The CMM2 3D engine

The new command is DRAW3D and has a number of sub commands.
DRAW3D CAMERA
DRAW3D CLOSE
DRAW3D CLOSE ALL
DRAW3D CREATE
DRAW3D HIDE
DRAW3D HIDE ALL
DRAW3D RESET
DRAW3D ROTATE
DRAW3D SHOW
DRAW3D WRITE

Before looking at the commands in detail I will try and explain the concept and the limitations of the engine.
The 3D world is an area of space 65532 x 65532 x 32766 units (x, y, z) centred at 0,0,0
In other words an object can be placed from -32766 to 32766 in the x-axis, -32766 to 32766 in the y-axis and 0 to 32766 in the z-axis
You can define up to 128 3D objects numbered 1 to 128.
Each object is described in terms of how many vertices it has, how many faces it has, which vertices make up each face and the colours of the edges and infill of each face. Objects can have any number of vertices and faces limited only by system memory.

The vertices are specified as x,y,z coordinates referenced to the object centre at 0,0,0

In addition for each object you will define the "camera" that is used to view the object. The 3D engine supports up to 6 camera positions numbered 1 to 6

All cameras look along their Z axis and before you display a 3D object the associated camera must be initialised by defining the x,y position of the camera and its viewplane. In camera terms the viewplane can be thought of as the focal length of the camera lens. So the bigger the value of the viewplane the more the camera "magnifies" the image.
For example, if we position a 3D object 1000 units away from the camera and the viewplane is at 200 then the projected image of the object onto the viewplane will be 20% of its original size. If we "zoom" the viewplane to a "focal length" of 800 the projected image will now be 80% of its original size.

When a 3D object is created the data used to create it is stored in CMM2 memory and any MMBasic arrays used to create the object can be "erased" if required.

All objects are stored in their initial orientation as defined by their initialising data but they can be rotated in three dimensions using the DRAW3D ROTATE command. This command acts upon the initial orientation and stores a rotated copy transparently in the object data internally in the firmware. Rotation takes place around the objects own centre. If you wish to rotate around any position in the 3D world this should be done as first a rotation of the object and then a move of the object. It is important to understand that every rotation requested for an object starts from the initial orientation and is not cumulative. (However, this can be overridden - see the DRAW3D RESET command)

Rotation is specified using a quaternion but don't worry I've included a very simple MATH command to convert yaw, pitch and roll angles into the required quaternion (MATH Q_EULER)

Rotation has no effect on a displayed object but merely updates the internal memory definition of the object.

There are two commands used to place an object in the 3D world - DRAW3D WRITE and DRAW3D SHOW. The only difference is that, assuming the object was already displayed, the SHOW command will clear a rectangle on the current write page sufficient to remove the existing object image before displaying it whereas DRAW3D WRITE just overwrites whatever in on the write page with the 2D representation of the object.

It is entirely up to the MMBasic programmer to deal with things like overlap of objects in the 3D world and on the screen but to aid this objects that have been SHOWn can be removed and the rectangular area of the screen in which they were drawn cleared using the DRAW3D HIDE command.

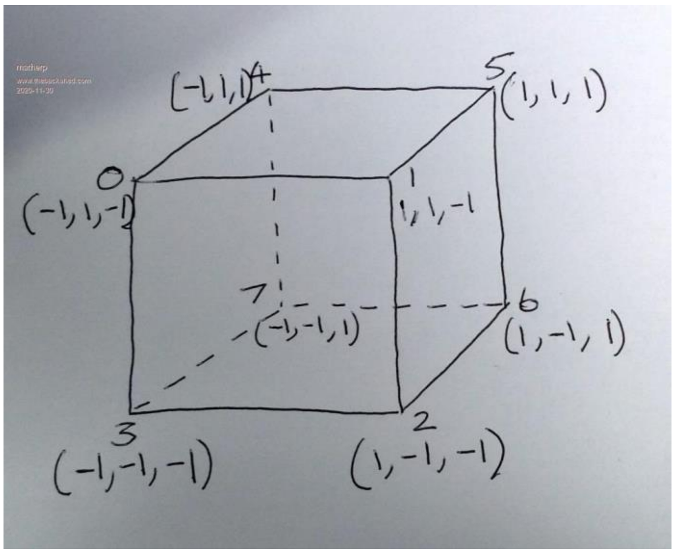
All objects and camera positions are deleted on any mode change and every time a program is run.

