ObjectDesign

COLLABORATORS			
	TITLE : ObjectDesign		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 29, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ObjectDesign

1.1 ObjectDesign.guide

Documentation for Filled Vector Module for AmigaE 3.0+, object design

1.2 Creating Objects

Object Creation

Creating objects by hand is very difficult ... unfortunately,
this is the iinly real way to create objects at the moment.

The utility can however be used to simplify
the entry of data into the computer, and if you dont mind
being limited to objects made of triangles, you can even use it
to read in objects created with Imagine (not without more
work mind you).

The following steps need to be taken whether designing objects
with or without using ReadNFF, its just the final stages that
can be simplified using it.

Starting

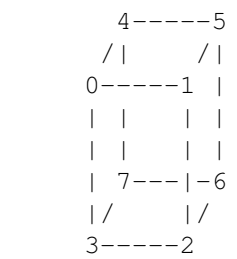
First, you need an idea. This can often be difficult, especially
since you will want to create the object so it uses as few polygons
as possible, yet still gets drawn correctly.

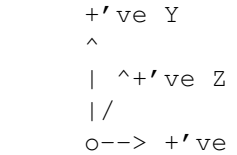
Working out the points

Once you have the idea though, it helps to draw it down on a piece of paper. You need to be able to identify and number all of the vertices or points in the object. The actual coordinates of these vertices will also need to be calculated - in 3d. Drawing several views of the object may help with this.

After this stage, you should have a list of coordinates, and you should also have identified each of the points by an index (start it at 0).

For example, a cube.





The numbers show the points we have assigned to each of the 8 corners of the cube.

Say, we want the cube to be 200*200*200 in size, we also would like to have centered about its centre of gravity (in the middle) so we place each point 100 from the origin.

This, our points table becomes:

```
-100,100,-100 0
100,100,-100  1
100,-100,-100 2
-100,-100,-100  3
-100,100,100  4
100,100,100   5
100,-100,100  6
-100,-100,100 7
```

Ok, so far so good, this is actually the easy part for most objects ...

Working out the polygons

This is where it starts to get tricky. Each "face" of an object (i.e. polygon) requires several pieces of information.

Firstly, it requires a set of line end-points which go up to make
lines which form the polygon edges. This list of points must
form a CLOSED shape. It can however form this shape in whatever
way it likes, including crossing lines, and even several closed
shapes. If multiple shapes are present, then they act as if each
of them is complemented together (i.e. they will be transparent
where they overlap). There is one restriction however for
each polygon or face formed, all of the points of the face
should be in the same "plane" otherwise it wont draw correctly.

So, going back to our cube example, the front face would consist
of the following linking lines:

point 0 connected to 1, 1 to 2, 2 to 3 and 3 to 0.

This is 4 lines, so our polygon definition becomes:

[4, 0,1,1,2,2,3,3,0]:INT

The other polygons can be similarly defined.

Working out the order

One very important thing to note about how this vector module works
is that individual polygons are NOT depth sorted! And yet, even
very complex shapes like the 2 from the A1200 logo are drawn
correctly. How can this be so?

This is where the 2 other pieces of information provided to the
rendering engine are used. Firstly, the "crossn" members of
the face OBJECT are used to describe which way a given polygon
is facing (out or in), this allows the render engine to automatically
forget about drawing the back sides of objects (known as backface
removal). Secondly, the order in which the polygons are drawn
(which is the same as the order they are defined in the creation
function) becomes important.

This takes a bit of a mind-leap, but if you think about it carefully
enough, you can see that only drawing faces of objects that are indeed
facing you, and drawing them in the right order, you can get away
without having to do any depth sorting for most objects you are
likely to try to design.

Take a table as an example. If you wish to create a 3d table, with
a solid, filled tabletop, and 4 square legs, you order the polygons
like this:

draw the underside of the table
draw the 2 inside (the ones facing the centre of the table) faces of all
 legs
draw the 2 outside faces of all of the legs
draw the sides of the table
draw the tabletop

Since faces which are not "facing" the viewer are automatically culled,
it can see that the above scheme can be used to draw the table
accurately from EVERY concievable external view.

And whats more, since no depth sorting step is required, computation time is reduced.

Some objects however cannot be drawn accurately all the time using this technique - either make sure you get it right for the cases where it is visible, break it into seperate objects, or simplify your object :)

In the case of this simple cube, the order is unimportant, since there are no overlapping polygons (when drawn).

Cross product indices

The cross product points (the "crossn" members mentioned above) are very necessary though. Its not too difficult to work out these either, provided you have a good spatial sense (unless you want to keep drawing rotated views of your object anyway!).

What you need to do for each face, is work out 3 points that are in the same plane as the face (usually 3 points of the face/polygon itself). Ideally, the 3 points should be as far apart as possible, and lines from the middle one to the other 2 should be close to 90 degrees (this is not necessary, but it provides for a more accurate result). These 3 points must then be ordered so that if you were to draw a line from point 0 to point 1, then point 1 to point 2, the lines would move around in a clockwise direction - this is if the object were rotated so that the given polygon was facing you.

