

**HowToCode7**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> HowToCode7	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY		November 28, 2024
		<i>SIGNATURE</i>

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>HowToCode7</b>	<b>1</b>
1.1	HowToCode: Vectors . . . . .	1
1.2	Introduction . . . . .	1
1.3	Preface . . . . .	2
1.4	Introduction To Vectors . . . . .	3
1.5	Coding Techniques . . . . .	5
1.6	Vector Rotation . . . . .	8
1.7	Polygons! . . . . .	13
1.8	Planes in three dimensions . . . . .	17
1.9	Special techniques - Sorting Algorithms . . . . .	18
1.10	vectorballs . . . . .	21
1.11	sources . . . . .	21
1.12	Optimised rotation matrix calculation . . . . .	22
1.13	A line draw routine for filled vectors . . . . .	24
1.14	Quicksort in 68000 assembler . . . . .	25
1.15	Insert Sort in 68020 assembler . . . . .	26
1.16	Further Information . . . . .	28

---

## Chapter 1

# HowToCode7

### 1.1 HowToCode: Vectors

How to code vectordemos

-----

Introduction

- 1 Preface
- 2 Introduction to vectors
- 3 Coding techniques
- 4 Vector rotations
- 5 Polygons
- 6 Planes in three dimensions
- 7 Special techniques - Sorting Algorithms
- 8 Special techniques - Vector Balls
  
- A Example sources
- B Further Information

### 1.2 Introduction

An introduction by Asterix of Movement ...

=====

This text is an addition to How to code written by Comrade J of SAE.  
 It was written by Carl-Henrik Skårstedt during his easter holidays.  
 Hi ho to all friends of movements....  
 Any comments on this text/additions to vectors.txt should be Emailed to  
 mnlcst@cs.umu.se, or by mail: Rullstensg6-210,90655Ume},Swe  
 If you think I should be clearer on some points, or if you think I've  
 totally forgotten something, just report this for later issues..

```

      |
      |
  /|\ /|
  | /
  |/_____ \
  
```

/

## 1.3 Preface

### 1. Preface

=====

The sources of this text has more or less indirectly been some books from my school. Some sources worth mentioning are:  
Elementary Linear Algebra (by Howard Anton, released by John Wiley)  
Calculus - A complete course (By Robert K. Adams)  
The DiscWorld series (by T. Pratchett)

By reading this text, you should also be able to know what it is you're doing. Not just converting given formulas to 680x0 code, but also know what the background of it is. If you know the theory behind your routine, you also know how to optimize it! NO text will here mention GLENZ-vectors, since they are amazingly depressive.

This text is meant for democoders on all computers that supports a good graphic interface, which is fast enough to do normal concave objects in one frame (not PC).  
sqr() means sqare root in this text.

I'm curious about what support Commodore has for this kind of programming in their latest OS, it could be a great Idea if rotations etc that used many multiplications was programmed in ROM. The methods described are used by most well-known demo-coders in "vector" demos.

The rights of this part stays with the author.  
I've coded Blue House I+2, most of Rebels Megademo II, my own fantastic and wonderful cruncher (not released), Amed (also not released), some intros, and the rubiks snake in Rebels Aars-intro, and the real slideshow from ECES. Sorry for most of my demos not working on other machines than real A500's, but that's the only computer I've used for bugtesting.

The meaning of this text is that it shall be a part of How To Code.txt and that the same rules works for this text as for that.

The rights of this part stays with the author.  
Sourcecodes should work with most assemblers except for Insert sorting, which needs a 68020 assembler.

Hi to all my friends who really supported me by comments like:  
"How can you write all that text?"  
"Who will read all that?"  
"Can I have it first, so I can get more bbs-access?"  
"Why are you writing that?"  
"I want to play Zool!" (-My youngest brother)  
"My dog is sick..."  
"You definitely have the right approach to this!"

"" (-Commodore Sweden)  
 (But in swedish of course!)

The reason why Terry Pratchetts DiscWorld series is taken as a serious source is that he is a great visualizer of strange mathematical difficulties. If you ever have problems with inspiration, sit back, read and try to imagine how demos would look like in the DiscWorld... (Glenz-Turtles?)

Now read this text and impress me with a great AGA-demo...

(C) MOVEMENT 1993.

"Death to the pis" /T. Domar

## 1.4 Introduction To Vectors

### 2. Introduction to vectors

=====

What is a vector?

-----

If you have seen demos, you have probably seen effects that is called, in a loose term, vectors. They may be balls, filled polygons, lines, objects and many other things.

The thing that is in common of these demos are the vector calculations of the positions of the objects. It can be in one, two or three Dimensions (or more, but then you can't see the ones above 3)

You can for example have a cube. Each corner on the cube represent a vector TO the center of rotation.

All vectors go FROM something TO something, normally we use vectors that goes from a point (0,0) to a point (a,b).

This vector has the quantity of (a,b).

Definition of vector:

A QUANTITY of both VALUE and DIRECTION.

or, in a laymans loose terms: a line.

A line have a length that we can call r, and a direction we can call t.

We can write this vector (r,t) = (length,angle).

But there is also another way, which is more used when dealing with vector objects with given coordinates.

The line from (0,0) to (x,y) has the length  $\text{sqr}(x*x+y*y)$  and that is the VALUE of the vector. The direction can be seen as the angle between the x-axis and the line described by the vector.