Hopefully the above gives you a basic understanding of how the 3D engine works and its limitations. The way the camera works may seem to create a specific limitation in terms of multiple views of an object but we will see in a subsequent post how this can be overcome.

DRAW3D commands

DRAW3D CREATE

DRAW3D CREATE n, nv, nf, camera, vertices(), fc(), faces(), colours() , edge() [,fill()]
DRAW3D CREATE is the command that creates a 3D object and all the information needed for the object is included in the parameter list. We will use a cube as an example.



n is the object number (1-128)

nv: number of vertices (e.g. 8 for a cube)

nf: number of faces (e.g. 6 for a cube)

camera: number of the camera to use when displaying the object (1 to 6)

vertices(): This is a 3 by nv array that holds the x,y,z coordinates of the 3D object. For example the vertex definition for our cube with side length 2 with Option Base 0 centred on 0,0,0 could be:
DIM FLOAT vertices(2,7) = (-1,1,-1, 1,1,-1, 1,-1,-1, -1,-1,-1, -1,1,1, 1,1,1, 1,-1,1, -1,-1,1)
Note that the negative values represent the vertices closest to the camera.

facecount(): is a count of how many vertices are needed to define each face, so in our example for the cube which has 6 faces it would be:
DIM INTEGER facecount(5)=(4,4,4,4,4,4)

faces(): This is a very important array and defines the vertices that make up each face of the 3D object. There is one critical thing in setting up this array. The vertices must be listed in a clockwise order for each face as though you were looking at that face from in front of it. It doesn't matter which order the faces are listed as long as they match the correct vertex count in fc() and it doesn't matter which vertex you start on for each face. In our example this array could be:
DIM INTEGER faces(23)=(0,1,2,3, 1,5,6,2, 0,4,5,1, 5,4,7,6, 2,6,7,3, 0,3,7,4)

colours(): This is an array that holds a simple list of all the colours you want to use to colour the 3D object. So if we want a different colour for each face and another one for all the edges we could set this array as follow:
DIM INTEGER colours(6)=(rgb(blue), rgb(green), rgb(yellow), rgb(cyan), rgb(red), rgb(magenta), rgb(yellow))

edge(): This arrays specifies which of our colours to use for each edge of the 3D object. We will set them all to the array index in colours() holding the value yellow
DIM INTEGER edge(5)=(6,6,6,6,6,6)

fill(): This array specifies which colour to use for each face of the 3D object. We will set them each to a different colour by specifying the array index into colours()
DIM INTEGER fill(5)=(0,1,2,3,4,5)

If the optional parameter fill() is omitted then the 3D object will be drawn as a wireframe

Those familiar with 3D graphics will notice that the parameters to DRAW3D CREATE match the way 3D objects are defined in many 3D description files like wrl or ply files.

DRAW3D ROTATE

DRAW3D ROTATE q(), n [,n1 [,n2...]}
This rotates one or more 3D objects about their centres.
q() is a matrix (quaternion) that defines the required rotation. We use quaternions because they don't suffer from gimbal lock and are computationally fairly efficient but the math is completely hidden by the firmware.

The n values are the 3D object IDs assigned when the object(s) was(were) created.

From the perspective of the MMBasic user a quaternion is simply a 5 element floating point array and it is loaded using one of two methods

MATH Q_EULER yaw, pitch, roll, q()
MATH Q_CREATE theta, x, y, z, q()

MATH Q_CREATE defines a rotation around the vector x,y,z by theta degrees (defaults to radians but supports OPTION ANGLE). If x is zero and y is zero then the rotation is around the z-axis which is equivalent to rolling the object. If only x is non-zero then the rotation will pitch the object and y non-zero will yaw the object.