So, for the cube, for front face, the 3 points could be:
0,1,2
They could also be 1,2,3 or 2,3,0 or 3,0,1 as well.

As an example, the back face would have to use 6,5,4 or and combination in the same direction. If we used 4,5,6 then it would mean that that back face was facing inwards instead of outwards, and the object wouldn't be drawn properly. Remember, the points need to traverse in a clockwise direction around the polygon.

Incidentally, theres nothing stopping you defining 2 faces with the same points, but one facing outwards, and one inwards, if you really needed two sided polygons.

Finally, choose which colour index you want for each of the polygons, taking into account the palette of the screen on which you willbe working.

You now basically have all the information required to generate a new vector object. The following sections describe how you do this by hand, or alternatively, how to use to help you.

Putting it together, all by hand

For the cube, we have:

```
front face:
  [0,1,2, 1, [4, 0,1,1,2,2,3,3,0]:INT, 0]:face

   ^^^^^ - The 3 cross product indices calculated above

         ^ - The colour of this polygon

             ^ - The number of edges in the polygon

                ^^^^^^^^^^^^^^^ - The edge link table,
                                  pairs of line endpoints

              ^ - padding
```

This can be done for all of the 6 faces of the cube to create
the face array that needs to be fed into the newVectorObject()
call.

With a bit more work, we get:

```
  [0,1,2, 1, [4, 0,1,1,2,2,3,3,0]:INT, 0, -> front
  6,5,4, 2, [4, 4,5,5,6,6,7,7,4]:INT, 0,  -> back
  2,1,5, 3, [4, 1,5,5,6,6,2,2,1]:INT, 0,  -> right
  4,0,3, 4, [4, 4,0,0,3,3,7,7,4]:INT, 0,  -> left
  1,0,4, 5, [4, 0,1,1,5,5,4,4,0]:INT, 0,  -> top
  7,3,2, 6, [4, 3,2,2,6,6,7,7,3]:INT, 0]:face -> bottom
```

And in this case, the ordering is unimportant.

Although above i usually defined the polygon using point indices
which followed on from each other around the edges of the
polygon, any ordering is allowed. For example, the front
could also have been defined as:

```
  [0,1,2, 1, [4, 3,2,0,1,0,3,1,2]:INT, 0, -> front
```

Since, the lines formed still create the same shape.

Combining everything so far, we can come up with the function call
required to make our cube exist.

```
 cube:=newVectorObject(0, -> basic type
  8,      -> 8 points
  6,      -> 6 faces
    [-100,100,-100, -> 0  -> points array
    100,100,-100, -> 1
    100,-100,-100,  -> 2
    -100,-100,-100, -> 3
    -100,100,100, -> 4
    100,100,100,  -> 5
    100,-100,100, -> 6
    -100,-100,100]:INT, -> 7  -> faces array below
  [0,1,2, 1, [4, 0,1,1,2,2,3,3,0]:INT, 0, -> front
  6,5,4, 2, [4, 4,5,5,6,6,7,7,4]:INT, 0,  -> back
  2,1,5, 3, [4, 1,5,5,6,6,2,2,1]:INT, 0,  -> right
  4,0,3, 4, [4, 4,0,0,3,3,7,7,4]:INT, 0,  -> left
  1,0,4, 5, [4, 0,1,1,5,5,4,4,0]:INT, 0,  -> top
```

```
  7,3,2, 6, [4, 3,2,2,6,6,7,7,3]:INT, 0]:face); -> bottom
```

And thats it! _phew_ Have a look at some of the examples, you
can appreciate how difficult some of them were to make :)

Putting it together, using ReadNFF

Checkout the section for this.

1.3 readnff

ReadNFF information

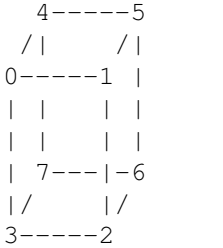
This is a badly written, kludgy, and awkward to use utility
that can however greatly simplify object creation.

There not many that need explaining.

It reads in a file in NFF format (neutral file format?) and
converts it to sourcecode for the function call to create
the object in AmigaE format.

Apart from designing objects using NFF by hand, there are
utilities such as tddd2nff which can be used to convert
Imagine objects to NFF, then NFF can be read by ReadNFF
to create a function call. However, imagine doesn't
know anything about polygon ordering, or correct
cross product direction (since imagine uses triangles,
ReadNFF is able to create the right cross product for
you - but it will often be in reverse of what you want)
doesn't work. So, it often requires a lot of hand-work
to get the objects to work anyway ...

Going back to the cube example, we'll see how to use
ReadNFF to do some of the work.



For the ReadNFF file, what we need to do is to define all
of the coordinates for each polygon. We can also try
to order these points so that the polygons themselves
are defined in clockwise order (to help the code
create accurate cross product points), but this wont
always work if the object has >3 points because
of the point optimisation method used.