If we study this in two dimensions, we can have an example vector as following:

^ y  
 |     \_ . (a,b)



$V = (v_x, v_y, v_z)$ ,  $W = (w_x, w_y, w_z) \Rightarrow$   
 $\Rightarrow U = (v_x + w_x, v_y + w_y, v_z + w_z)$

- \* The negation of a vector  $U = -V$  is defined as  
 $V = (x, y, z) \Rightarrow U = (-x, -y, -z)$
- \* The difference between two vectors  $U = V - W$  are defined as  
 $U = V + (-W)$
- \* A vector between two points (FROM  $P_1(x_1, y_1, z_1)$  TO  $P_2(x_2, y_2, z_2)$ )  
 can be computed:  
 $V = (x_2 - x_1, y_2 - y_1, z_2 - z_1, \dots)$   
 (V goes FROM  $P_1$  TO  $P_2$ )
- \* A vector can be multiplied by a constant like:  
 $U = k * V$   
 $(x * k, y * k, z * k) = k * (x, y, z)$
- \* A coordinate system can be "Translated" to a new point with the  
 translation formula:  
 $x' = x - k$   
 $y' = y - l$   
 $z' = z - m$   
 Where  $(k, l, m)$  is the OLD point where the NEW coordinate system  
 should have its point  $(0, 0, 0)$   
 This is a good operation if you want to ROTATE around A NEW POINT!
- \* A vector can be rotated (Check chapter 4)  
 The vector is always rotated around the point  $(0, 0, 0)$  so you  
 may have to TRANSLATE it.
- \* We can take scalar product and cross-product of vectors  
 (see any book about introduction to linear algebra  
 for these. everything is evaluated in this text, so you  
 don't have to know what this is)

## 1.5 Coding Techniques

### 3. Coding techniques

=====

Presenting a three dimensional point on a two dimensional screen

-----

Assume that you have a point in space (3d) that you want to  
 take a photo of. A photo is 2d, so this should give us  
 some sort of an answer.

Look at the picture below:

```

Point
/   Screen ("photo")
.  | /
 \ | ^y
  \| |
   | \|

```

```

<---+--x <- Eye of observer
z   |
    |
    |
    |

```

Inspecting this gives us the following formula:

Projected Y = Distance of screen \* Old Y / ( Distance of point )  
 (The distances is of course the Z coordinates from the Eyes position)

And a similar way of thinking gives us the projection of X.

```

New Y=k*y/(z+dist)
X=k*x/(z+dist)

```

(where k is a constant for this screen, dist is the distance from the ROTATION point to the EYE on the Z-axis)

A way of presenting real numbers with Integers

---

Until now we have only seen a lot of formulas, but how can we use them in Assembler where we only can have bytes/words/longwords? (If you don't have a FPU, and only want people with FPU's to be able to see your demos)

For 68000 coding (compatible with all better processors) it is comfortable to be able to do multiplications etc. with words (680x0, {x>=2}) can do it with longwords, but this won't work very good with lower x's).

But we need the fract parts of the numbers too, so how do we do? We can try to use numbers that are multiplied by a constant p. Then we can do the following operation:

```
[cos(a)*p] * 75      (for example from a list with cos(x) mult. with p)
```

But as you can see this number grows for each time we do another multiplication, so what we have to do is to divide by p again:

```
[cos(a)*p] * 75 / p
```

If you are a knower of digital electronics, you say "Oh no, not a division that takes so long time". But if you choose p carefully (i.e. p = 2 or 4 or 8 ...) you can use shifting instead of clean division. Look at this example:

```

mulu    10(a0),d0      ;10(a0) is from a list of cos*256 values
asr.l   #8,d0         ;and we "divide" by 256!

```

Now we have done a multiplication of a fixed point number!

(A hint to get the error a little smaller:

clear a Dx-register and use an addx after the asr,

---

and you will get a round-off error instead:

```

    moveq    #0,d7
    :
    :
    mulu     10(a0),d0
    asr.l    #8,d0
    addx.l   d7,d0
    :
    rts

```

This halves the error!)

The same thinking comes in with divisions, but in the other way:

```

    :
    ext.l    d0
    ext.l    d1
    asl.l    #8,d0           ;"Multiply" by 256
    divs     d1,d0           ;and divide by z*256 ...
    :
    rts

```

Additions and subtractions are the same as normal integer operations: (no shifting needed)

```

    :
    add.w    10(a0),d0
    :
    :
    sub.w    16(a1),d1
    :

```

So, With multiplications you MUL first, then LSR.  
With divisions you LSL first, then DIV.

If you wish to have higher accuracy with the multiplications, the 68020 and higher processors offers a cheap way to do floating point operations (32-bit total) instead. You can also do integer multiplications 32\*32->32, and use 16-bit coses and sins instead, which enables you to use 'swap' instead of 'lsr'.

How can I use Sin and Cos in my assembler code?

-----  
The easiest and fastest way is to include a sinus-list in you program. Make a basic-program that counts from 0 to 2\*pi, for example 1024 times. Save the values and include them into your code.

If you have words and 1024 different sinus values then you can get sinus and cosinus this way:

```

    lea      sinuslist(pc),a0           ;sinuslist is calculated list
    and.w    #$7fe,d0                   ;d0 is angle
    move.w   (a0,d0.w),d1                ;d1=sin(d0)

```

```

add.w    #$200,d0
and.w    #$7fe,d0
move.w   (a0,d0.w),d0      ;d0=cos(original d0)
:
:
```

Your program could look like this: (AmigaBasic, HiSoft basic)  
(NEVER use AmigaBasic on other processors than 68000)

```

open "ram:sinuslist.s" for output as 1
pi=3.141592654#
vals=1024
nu=0
pretxt=chr$(10)+chr$(9)+chr$(9)+'dc.w'+chr$(9)
for L=0 to vals
  angle=L/vals*2*pi
  y='$'+hex$(int(sin(angle)*255.4))
  nu=nu+1
  if nu=8 then print #1,pretxt;:nu=0 else print #1,',,';
  print #1,y$;
next L
close 1
```