MATH Q_EULER sets q() to perform a rotation as defined by the yaw, pitch and roll angles
With the camera facing the object yaw is looking from the top of the object and rotates clockwise, pitch rotates the top away from the camera and roll rotates around the z-axis clockwise.
The yaw, pitch and roll angles default to radians but respect the setting of OPTION ANGLE

All objects specified in the ROTATE command are rotated by the same amount. Nothing happens on the screen but internally the firmware stores the rotated coordinates as well as the original ones.

It is very important to note that the rotate command acts on the original object as defined in the CREATE command. Rotate commands are not cumulative. This ensures that rounding errors cannot affect the accuracy.

However, there is a command that can override this:

DRAW RESET

DRAW3D RESET n [,n1 [,n2...]}
This command takes the current rotated version of the object(s) and copies it into the initialisation data. Whilst this isn't recommended for iterative rotations it is very useful in establishing multiple views of the same object. Later in this document I will show how to use DRAW3D RESET to create and simultaneously manipulate front and plan views of identical objects

DRAW3D CAMERA

DRAW3D CAMERA n, z_viewplane[,x_camera [,y_camera] [,PAN_X] [,PAN_Y]

The camera number “n” and the “viewplane” z distance are mandatory, all other parameters are optional and all default to 0
The camera can be placed anywhere in the plane x, y, 0 but always looks out along the z axis.
The viewplane is perpendicular to the Z axis and is a plane sized 65532 x 65532 in the x and y axis stretching, like the world from -32766 to 32766 in the x-axis and -32766 to 32766 in the y-axis
However, our VGA display can only show a very small part of the viewplane as limited by the screen dimensions (MM.HRES x MM.VRES). We could call this the "viewport".
By default the viewport will be set to +/- MM.HRES\2 either side of the camera x position and +/- MM.VRES either side of the camera y position.
This means If I place a 3D object at 0,0,Z in the 3D world and set my camera at 0,0,0 in the 3D world then the object will project into the centre of the screen.
Likewise, if I place a 3D object at 400,400,Z in the 3D world and set my camera at 400,400,0 in the 3D world then the object will also project into the centre of the screen.
However, there are occasions when this may not be what we want so there are two extra parameters to the CAMERA command - PAN-X and PAN-Y.
These move the position of the viewport on the viewplane relative to the camera position.
A practical example makes this clearer:
Suppose we position a number of objects in the 3D world with their lower extremities at x, 0, z. In other words they are all sitting on the ground.
To look at them we may want the camera somewhere above the ground so we are looking down on them.
If the viewport is centred on the camera (the default) then all the objects will appear in the bottom half of the screen.
Now this may be exactly what we want but the firmware allows you to pan the viewport up and down and/or left and right relative to the camera.
So in our example we could pan the viewport down to better frame the image on the screen.
This does not change the perspective of the image, that is locked in by the relative positions of the object and the camera.
It merely allows us to frame the image better given our limited screen resolution

DRAW3D SHOW

DRAW3D WRITE

DRAW3D SHOW n, x, y, z [,nocull]
DRAW3D WRITE n, x, y, z [,nocull]
This says that we want to position the centre of the object at coordinates x, y, z in our virtual 3D world. This command also projects the object 'n' onto the imaginary screen at "viewpoint" from the camera. The mechanism of projection interprets the relative position of the object in 3 dimensions and does full perspective compensation taking into account the relative positions of each vertex in three dimensional space relative to the viewplane and the x,y coordinates of the camera. As it displays the object it calculates the screen coordinates of the minimum rectangle into which the rendered object fits. This allows a subsequent SHOW command (but not WRITE command) to erase

the previous render and draw the object onto a clean screen.
The firmware uses the technique of a combination of hidden face culling (faces pointing away from the viewer are not drawn) and Painter’s algorithm (furthest face drawn first) to render the 3D object onto the viewplane.
Set nocull to 1 to disable hidden face culling and rely on Painter's algorithm for the display. Omit or set to 0 for normal culling. Setting nocull to 1 allows you to see inside hollow objects but decreases rendering efficiency and may result in artefacts on objects that have concave faces.

DRAW3D HIDE

DRAW3D HIDE n [,n1 [,n2...]]
This command hides one or more 3D objects that have been rendered using SHOW by clearing the screen in the area occupied by the object.