So, we come up with the following definition for our cube:

```
# a cube, in NFF format
# Comments start with #, and can be anywhere outside of
# polygon definitions
# A polygon definition is started with a p, followed by
# exactly one space, then a number which defines
# the number of points in that polygon
# There is 1 point per line, seperated with spaces
# with the x coordinate flush left (no leading
# spaces), with spaces (not tabs!) seperating
# each number
# front face
p 4
-100 100 -100
100 100 -100
100 -100 -100
-100 -100 -100
# back face
p 4
-100 -100 100
100 -100 100
100 100 100
-100 100 100
# right
p 4
100 100 -100
100 100 100
100 -100 100
100 -100 -100
# left
p 4
-100 100 -100
-100 -100 -100
-100 -100 100
-100 100 100
# top
p 4
-100 100 -100
-100 100 100
100 100 100
100 100 -100
# bottom
p 4
100 -100 -100
-100 -100 -100
-100 -100 100
100 -100 100
```

Ok, this can be fed into ReadNFF, and it comes up with:
(with a little editing)

```
 obj:=newVectorObject(0,8,6,
        [-100,100,-100,
        100,100,-100,
        100,-100,-100,
```

```
        -100,-100,-100,
        -100,-100,100,
        100,-100,100,
        100,100,100,
        -100,100,100]:INT,
  [0,1,2,0,[4,0,1,1,2,2,3,3,0]:INT,0,
  4,5,6,1,[4,4,5,5,6,6,7,7,4]:INT,0,
  1,6,5,2,[4,1,6,6,5,5,2,2,1]:INT,0,
  0,3,4,3,[4,0,3,3,4,4,7,7,0]:INT,0,
  0,7,6,4,[4,0,7,7,6,6,1,1,0]:INT,0,
  2,3,4,5,[4,2,3,3,4,4,5,5,2]:INT,0]:face

The editing involved:
  deleting a spurios point at the end of the point array
    (ok, so its buggy as hell!)
  adding the []:face to the face array, and moving
    it from the start of the output to the end
  and thats about it ...

So, will this object work all on its own?  I think so, since
because the way the points were ordered in the file, they
came out the same as the original.  In this case, no more
editing is required, luckily.  All that needs to be modified
is the colours, which isn't too difficult.

With more complex objects, you may have to manually enter
in the cross product indices as well as the colours.  It
does save a bit of work though.    shows
an example of an object created in Imagine 2.0 (using
create primitive/torus), which was then hand-edited to make
it work (every second polygon needed its cross indices
reversed).

Hopefully one day i'll sit down and design either a proper
script language to make designing objects simpler, or
a 3d editor ... (although, other commitments may affect
this!).
```

1.4 ReadNFF command line options

```
 ReadNFF options:

 When you run it, it will open a window and the file, and as the
 file is read in, a wireframe version of the object will
 be displayed.  The output of the program is simply dumped
 to the cli inwhich it was run.  Once finished, it will
 automatically close the window.

 NAME/A
  Supply the filename of the NFF file to read.  Not much
  error checking is done, so if this is in an invalid
  format, you're not going to get much more than a bloody
  mess out of it :)

 SHIFT/N
```

```
  Amount to shift values before they are output.  This is
  only important with floating point input, or where
  some scaling is needed.  Each number conforms to a shift
  up by 1.  So, a shift value of 2 will mean the values
  are multiplied by 4 before being converted to ingeter
  format.
```

1.5 Modifying Objects

```
Modifying Objects

Currently there is really only one way to modify objects - the
points in the object can be modified.

You can't even modify the colours of polygons and so on
for copied objects (which would be handy ...) this should
be fixed soon ...

Playing with Points

Using  the application programmer
has direct access to the coordinate table used by the object.
This can be modified in any way you see fit, say by scaling
the object and so on.

One thing that must be noted thouh, is that the points should
continue to retain the same relationship to each other as
when the object was defined - otherwise the face setup,
backface removal and so on will not operate correctly.

This still allows a bit of scope for object morhping and things
like that.  (If i had the effort, i'd give you an example
of morphing objects :)


For objects that have been copied using
you can modify the points freely, as each copy of the object
gets its own points.  For cloned objects (
the point lists are NOT copied, so in this case, you must
be careful you dont accidentally change the wrong points!


When objects have been copied, this is where being able
to modify them becomes useful.  You can use the rotation
and scaling functions in matrix operations to
modify the copied object to create new versions.
```

1.6 Using Objects

```
Using Objects

Its all very well being able to create and bend objects and so
```

on, but how do you get them onto the screen usefully?

One-off objects, like a swirling Zed logo are easy enough.
This uses a vector object list to allow the system itself
to handle the nitty gritty of depth sorting and so on,
and then uses the object's positions to position the
letters within their own "space".

Using the functions, you
could build up your own library of functions designed to
position the objects (using thier position specifiers),
rotate them to the view you want, and then use the
vector object list functions to do the depth sorting
and rendering for you.

Finally, you could code everything yourself except for the
object rendering function, and just use
to do the hard work, at reasonable speed.