You can of course do a program that calculates the  
sins in assembler code, by using ieee-lib's or  
coding your own floating point routines.  
the relevant algorithm is... (for sinus)

indata: v=angle (given in radians)  
Laps=number of terms (less=faster and more error, integer)

```

1> Mlop=1
   DFac=1
   Ang2=angle*angle
   Talj=angle
   sign=1
   Result=0
2> FOR terms=1 TO Laps
2.1> Temp=Talj/DFac
2.2> Result=sign*(Result+Temp)
2.3> Talj=Talj*Ang2
2.4> Mlop=Mlop+1
2.5> Dfac=Dfac*Mlop
2.6> sign=-sign
3> RETURN sin( )=Result
```

where the returned sin() is between -1 and 1...  
The algorithm uses MacLaurin polynoms, and are therefore  
recommended only for values that are not very far away from 0.

## 1.6 Vector Rotation

### 4. The rotation of vectors

=====

---

\* In two dimensions

Now we know what a vector is, and we want to rotate it.  
This is very simple, if we have a vector given  
with length and angle, we just add the rotation-angle to the  
angle and let the length be like it is:  
rotate  $V=(r,t)$  with  $a \rightarrow V'=(r,t+a)$

But normally we don't have this simple case, we have a  
vector given by two coordinates like:  
 $V=(x,y)$  where  $x$  and  $y$  are coordinates in the  $xy$ -plane.

In THIS text we denote the rotation of a vector  $V=(r,t)$  with  
 $\text{rot}(V,a)$ . With this I mean the rotation of the vector  $V$  with the  
angle  $a$ .

The rotation of this vector could have been done  
by transforming  $V$  to a vector of a length and a direction,  
but since this involves taking squares, tangens, squareroots  
etc., we would like a faster method.  
Here is where trigonometry comes in.

But let us first assume that we have a vector  $V=(x,0)$   
what would be the rotation of this vector?

V  
----->

Now, let us rotate with an angle of  $a$ :

/|\y' /|  
| /  
|V' /  
| /  
|/\a x'  
----->

What is the new vectors components  $(x',y')$  ?

Remember these "definitions":

Cosine:  
Hypotenuse/side close to angle

Sine:  
Hypotenuse/side not close to angle

/|  
/|  
Length>/ |< Length \* sin(a)  
/a |  
'----+

Length \* cos(a)

If we put in this in the original rotation formula ( $V'=\text{rot}(V,a)=V(r,t+a)$ )  
we can see that we can convert  $r$  and  $t$  to  $x$  and  $y$  with:

$$x=r*\cos(t)$$


---

$$y=r*\sin(t)$$

Let's get back to our problem of the rotated vector  $V=(x,0)$ .  
 Here is  $r=x$  ( $=\sqrt{x*x+0*0}$ ),  $t=0$  ( $=\arctan(0/x)$ )  
 if we put this in our formula we get:

$$V=(r,t) \text{ if } r=x, t=0$$

If we rotate this vector with the angle  $a$  we get:

$$V=(r,t+a)$$

And if we translate back to our coordinate denotion:

$$V=(r*\cos(t+a), r*\sin(t+a))=(x*\cos(a), x*\sin(a))$$

^We insert  $x=r, t=0$

And that is the formula for rotation of a vector that has no y-composant.  
 For a vector  $V=(0,y)$  we get:  
 $r=y, t=\pi/2$  ( $=90$  degrees) since we now are in the y-axis,  
 which is 90 degrees from the x-axis.

$$V=(r,t) \Rightarrow V'=(r,t+a) \Rightarrow V'=(r*\cos(t+a), r*\sin(t+a)) \Rightarrow$$

$$V'=(y*\cos(\pi/2+a), y*\sin(\pi/2+a))$$

Now, there are a few trigonometric formulas that says that  $\cos(\pi/2+a)=-\sin(a)$  and  $\sin(\pi/2+a)=\cos(a)$

We get:

$$V'=( y * \sin(a) , y * ( -\cos(a) ) )$$

But if we look in the general case, we have a vector  $V$  that has both  $x$  and  $y$  composants. Now we can use the single-cases rotation formulas for calculating the general case with an addition:

$$Vx'=\text{rot}((x,0),a) = (x*\cos(a) , x*\sin(a))$$

$$+ Vy'=\text{rot}((0,y),a) = ( +y*\sin(a), -y*\cos(a) )$$

---


$$V' =\text{rot}((x,y),a) = (x*\cos(a)+y*\sin(a), x*\sin(a)-y*\cos(a))$$

( $Vx'$  means rotation of  $V=(x,0)$  and  $Vy'$  is rotation of  $V=(0,y)$ )  
 And we have the rotation of a vector given in coordinates!

FINAL FORMULA OF ROTATION IN TWO DIMENSIONS

.. .

$$\cdot \text{rot}((x,y),a)=( \underset{\text{x-composant}}{x*\cos(a)+y*\sin(a)} , \underset{\text{y-composant}}{x*\sin(a)-y*\cos(a)} )$$

\* Three dimensions

Now we are getting somewhere!

In the 2 dimensions case, we rotated  $x$  and  $y$  coordinates,  
 and we didn't see any  $z$  coordinates that changed.  
 Therefore we call this a rotation around the  $Z$  axis.

Now, the simplest thing to do in three dimensions is to still do the same thing, just rotate around any axis to get the new coordinate. Leave out the variable that represents the coordinate of the current rotation-axis, and you can use the very same expression.

If you want to rotate only one or two coordinates, you can use the normal method of rotation, because then you won't have to calculate a 3x3 transformation matrix. But if you have more points, I recommend the optimized version.