DRAW3D HIDE ALL

DRAW3D HIDE ALL
This command does the same as HIDE but for all 3D objects

DRAW3D CLOSE

DRAW3D CLOSE n [,n1 [,n2...]]
This command both hides any 3D objects that have been rendered using SHOW and deletes the object in memory freeing up both the memory used and the object "slot"

DRAW3D CLOSE ALL

DRAW3D CLOSE ALL
This command does the same as CLOSE but for all 3D objects

DRAW3D DIAGNOSE

DRAW3D DIAGNOSE objectno, x, y, z
This command calculates the position of the 3D object and then lists the faces in depth order with an analysis of whether they would be hidden or not based on their surface normal. Using the command will help to test 3D object definitions to make sure all faces are correctly specified with their vertices correctly clockwise ordered when looking at the face from the outside of the object.

A simple example

Now let’s display a 3D object on the screen using the DRAW3D SHOW and DRAW3D WRITE commands.

We set up the data defining the cube as follows:

We set up the vertices using:

```
DIM FLOAT vertices(2,7) = (-1,1,-1,  1,1,-1,  1,-1,-1,  -1,-1,-1,  -1,1,1,  1,1,1,  1,-1,1,  -1,-1,1)
DIM INTEGER facecount(5)=(4,4,4,4,4,4)
DIM INTEGER faces(23)=(0,1,2,3,  1,5,6,2,  0,4,5,1,  5,4,7,6,  2,6,7,3, 0,3,7,4)
DIM INTEGER colours(6)=(rgb(blue), rgb(green), rgb(yellow), rgb(cyan), rgb(red), rgb(magenta), rgb(yellow))
DIM INTEGER edge(5)=(6,6,6,6,6,6)
DIM INTEGER fill(5)=(0,1,2,3,4,5)
```

So we have 8 vertices and 6 faces. We will use camera number 1 to display the cube and we will create object number 1

```
DIM n=1, nv=8, nf=6, camera=1
```

But first it is worth revisiting one aspect of object creation. Our example cube has a side length of 2 so even at 100% it would only cover a maximum of 4 pixels on the screen - not very useful. Luckily there is a simple command we can use to scale the vertices before creating the object.

We can make the cube the size we want by simply multiplying all the elements in the vertex array by the amount we want. So to make the length of each side 200 we can use:

```
MATH SCALE vertices(), 100.0, vertices()
```

Now we can create the object in memory

```
DRAW3D CREATE  n, nv, nf, camera, vertices(), facecount(), faces(), colours(), edge(), fill()
```

We need to set up the camera specified for the object in the create command before we can display it. To do this we use the command

```
DIM INTEGER viewplane=500
DRAW3D CAMERA n, viewplane
```

We are not specifying any optional parameters for the CAMERA command so this says position the camera at 3D coordinates 0, 0, 0 with a "viewplane" 500 units in front of the camera along the z axis and orthogonal to it. In our world the camera is always at a z position of zero and objects will always be positioned with a positive value of z.

As defined above there are two commands for displaying (rendering) a 3D object: DRAW3D WRITE and DRAW3D SHOW. The only difference is that, assuming the object was already displayed, the SHOW command will clear a rectangle on the current write page sufficient to remove the existing object image before displaying it whereas DRAW3D WRITE just overwrites whatever in on the write page with the 2D representation of the object.

The syntax for both commands is the same so will concentrate on DRAW3D SHOW

```
DIM INTEGER x=0, y=0, z=1000
PAGE WRITE 1
DRAW3D SHOW n, x, y, z
```

It couldn't get any easier. This says that we want to position the centre of the object at coordinates x, y, z in our virtual 3D world. The camera specified for our object is at a position 0, 0, 0. This command projects the object 'n' onto the imaginary screen at "viewpoint" from the camera.
We will output to page 1 so we can get a nice tear free display.

Unlike sprites, 3D objects do not store the background image when the object is written or restore it when the object moves. It is recommended that 3D objects are written onto a blank page and are blitted or page copied (with transparency) onto the background image. Alternatively, putting 3D objects onto page 1 in 12-bit mode with the background on page 0 will work very well.

