+round(r[corner,1]*scale);
        y[i]:=cy+round(r[corner,2]*scale);
      end;
      case face of
         1:brush.color:=clRed;
         2:brush.color:=clYellow;
         3:brush.color:=clLime;
         4:brush.color:=clTeal;
         5:brush.color:=clBlue;
         6:brush.color:=clWhite;
      end;
      Polygon
      ([Point(x[1],y[1]),Point(x[2],y[2]),
           Point(x[3],y[3]),Point(x[4],y[4])]);
    end;
  end;
end;
```

Instead of calculating the x- and y-coordinates of each corner while a face is being drawn, it is more convenient to calculate the coordinates beforehand and store them in arrays:

```
for i:=1 to 4 do
begin
    corner:=f[face,i];
    x[i]:=cx+round(r[corner,1]*scale);
    y[i]:=cy+round(r[corner,2]*scale);
end;
```

A **case** structure sets the colour for each of the six faces of the cube:

```
case face of
    1:brush.color:=clRed;
              . . . . .
    6:brush.color:=clWhite;
 end;
```

We then use the ***Polygon*** command to draw the face. The structure of this command is:

Polygon([first corner, second corner, third corner, fourth corner **]);**

The position of each corner is specified in the form:

Point (x-coordinate, y-coordinate **)**

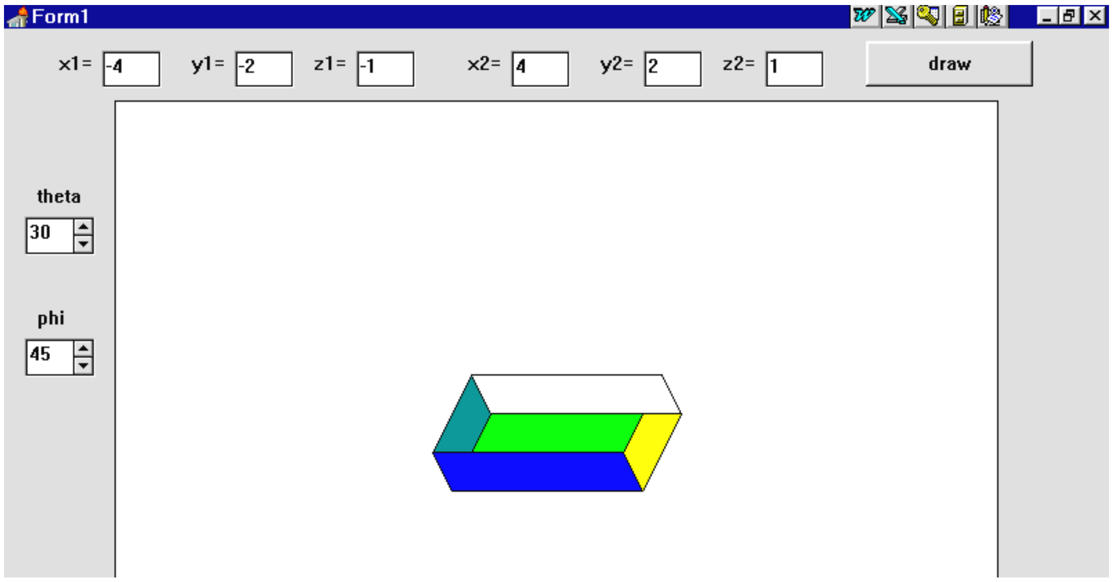
Before testing the program, go to the top of the drawcube procedure and change the x and y variables to arrays:

```
procedure TForm1.drawcube;
var
  x1,y1,z1,x2,y2,z2:real;
  face,corner,startx,starty,i:integer;
  x,y:array[1..4] of integer;
```

Compile and run the program. Enter the test coordinates:

x1 = - 4	**y1 = -2**	**z1 = -1**
x2 = 4	**y2 = 2**	**z2 = 1**

then press the '**draw**' button. Adjust the angles ❑ and ⤤ and watch what happens as the cube rotates:



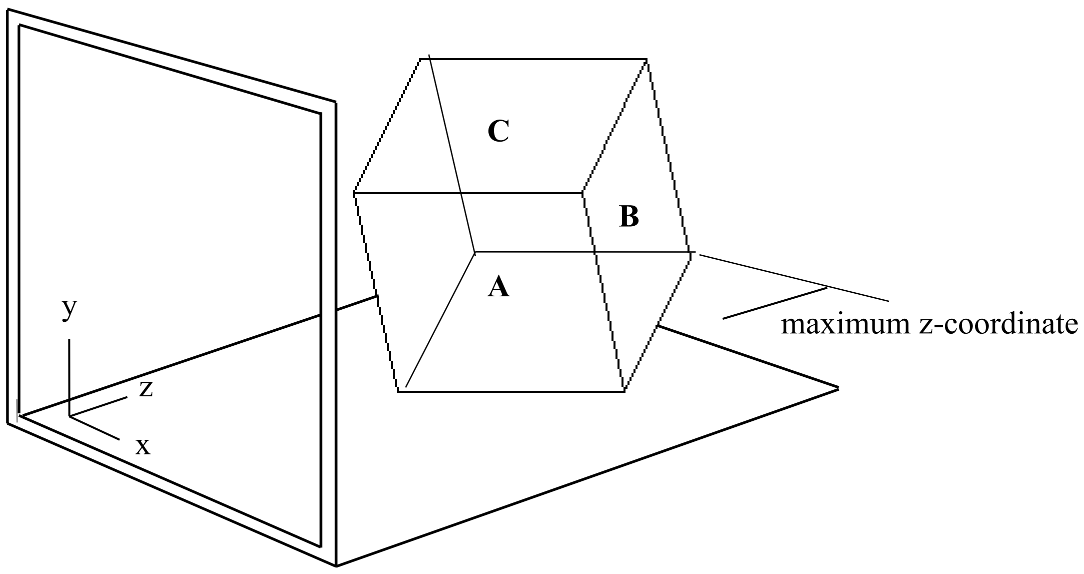
x1= -4 y1= -2 z1= -1 x2= 4 y2= 2 z2= 1

theta 30

phi 45

The faces are being plotted with the correct shape and colour, but they are not overlapping each other in the correct sequence - some faces which should be hidden at the back of the cube are being drawn on top. Return to the Delphi editing screen to tackle this problem.

We need to carry out a '**depth sort**' of the faces. The faces furthest from the viewer need to be drawn first, then the nearer faces plotted on top to build up the correct picture of the cube:



The order in which faces should be plotted depends on the z-coordinates. In this example, **face B** should be plotted first as it is furthest from the viewer and contains the **maximum z-coordinate** of any corner of the cube. **Faces C and A** have **lower z-coordinate values** for the corners, so will be plotted on top. Face B will actually be hidden from view in the finished picture of the cube.

A strategy for drawing the cube, known as a '***depth sort algorithm***' can be written:

```
1. LOOP for face = 1 to 6
2.     find the maximum z-coordinate of any of the corners of the face
3.     record the number of the face and the maximum z-coordinate
4. END LOOP
5. carry out a bubble sort on the parallel arrays containing the face
   numbers and their maximum z values
6. plot the faces in order, starting with the face which contains the
   maximum z-coordinate value
```

Go to the **drawcube** procedure and add the program lines to carry out the depth sort:

```
  .......
 scale:=20;
 with image1.canvas do
 begin
  brush.color:=clWhite;
  rectangle(0,0,640,480);
  for face:=1 to 6 do
  begin
   sequence[face]:=face;
   corner:=f[face,1];
   max[face]:=r[corner,3];
   for i:=2 to 4 do
   begin
    corner:=f[face,i];
    if r[corner,3] > max[face] then
      max[face]:=r[corner,3];
   end;
  end;
  for i:=1 to 5 do
   for j:=i+1 to 6 do
   begin
    if max[j]>max[i] then
    begin
     tempmax:=max[i];
     tempsequence:=sequence[i];
     max[i]:=max[j];
     sequence[i]:=sequence[j];
     max[j]:=tempmax;
     sequence[j]:=tempsequence;
    end;
   end;
  for j:=1 to 6 do
  begin
   face:=sequence[j];
   for i:=1 to 4 do
   begin
    corner:=f[face,i];
    x[i]:=cx+round(r[corner,1]*scale);
    y[i]:=cy+round(r[corner,2]*scale);
          ........
```

The algorithm begins with a loop to check each face in turn:

for face:=1 to 6 do. . . .

The number of the face is recorded in an array called **'sequence':**

sequence[face]:=face;

We then find the maximum z-coordinate for any of the four corners of the face:

```
        corner:=f[face,1];
        max[face]:=r[corner,3];
        for i:=2 to 4 do
        begin
           corner:=f[face,i];
           if r[corner,3] > max[face] then
               max[face]:=r[corner,3];
        end;
      end;
```

This uses the technique you learned in **Chapter 8.** We begin by assuming that corner 1 has the maximum z-value and store this in the '**max**' array. The other corners are then checked in turn, and '**max**' is updated each time a larger value is found.

We now have two parallel arrays containing the maximum z-values for each of the faces, e.g.:

sequence	**max**
1	**2.75**
2	**3.82**
3	**1.49**

etc...

It is now just necessary to sort the arrays so that the faces are arranged in descending order of z-value, i.e.:

sequence	**max**
2	**3.82**
1	**2.75**
3	**1.49**

etc...

This is done by a bubble sort:

```
for i:=1 to 5 do
    for j:=i+1 to 6 do
   begin
       if max[j]>max[i] then
       begin
            tempmax:=max[i];
            tempsequence:=sequence[i];
            max[i]:=max[j];
            sequence[i]:=sequence[j];
            max[j]:=tempmax;
             sequence[j]:=tempsequence;
```

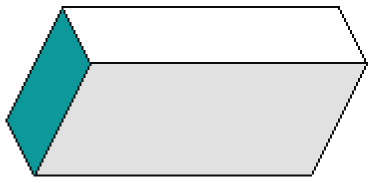
The loop to draw the faces can then begin. The number of the face to be plotted is obtained from the **sequence** array at the start of each pass through the loop:

```
for j:=1 to 6 do
begin
   face:=sequence[j];
   {continue with drawing the face}
```

Go to the top of the drawcube procedure and add the variables **tempmax, j, tempsequence**, and the **max** and **sequence** arrays:

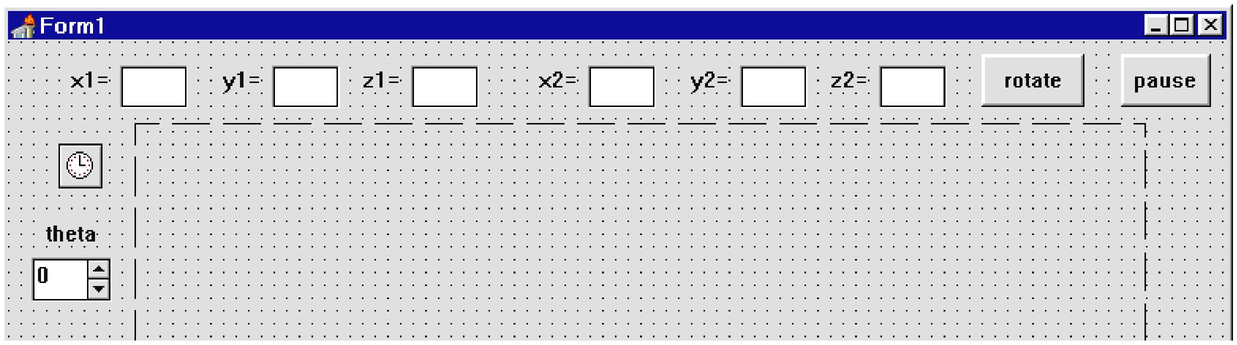
```
procedure TForm1.drawcube;
var
  x1,y1,z1,x2,y2,z2,tempmax:real;
  face,corner,startx,starty,
                   i,j,tempsequence:integer;
  x,y:array[1..4] of integer;
  max:array[1..6] of real;
  sequence:array[1..6] of integer;
```

Compile and run the program. Enter the test coordinates, the try altering the θ and ϕ angles. The cube should be correctly displayed now:



Return to the Delphi editing screen and bring the ***Form1*** window to the front. To finish the program, it would be nice to have the cube rotating by itself on screen. We will do this now.

Change the caption on the 'draw' button so that it reads '**rotate**'. Add another ***Button*** with the caption 'pause', and place a ***Timer*** component on the grid:



Double-click the '**rotate**' button to bring up the event handler. Change the procedure to:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  timer1.enabled:=true;
end;
```

Produce an event handler for the '**pause**' button and add the line:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  timer1.enabled:=false;
end;
```

Click the ***Timer*** icon and press ENTER to bring up the Object Inspector. Set the **Interval** property to **150,** and **Enabled** to **False**. Return to the Form1 grid and double-click the ***Timer*** icon to produce an event handler. Add the lines:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  spinedit1.value:=spinedit1.value+6+random(6);
  if spinedit1.value>360 then
    spinedit1.value:=spinedit1.value-360;
  spinedit2.value:=spinedit2.value+3+random(3);
  if spinedit2.value>360 then
    spinedit2.value:=spinedit2.value-360;
  theta:=spinedit1.value*pi/180;
  phi:=spinedit2.value*pi/180;
  drawcube;
end;
```

The purpose of this procedure is to increase the θ and ϕ angles at the end of each time interval, then redraw the cube so that it appears to be rotating. A random value has been introduced so that the motion is more interesting and unpredictable.

Compile and run the completed program. Enter coordinates for a cube and test the automatic rotation. It is possible to stop the motion with the '**pause**' button, then manually adjust the cube by means of the Spin Edit boxes.