But there are optimizations in this field, but let's first look at ONE way to do this:

#### NORMAL METHOD OF ROTATING A VECTOR WITH THREE GIVEN ANGLES IN 3D:

Assume we want to rotate  $V=(x,y,z)$  around the z-axis with the angle  $a$ , around y with  $b$  and around x with  $c$ .  
The first rotation we do is around the Z-axis:  
 $U=(x,y)$  ( $x,y$  from  $V$ -vector)  $\Rightarrow$   
 $\Rightarrow U'=\text{rot}(U,a)=\text{rot}((x,y),a)=(x',y')$

Then we want to rotate around the Y-axis:  
 $W=(x',z)$  ( $x'$  is from  $U'$  and  $z$  is from  $V$ )  $\Rightarrow$   
 $\Rightarrow W'=\text{rot}(W,b)=\text{rot}((x',z),b)=(x'',z')$

And finally around the X-axis:  
 $T=(y',z')$  ( $y'$  is from  $U'$  and  $z'$  is from  $W'$ )  $\Rightarrow$   
 $\Rightarrow T'=\text{rot}(T,c)=\text{rot}((y',z'),c)=(y'',z'')$

The rotated vector  $V'$  is the coordinate vector  $(x'',y'',z'')$  !

With this method we can extend out rot-command to:

$V'' = \text{rot}(V, \text{angle1}, \text{angle2}, \text{angle3})$  where  $V$  is the original vector!  
(  $V'' = \text{rot}((x,y,z), \text{angle1}, \text{angle2}, \text{angle3})$  )

I hope that didn't look too complicated.  
As I said, there are optimizations of this method. These optimizations can be skipping one rotation of the above ones, or some precalculations.

ORDER is very important. You won't get the same answer if you rotate X,Y,Z with the same angles as before.

#### Optimizations:

=====

For xyz vectors we can write the equations to form the rotations:

let  $c1=\cos(\text{angle1})$ ,  $c2=\cos(\text{angle2})$ ,  $c3=\cos(\text{angle3})$ ,

s1=sin(angle1), s2=sin(angle2), s3=sin(angle3)

(x\*cos(a)+y\*sin(a),x\*sin(a)-y\*cos(a))

x' = x\*c1+y\*s1  
y' = x\*s1-y\*c1

x''= x'\*c2+z\*s2 <- Rotated x-coordinate  
z' = x'\*s2-z\*c2

y''= y'\*c3+z'\*s3 <- Rotated y-coordinate  
z''= y'\*s3-z'\*c3 <- Rotated z-coordinate

which gives:

x''= (x\*c1+y\*s1)\*c2+z\*s2= c2\*c1 \*x + c2\*s1 \*y + s2 \*z  
^^^^^^^^^^^^^=x'            ^^^^^ xx       ^^^^^ xy       ^ z

y''= (x\*s1-y\*c1)\*c3+((x\*c1+y\*s1)\*s2-z\*c2)\*s3=  
c3\*s1 \*x - c3\*c1 \*y + s3\*s2\*c1 \*x + s3\*s2\*s1 \*y - s3\*c2 \*z=  
  
(s3\*s2\*c1+c3\*s1) \*x + (s3\*s2\*s1-c3\*c1) \*y + (-s3\*c2) \*z  
^^^^^^^^^^^^^^^^^ yx       ^^^^^^^^^^^^^^^^^^ yy       ^^^^^^^^^ yz

z''= (x\*s1-y\*c1)\*s3-((x\*c1+y\*s1)\*s2-z\*c2)\*c3=  
s3\*s1 \*x - s3\*c1 \*y - c3\*s2\*c1 \*x - c3\*s2\*s1 \*y + c3\*c2 \*z=  
  
(-c3\*s2\*c1+s3\*s1) \*x + (-c3\*s2\*s1-c3\*c1) \*y + (c3\*c2) \*z  
^^^^^^^^^^^^^^^^^ zx       ^^^^^^^^^^^^^^^^^^ zy       ^^^^^^^^^ zz

Now, look at the pattern of the solutions,  
for x'' we have calculated something times the original (x,y,z),  
the same for y'' and z'', What is the connection?

Say that you rotate many given vectors with three angles that are the same for all vectors, then you get this scheme of multiplications. When you rotated as above you had to use twelve multiplications to do one rotation, but now we precalculate these 'constants' and manage to get down to nine multiplications!

^^^^

FINAL FORMULA FOR ROTATIONS IN THREE DIMENSION WITH THREE ANGLES  
(x,y,z is the original (x,y,z) coordinate.  
c1=cos(angle1), s1=sin(angle1), c2=cos(angle2) and so on...)

If you want to rotate a lot of coordinates with the same angles you first calculate these values:

xx=c2\*c1  
xy=c2\*s1  
xz=s2  
yx=c3\*s1+s3\*s2\*c1  
yy=-c3\*c1+s3\*s2\*s1  
yz=-s3\*c2  
zx=s3\*s1-c3\*s2\*c1;s2\*c1+c3\*s1  
zy=-s3\*c1-c3\*s2\*s1;c3\*c1-s2\*s1  
zz=c3\*c2

Then, for each coordinate, you use the following multiplication to get the rotated coordinates:

$$\begin{aligned}x'' &= xx * x + xy * y + xz * z \\y'' &= yx * x + yy * y + yz * z \\z'' &= zx * x + zy * y + zz * z\end{aligned}$$

So, you only have to calculate the constants once for every new angle, and THEN you use nine multiplications for every point you wish to rotate to get the new set of points.

Look in the end of this text for an example of how this can be implemented in 68000-assembler.

If you wish to skip on angle, you can optimize further. if you want to remove angle3, set c3=1 and all s3=0 and put into your constant-calculation and it will be optimized for you.