The mechanism of perspective is quite complex but the second example program, attached below, shows how the movement of the camera (represented by the cursor) changes the image of the cuboid with the one nearest the camera bigger than the one further away, the bottom of the octahedron bigger than the top and the side of the octahedron nearest the camera bigger than the side further away.

One last point. Neither the x, y positions of 3D objects or the camera positions are constrained by the screen size. In fact they can go from -32766 to + 32766 (0-32766 for Z for the object - the camera is always at z=0). There are lots of combinations where a 3D object will render off the physical display page. This is perfectly acceptable and allow valid objects to exist in the "world" without the constraints of screen space.

Now a static cube is pretty boring so let's make it rotate continuously:

```
DIM FLOAT yaw=rad(1), pitch=rad(2), roll=rad(0.5)
DIM FLOAT q(4)
DO
        MATH Q_EULER yaw, pitch, roll, q()
        DRAW3D ROTATE q(),n
        DRAW3D show n,x,y,z
        INC yaw,RAD(1)
        INC pitch,RAD(2)
        INC roll,RAD(0.5)
        PAGE COPY 1 to 0,B
LOOP
```

And here is the whole program:

Simple cube example

```
DIM FLOAT vertices(2,7) = (-1,1,-1,  1,1,-1,  1,-1,-1,  -1,-1,-1,  -1,1,1,  1,1,1,  1,-1,1,  -1,-1,1)
DIM INTEGER facecount(5)=(4,4,4,4,4,4)
DIM INTEGER faces(23)=(0,1,2,3,  1,5,6,2,  0,4,5,1,  5,4,7,6,  2,6,7,3, 0,3,7,4)
DIM INTEGER colours(6)=(rgb(blue), rgb(green), rgb(yellow), rgb(cyan), rgb(red), rgb(magenta),
rgb(yellow))
DIM INTEGER edge(5)=(6,6,6,6,6,6)
DIM INTEGER fill(5)=(0,1,2,3,4,5)
DIM n=1, nv=8, nf=6, camera=1
MATH SCALE vertices(), 100.0, vertices()
DRAW3D CREATE  n, nv, nf, camera, vertices(), facecount(), faces(), colours(), edge(), fill()
DIM FLOAT yaw=rad(1), pitch=rad(2), roll=rad(0.5)
DIM FLOAT q(4)
DIM INTEGER viewplane=500
DRAW3D CAMERA n, viewplane
DIM INTEGER x=0, y=0, z=1000
PAGE WRITE 1
DRAW3D SHOW n, x, y, z
DO
        MATH Q_EULER yaw, pitch, roll, q()
        DRAW3D ROTATE q(),n
        DRAW3D show n,x,y,z
        INC yaw,RAD(1)
        INC pitch,RAD(2)
        INC roll,RAD(0.5)
        PAGE COPY 1 to 0,B
LOOP
```