What method you want to use depends of course on how much you want to code, but I prefer the optimized version since it's more to be proud of... If you only rotate a few points with the same angles, the first (non-optimized) version might be the choice.

If you want to, you can check that the transformation matrix has a determinant equal to 1.

## 1.7 Polygons!

### 5. Polygons!

=====

The word "polygon" means many corners, which means that it has a lot of points (corners) with lines drawn to.

If you have, for example, 5 points, you can draw lines from point 1 to point 2, from point 2 to point 3, from point 3 to point 4 and from point 4 to point 5.

If you want a CLOSED polygon you also draw a line from point 5 to point 1.

Points:2

```

      .
      .3
1
.
5..4

```

Open polygon of points above:

```

  /|
 / |
/  /
/  /
_ /

```

Closed polygon of points above:



"Filled vectors" is created by drawing polygons, and filling inside. Normally the following algorithm is used:

First you define all "corners" on the polygon as vectors, which allows you to rotate it and draw it in new angles, and then you draw one line from point 1 to point 2, and so on. The last line is from point 5 to point 1. When you're finished you use a BLITTER-FILL operation to fill the area.

You will need a special line drawing routine for drawing these lines so the BLITTER-FILL works, I have an example of a working line-drawing routine in the appendices (K-seka! Just for CJ!). Further theory about what demands there are on the line drawing routine will be discussed later (App. B 2).

There are also other ways to get a filled area (mostly for computers without blitter, or for special purposes on those that have) Information about that will be in later issues.

#### Creating objects from polygons

=====

An object is in this text a three-dimensional thing created with polygons. We don't have to think about what's inside, we just surround a mathematically defined sub-room with polygons.

But what happens to surfaces that is on the other side of the object? and if there are hidden "parts" of the object, what can we do about them?

We begin with a cube, it is easy to imagine, and also the rotation of it. we can see that no part of the cube is over another part of the cube in the viewers eyes. (compare, for example, with a torus, where there are sometimes parts that hides other parts of the same object) Some areas are of course AIMING AWAY FROM THE VIEWER, but we can calculate in what direction the polygon is facing (to or from the viewer)

Always define the polygons in objects in the same direction (clockwise or non-clockwise) in all of the object. imagine that you stand on the OUTSIDE MIDDLE of the plane, and pick all points in a clockwise order. Which one you start with has nothing to do with it, just the order of them.

Pick three points from a plane (point1, point2 and point 3) If all three points are not equal to any of the other points, these points define a plane.

You will then only need three points to define the direction of the plane. Examine the following calculation:

$$c = (x_3 - x_1) * (y_2 - y_1) - (x_2 - x_1) * (y_3 - y_1)$$

(This is AFTER 3d->2d projection, so there's no z-coordinate.)

If you want to know what this does, look in appendix b)

This needs three points, which is the minimum number of coordinates a polygon must be, to not be a line or a point (THINK). This involves two multiplications per plane, but that isn't very much compared to rotation and 3d->2d projection.

But let us study what this equation gives:

If c is negative, the normal vector of the plane which the three points span is heading INTO the viewer ( = The plane is fronting the viewer => plane should be drawn )...

If c is POSITIVE, the normal vector of the plane is heading AWAY from the viewer ( = The plane cannot be seen by the viewer => DON'T draw the plane) ...

But to question 2, what happens if parts of the object covers OTHER parts of the object...

#### Convex and concave objects =====

##### "Definitions"

A convex object has NO parts that covers other parts of the same object, viewed from all angles.

A concave object has parts that covers other parts of the same object, viewed from some angle.

--

For convex objects, this means that you can draw a straight line from every point inside the object to every other point in the object without having no line that passes outside the domain of the object.

If you have a CONVEX object, you can draw ALL lines around the visible planes and then fill with the blitter, because no drawn polygon will ever cover another polygon. With some struggle you can also find ways to omit some lines, since they will be drawn twice.

CONCAVE objects offers some further problems, the easiest way to use CONCAVE objects is to split them into smaller CONVEX objects. This works for all objects, even if you can have some problem doing it.

Of course, you can skip a lot of planes that will be "inside" the concave object.

When you have splitted the object you simply draw each convex object into a TEMPORARY memory area, and treat it like a VECTORBALL (Sorting and stuff), Which should be discussed in later parts of this text.

The Z coordinate can be taken from the middle of all z-values in the object (The sum of all Z's in the object divided by the number of coordinates)

When you're sorting the objects, you can sometimes have problems with parts of the concave object getting in the wrong order since you've picked a point at random from the OUTSIDE of the convex object, which the current object is sharing with another convex object. One way to solve this problem is to take a middle point that is inside the convex object, by adding all the Z-values around the object and dividing by the number of coordinates that you have added. In this case, you should take points from at least two planes in the object.

#### Object optimization

=====

Assume that you have a CONVEX object.

If it is closed, you have almost as few points as you have planes. If you have a list to every coordinate that exist (no points are the same in this list) that for each polygon shows what point you should fetch for this coordinate, you can cut widely on the number of ROTATIONS.

For example:

```
/* A cube */
/* order is important! Here is clockwise */

end_of_plane=0

pointlist
  dc.l pt4,pt3,pt2,pt1,end_of_plane
  dc.l pt5,pt6,pt2,pt1,end_of_plane
  dc.l pt6,pt7,pt3,pt2,end_of_plane
  dc.l pt7,pt8,pt4,pt3,end_of_plane
  dc.l pt8,pt5,pt1,pt4,end_of_plane
  dc.l pt5,pt6,pt7,pt8,end_of_plane
pt1 dc.w -1,-1,-1
pt2 dc.w 1,-1,-1
pt3 dc.w 1,-1,1
pt4 dc.w -1,-1,1
pt5 dc.w -1,1,-1
pt6 dc.w 1,1,-1
pt7 dc.w 1,1,1
pt8 dc.w -1,1,1
```

Now, you only have to rotate the points pt1-pt8, which is eight points. If you had computed four points for each plane, you would have to compute 24 rotations instead.

## 1.8 Planes in three dimensions

### 6. Planes in three dimensions

Lightsourcing

Lightsourcing is a way to find out how much light a plane receives from either a point of light (spherical) or a plane of light (planar). If the colour of the plane represents the light that falls on it, the object will be a bit more realistic.

What we are interested in is the ANGLE of the VECTOR from the NORMAL of the plane to the LIGHTSOURCE (=point of light) (this is for a spherical lightsource, like a lamp or something. If you are interested in planar light, like from the sun, you are interested in the ANGLE between the NORMAL of the plane and the LIGHTSOURCE VECTOR)

We are interested of the COSINE of the given angle.

Anyway, to get the normal of the plane you can pick three points in the polygon, create two vectors of these.

Example:

```
* we pick (x1,y1,z1) and (x2,y2,z2) and (x3,y3,z3)
  we create two vectors V1 and V2:
  V1=(x2-x1,y2-y1,z2-z1)
  V2=(x3-x1,y3-y1,z3-z1)
```

To get the normal of these we take the cross product of them:

$$N = V1 \times V2 = \begin{vmatrix} i & j & k \\ x2-x1 & y2-y1 & z2-z1 \\ x3-x1 & y3-y1 & z3-z1 \end{vmatrix} =$$

$$= ( (y2-y1) * (z3-z1) - (y3-y1) * (z2-z1), -((x2-x1) * (z3-z1) - (x3-x1) * (z2-z1)), (x2-x1) * (y3-y1) - (x3-x1) * (y2-y1) )$$

Now, we have N. We also have the LIGHTSOURCE coordinates (given)

To get COS of the ANGLE between two vectors we can use the scalar product between N and L (=lightsource vector) divided by the length of N and L:

$$\langle N, L \rangle / (||N|| * ||L||) =$$

```
* (n1*l1+n2*l2+n3*l3) / (sqr(n1*n1+n2*n2+n3*n3) * sqr(l1*l1+l2*l2+l3*l3))
|
* (can be (n1*l1+n2*l2+n3*l3)/k if k is a precalculated constant)
```

This number is between -1 and 1 and is cos of the angle between the vectors L and N. the SQUARE ROOTS take much time, but

if you keep the object intact (do only rotations/translatins etc.) and always pick the same points in the object, then  $||N||$  is intact and can be precalculated.

If you make sure the length of L is always 1, you won't have to devide by this, which saves many cycles.

The number will, as said, be between -1 and 1. You may have to multiply the number with something before dividing so that you have a larger range to pick colours from. If the number is negative, set it to zero.

The number can be NEGATIVE when it should be POSITIVE, this is because you took the points in the wrong order, but you only have to negate the result instead.

If you didn't understand a thing of this, look on the formulas with a '\*' in the border. n1 means the x-coordinate of N, n2 the y-coordinate and so on, and the same thing with L.

## 1.9 Special techniques - Sorting Algorithms

Special techniques - Sorting Algorithms

=====

When you come to sorting, most books begin with "Bubble-sorting" Bubble sorting is enourmously slow, and is described here only for explaining terms. But I don't advise you to code routines that uses this method since it's SLOOOOOOOOOOOOW.... A better way is to use Insert Sorting (which, in fact, is sorting, Acro!) or Quick Sorting or whatever you can find in old books (you have them, I'm sure!) about basic/pascal/c/ or whatever (and even a few assembler books!!!) contains different methods of sorting. Just make sure you don't use BUBBLE SORTING!!!!

Method 1) Bubble sorting

-----

Assume that you have a list of VALUES and WEIGHTS. The heaviest weights must fall to the bottom, and bringing the VALUES with it. The values in this case can be the 2d->3d projected x and y coordinates plus bob information. The Weights can be the Z coordinates before projection.

Begin with the first two elements, check what element is the HEAVIEST, and if it is ABOVE the lighter element, move all information connected with the WEIGHT and the WEIGHT to the place where the light element was, and put the light data where the heavy was. (This operation is called a 'swap' operation)

Step down ONE element and check element 2 and 3..

step further until you're at the bottom of the list.

The first round won't fix the sorting, you will have to go round the list THE SAME NUMBER OF TIMES AS YOU HAVE OBJECTS minus one!!!!

The many comparisons vote for a faster technique...

Algorithm:

```
1> FOR outer loop=1 TO Items-1
1.1> FOR inner loop=1 TO Items-1
1.1.1> IF Item(inner loop)>Item(inner loop+1)
1.1.1.1> Swap Item(inner loop),Item(inner loop+1)
```

(Items is the number of entries to sort, Item() is the weight of the current item)

Method 2) Insert sorting

-----

Assume the same VALUES and WEIGHTS as above.  
To do this, you have to select a wordlength for the sorting-table (checklist) and a size of the checklist.

The wordlength depends on the number of entries you have, and the size of every entry. Normally, it's convenient to use WORDS. The size of the checklist is the range of Z-values to sort, or transformed Z-values. If you, for example, know that your Z-values lies within 512-1023 you can first decrease every z-value by 512, and then lsr' it once, which will give you a checklist size of 256 words. You will also need a second buffer to put your sorted data in, this 2ndBUF will be like a copy of the original list but with the entries sorted.

For this method I only present an algorithm, it's easier to see how it works from that than from some strange text.

checklist(x) is the x'th word in the checklist.