Concave shape fixed in 3D space but with moving camera

```
option explicit
option default float
mode 1,8
dim integer viewplane = 500
const camera = 1
dim q(4)
dim yaw=rad(1),pitch=rad(2),roll=rad(0.5)
dim integer nv=9, nf=9 ' cube has 9 vertices and 9 faces
'array to hold vertices
dim v(2,nv-1)=(-1,1,-1,  1,1,-1, 1,-1,-1, -1,-1,-1, -1,1,1, 1,1,1, 1,-1,1,  -1,-1,1, 0,0,0)
math scale v(),100,v()
' array to hold number of vertices for each face
dim integer fc(nf-1) =(4,4,4,4,4,3,3,3,3)
dim integer
cindex(9)=(rgb(red),rgb(blue),rgb(green),rgb(magenta),rgb(yellow),rgb(cyan),rgb(white),rgb(brown),
rgb(gray),0)
dim integer fcol(nf-1)=(9,9,9,9,9,9,9,9,9)
dim integer bcol(nf-1)=(0,1,2,3,4,5,6,7,8)
'array to hold vertices for each face
dim integer fv(math(sum fc())-1)=(1,5,6,2, 1,0,4,5,  0,3,7,4,  5,4,7,6, 2,6,7,3, 0,1,8, 1,2,8, 3,8,2 , 3,0,8)
```

```
draw3d create 1, nv, nf, camera, v(), fc(), fv(),cindex(),fcol(),bcol()
dim integer c
page write 1
'draw3d diagnose 1,0,0,1000
gui cursor on 1,0,mm.vres\2
box 0,0,mm.hres,mm.vres
do
for c=-399 to 399
 gui cursor c+400,MM.Vres\2-c*600/800
 draw3d camera 1,viewplane,c,c*600/800
 math q_create roll,1,1,1,q()
 draw3d show 1,0,0,1000
 math q_euler yaw,pitch,roll,q()
 draw3d rotate q(),1
 inc yaw,rad(1)
 inc pitch,rad(2)
 inc roll,rad(0.5)
 page copy 1 to 0
 pause 20
next
for c=399 to -399 step -1
 gui cursor c+400,MM.Vres\2-c*600/800
 draw3d camera 1,viewplane,c,c*600/800
 math q_create roll,1,1,1,q()
 draw3d show 1,0,0,1000
 math q_euler yaw,pitch,roll,q()
 draw3d rotate q(),1
 inc yaw,rad(1)
 inc pitch,rad(2)
 inc roll,rad(0.5)
 page copy 1 to 0
 pause 20
next
loop
draw3d close all
```

Wall from a high view with panned viewport

```
option explicit
option default none
option base 0
mode 1,8
DIM INTEGER x=-600,y,z,layer=0
DIM FLOAT Q(4), yaw=0, pitch=0, roll=0
```

```
const camerax=0, cameray=700, viewplane=400, panx= -150, pany=-250
dim integer nv=8, nf=6 ' cube has 8 vertices and 6 faces
const camera1 = 1
dim float vertices(2,7) = (-1,1,-1,  1,1,-1,  1,-1,-1,  -1,-1,-1,  -1,1,1,  1,1,1,  1,-1,1,  -1,-1,1)
dim integer fc(5)=(4,4,4,4,4,4) ' define the number of vertices in each face
dim integer faces(23)=(0,1,2,3,  1,5,6,2,  0,4,5,1,  5,4,7,6,  2,6,7,3, 0,3,7,4) 'define the vertice
dim integer colours(6)=(rgb(blue), rgb(green), rgb(magenta), rgb(cyan), rgb(red), rgb(brown),
rgb(yellow))
dim integer edge(5)=(6,6,6,6,6,6) 'define the colours used for the edges of each face
dim integer fill(5)=(0,1,2,3,4,5) 'define the colours used to fill each face
'
MATH SCALE vertices(), 20, vertices()
dim float slice(7)
math slice vertices(),2,,slice()
math scale slice(),2,slice()
math insert vertices(),2,,slice()
DRAW3D CREATE 1, nv, nf, camera1, vertices(), fc(), faces(), colours(), edge(), fill()
DRAW3D CAMERA 1, viewplane, camerax, cameray, panx, pany
PAGE WRITE 1
circle camerax,MM.VRES-cameray,4,,,rgb(white),rgb(white)
timer=0
for y=0 to 550 step 45
for z=0 to 960 step 85
DRAW3D write 1,x,y,1600-z+layer
'pause 500
next z
if layer=0 then
 layer=40
else
 layer=0
endif
next y
print timer
page copy 1 to 0
do:loop
```