Algorithm:

```
1> CLEAR the checklist (set all words=0)
2> TRANSFORM all weights if necessary.
3> FOR L=0 TO number of objects
3.1> ADD ENTRYSIZE TO checklist(transformed weight)
4> FOR L=0 TO checklist size-1
4.1> ADD checklist(L),checklist(L+1)
5> FOR L=0 TO number of objects
5.1> PUT ENTRY at 2ndBUF(checklist(transformed weight))
5.2> ADD ENTRYSIZE TO checklist(transformed weighth)
```

Now, your data is nicely sorted in the list 2ndBUF, the

---

original list is left as it was (except for Z-transformation).  
 (ENTRYSIZE is the size of the entry, so if you have x,y,z coordinates  
 in words, your size is 3 words=6 bytes.)  
 Also try to think a little about what you get when you  
 transform. The subtraction is useful since it minimizes the  
 loops, but lsr-ing the weights take time and makes the  
 result worse. Of course you don't have to scan the list every time,  
 just make sure that you know what the lowest possible and the  
 highest possible weight is.

### Method 3) the Quick-Sort

-----

This is another kind of sorting, and here it is most efficient  
 to use pointers, so that each entry have a pointer to NEXT entry.

you can one entry like this:

```
NEXT OFFSET=word
x,y,z=coordinates.
```

(offsets are from sortlist start address...)

To access this routine you will have to give a FIRST entry  
 and number of entries. In the original execution, first entry  
 is of course 0 (=first entry) and the number of entries is  
 of course the total number of entries.

You must set all previous/next pointers to link a chain.

Quicksort is recursive, which means that you will have to  
 call the routine from within itself. This is not at  
 all complicated, you just have to put some of your  
 old variables on the stack for safe-keeping.

What it does is this:

```
+> The first entry in the list is the PIVOT ENTRY.
| For each other ENTRY, we put it either BEFORE or AFTER
| the PIVOT. If it is lighter than the PIVOT we put it BEFORE,
| otherwise we put it AFTER.
| Now we have two new lists, All entries BEFORE the PIVOT,
| and all entries AFTER the PIVOT (but not the pivot itself,
| which is already sorted).
| Now we quicksort All entries BEFORE the pivot separately
+< and then we quicksort all entries AFTER the pivot.
  (We do this by calling on the routine we're already in)
  This may cause problems with the stack if there's too
  many things to sort.
```

The recursion loop is broken when there's <=1 entry  
 to sort.

Contrary to some peoples belief, you don't need any extra  
 lists to solve this.

Algorithm:

Inparameters: (PivotEntry=first element of list

---

```

                List size=size of current list)
1> If list size <= 1 then exit
2> PivotWeight=Weight(PivotEntry)
3> for l=2nd Entry to list size-1
3.1> if weight(l) > PivotWeight
3.1.1> insert entry in list 1
3.2> ELSE
3.2.1> insert entry in list 2
4> Sort list 1 (bsr quicksort(first entry list 1, size of list 1))
5> Sort list 1 (bsr quicksort(first entry list 2, size of list 2))
6> Link list 1 -> PivotEntry -> list 2

```

(PivotEntry = FirstEntry, it don't have to be like this, but I prefer it since I find it easier.)

## 1.10 vectorballs

Special techniques - Vector Balls

=====

Vector balls are simple. Just calculate where the balls are (with rotations, translations or whatever it can be). Sometimes you also calculate the size of the ball and so on.

You don't have to have balls. You can have the Convex parts of an concave filled object, or you can have images of whatever you like. In three dimensions you will have the problem with images (balls or whatever) that should be in front of others because it is further away from you. Here is where SORTING comes in. If you BEGIN blitting the image that is most distant to you, and step closer for each object, you get a 3d-looking screen. The closest image will be the closest.

Normally, you start with clearing the screen you're not displaying at the moment (Parts of it anyway. A person in Silents only cleared every second line...)

Then (while the blitter is working) you start rotating, sorting and preparing to finally bob the images out

and when you've checked that the blitter is finished, you start bobbing out all images, and when the frame is displayed, you swap screens so you display your finished screen the next frame.

## 1.11 sources

Appendix A: Example sources.

- 1 Optimised rotation matrix calculation
- 2 A line draw routine for filled vectors
- 3 Quicksort in 68000 assembler
- 4 Insert Sort in 68020 assembler

## 1.12 Optimised rotation matrix calculation

A 1. An example of an optimized rotation matrix calculation

```
=====
* For this routine, you must have a sinus table of 1024 values,
* and three words with angles and a place (9 words) to store
* the transformation matrix.
*
*      .
* / ( | ( ) | \ / ' ( | )
* / ) | ( | \ | / \      | )
```