Football with missing face and internal view

```
Option explicit
option default none
mode 2,16
dim float phi=(1+sqr(5))/2
' data for location of verticies for verticesahedron of edge length 2
data 0,1,3*phi
data 0,1,-3*phi
data 0,-1,3*phi
data 0,-1,-3*phi
data 1,3*phi,0
data 1,-3*phi,0
data -1,3*phi,0
data -1,-3*phi,0
data 3*phi,0,1
data 3*phi,0,-1
data -3*phi,0,1
data -3*phi,0,-1
data 2,(1+2*phi),phi
data 2,(1+2*phi),-phi
data 2,-(1+2*phi),phi
data 2,-(1+2*phi),-phi
data -2,(1+2*phi),phi
data -2,(1+2*phi),-phi
data -2,-(1+2*phi),phi
data -2,-(1+2*phi),-phi
data (1+2*phi),phi,2
data (1+2*phi),phi,-2
data (1+2*phi),-phi,2
data (1+2*phi),-phi,-2
data -(1+2*phi),phi,2
data -(1+2*phi),phi,-2
data -(1+2*phi),-phi,2
data -(1+2*phi),-phi,-2
data phi,2,(1+2*phi)
data phi,2,-(1+2*phi)
data phi,-2,(1+2*phi)
data phi,-2,-(1+2*phi)
data -phi,2,(1+2*phi)
data -phi,2,-(1+2*phi)
data -phi,-2,(1+2*phi)
data -phi,-2,-(1+2*phi)
data 1,(2+phi),2*phi
data 1,(2+phi),-2*phi
data 1,-(2+phi),2*phi
data 1,-(2+phi),-2*phi
data -1,(2+phi),2*phi
data -1,(2+phi),-2*phi
data -1,-(2+phi),2*phi
data -1,-(2+phi),-2*phi
data (2+phi),2*phi,1
data (2+phi),2*phi,-1
data (2+phi),-2*phi,1
data (2+phi),-2*phi,-1
data -(2+phi),2*phi,1
data -(2+phi),2*phi,-1
data -(2+phi),-2*phi,1
data -(2+phi),-2*phi,-1
data 2*phi,1,(2+phi)
data 2*phi,1,-(2+phi)

data 2*phi,-1,(2+phi)
data 2*phi,-1,-(2+phi)
data -2*phi,1,(2+phi)
data -2*phi,1,-(2+phi)
data -2*phi,-1,(2+phi)
data -2*phi,-1,-(2+phi)
' 12 faces with 5 sides
data 0,28,36,40,32
data 33,41,37,29,1
data 34,42,38,30,2
data 3,31,39,43,35
data 4,12,44,45,13
data 15,47,46,14,5
data 17,49,48,16,6
data 7,18,50,51,19
data 8,20,52,54,22
data 23,55,53,21,9
data 26,58,56,24,10
data 25,57,59,27,11
' 20 faces with 6 sides
data 32,56,58,34,2,0
data 0,2,30,54,52,28
data 29,53,55,31,3,1
data 1,3,35,59,57,33
data 13,37,41,17,6,4
data 4,6,16,40,36,12
data 5,7,19,43,39,15
'data 14,38,42,18,7,5
data 22,46,47,23,9,8
data 8,9,21,45,44,20
data 10,11,27,51,50,26
data 24,48,49,25,11,10
data 36,28,52,20,44,12
data 13,45,21,53,29,37
data 14,46,22,54,30,38
data 39,31,55,23,47,15
data 16,48,24,56,32,40
data 41,33,57,25,49,17
data 42,34,58,26,50,18
data 19,51,27,59,35,43
'
dim float q1(4)
dim float yaw=rad(1),pitch=rad(2),roll=rad(0.5)
dim integer i, j, nf=31, nv=60, camera=1
dim float vertices(2,59)
' read in the coordinates of the verticies and scale
for j=0 to 59
for i=0 to 2
  read vertices(i,j)
  vertices(i,j)=vertices(i,j)*50
next i

next j
'math scale vertices(),50,vertices()
'
dim integer faces(173)
for i=0 to 173
 read faces(i)
next i
dim integer fc(30)=  (5,5,5,5,5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6)
dim integer colours(2)=(rgb(red),rgb(white),rgb(black))
dim integer edge(30)=(2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2)
dim integer fill(30)=(0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
math q_create rad(2),1,0.5,0.25,q1()
draw3d create 1, nv,nf, camera, vertices(), fc(), faces(), colours(), edge(), fill()
draw3d camera 1,800,0,0
'draw3d diagnose 1,mm.hres\2,mm.vres\2,1000
page write 1
draw3d show 1,0,0,1000,1
do
 math q_euler yaw,pitch,roll,q1()
 inc yaw,rad(1)
 inc pitch,rad(2)
 inc roll,rad(0.5)
 draw3d rotate q1(),1
 draw3d show 1,0,0,1000,1
 page copy 1 to 0,b
loop
```