Calculate\_Constants

```

lea    Coses_Sines(pc),a0
lea    Angles(pc),a2
lea    Sintab(pc),a1

move.w (a2),d0
and.w  #$7fe,d0
move.w (a1,d0.w), (a0)
add.w  #$200,d0
and.w  #$7fe,d0
move.w (a1,d0.w), 2(a0)
move.w 2(a2),d0
and.w  #$7fe,d0
move.w (a1,d0.w), 4(a0)
add.w  #$200,d0
and.w  #$7fe,d0
move.w (a1,d0.w), 6(a0)
move.w 4(a2),d0
and.w  #$7fe,d0
move.w (a1,d0.w), 8(a0)
add.w  #$200,d0
and.w  #$7fe,d0
move.w (a1,d0.w), 10(a0)

;xx=c2*c1
;xy=c2*s1
;xz=s2
;yx=c3*s1+s3*s2*c1
;yy=-c3*c1+s3*s2*s1
;yz=-s3*c2
;zx=s3*s1-c3*s2*c1;s2*c1+c3*s1
;zy=-s3*c1-c3*s2*s1;c3*c1-s2*s1
;zz=c3*c2

lea    Constants(pc),a1
move.w 6(a0),d0
move.w (a0),d1
```

```

move.w  d1,d2
muls    d0,d1
asr.l   #8,d1
move.w  2(a0),d3
muls    d3,d0
asr.l   #8,d0
move.w  d0,(a1)
;neg.w  d1
move.w  d1,2(a1)
move.w  4(a0),4(a1)
move.w  8(a0),d4
move.w  d4,d6
muls    4(a0),d4
asr.l   #8,d4
move.w  d4,d5
muls    d2,d5
muls    10(a0),d2
muls    d3,d4
muls    10(a0),d3
add.l   d4,d2
sub.l   d5,d3
asr.l   #8,d2
asr.l   #8,d3
move.w  d2,6(a1)
neg.w   d3
move.w  d3,8(a1)
muls    6(a0),d6
asr.l   #8,d6
neg.w   d6
move.w  d6,10(a1)
move.w  10(a0),d0
move.w  d0,d4
muls    4(a0),d0
asr.l   #8,d0
move.w  d0,d1
move.w  8(a0),d2
move.w  d2,d3
muls    (a0),d0
muls    2(a0),d1
muls    (a0),d2
muls    2(a0),d3
sub.l   d1,d2
asr.l   #8,d2
move.w  d2,12(a1)
add.l   d0,d3
asr.l   #8,d3
neg.w   d3
move.w  d3,14(a1)
muls    6(a0),d4
asr.l   #8,d4
move.w  d4,16(a1)

rts

```

```

Coses_Sines  dc.w  0,0,0,0,0,0
Angles       dc.w  0,0,0
Constants    dc.w  0,0,0,0,0,0,0,0,0,0

```

;Sintab is a table of 1024 sinus values with a radius of 256  
;that I have further down my code...

## 1.13 A line draw routine for filled vectors

A 2. A line drawing routine for filled vectors in assembler:

```
=====
* written for kuma-seka ages ago, works fine and
* can be optimized for special cases...
* the line is (x0,y0)-(x1,y1) = (d0,d1)-(d2,d3) ...
* Remember that you must have DFF000 in a6 and
* The screen start address in a0.
* Only a1-a7 and d7 is left unchanged.
*
* / ( | ( ) | \ / ' ( | )
* / ) | ( | \ | / \   | )

Screen_widht=40 ;40 bytes wide screen...
fill_lines:      ;(a6=$dff000, a0=start of bitplane to draw in)

        cmp.w    d1,d3
        beq.s    noline
        ble.s    lin1
        exg     d1,d3
        exg     d0,d2
lin1:    sub.w    d2,d0
        move.w   d2,d5
        asr.w   #3,d2
        ext.l   d2
        sub.w   d3,d1
        muls    #Screen_Widht,d3          ;can be optimized here..
        add.l   d2,d3
        add.l   d3,a0
        and.w   #$f,d5
        move.w  d5,d2
        eor.b   #$f,d5
        ror.w   #4,d2
        or.w    #$0b4a,d2
        swap   d2
        tst.w   d0
        bmi.s   lin2
        cmp.w   d0,d1
        ble.s   lin3
        move.w  #$41,d2
        exg    d1,d0
        bra.s   lin6
lin3:    move.w  #$51,d2
        bra.s   lin6
lin2:    neg.w   d0
        cmp.w   d0,d1
        ble.s   lin4
        move.w  #$49,d2
```

```

        exg      d1,d0
        bra.s    lin6
lin4:   move.w   #$55,d2
lin6:   asl.w    #1,d1
        move.w   d1,d4
        move.w   d1,d3
        sub.w    d0,d3
        ble.s    lin5
        and.w    #$ffbf,d2
lin5:   move.w   d3,d1
        sub.w    d0,d3
        or.w     #2,d2
        lsl.w    #6,d0
        add.w    #$42,d0
bltwt:  btst     #6,2(a6)
        bne.s    bltwt
        bchg     d5,(a0)
        move.l   d2,$40(a6)
        move.l   #-1,$44(a6)
        move.l   a0,$48(a6)
        move.w   d1,$52(a6)
        move.l   a0,$54(a6)
        move.w   #Screen_Widht,$60(a6)    ;width
        move.w   d4,$62(a6)
        move.w   d3,$64(a6)
        move.w   #Screen_Widht,$66(a6)    ;width
        move.l   #-$8000,$72(a6)
        move.w   d0,$58(a6)
noline: rts

```

## 1.14 Quicksort in 68000 assembler

A 3. The quicksort in 68000 assembler

=====

```

* Sorts a list that looks like:
* Next entry offset.w, (x,y,z).w.
* all offsets must be set except for first entry's previous offset
* and the last entry's next offset.
* Offsets are FROM FIRST ADDRESS of sorting list
* a5=first address of sorting list!
*
*   _ .
* / ( | ( ) | \ / ' ( | )
* / ) | ( | \ | / \   | )

```

WghtOffs=6

NextOffs=0

```

QuickSort      ;(a5=start of sortlist,
                ; d0=0 (pointer to first entry, first time=0)
                ; dl=number of entries)

```

```

        cmp.w    #1,d1
        ble.s   .NothingToSort           ;don't sort if <=1 entries
        moveq   #0,d4                     ;size list 1
        moveq   #0,d5                     ;size list 2
        move.w  d0,d6                     ;first Nentry=d0

        move.w  WghtOffs(a5,d0.w),d2      ;d2=Pivot weight
        move.w  NextOffs(a5,d0.w),d3     ;d3=2nd entry
        subq.w  #2,d1                     ;Dbf-loop+skip first

.Permute
        cmp.w   WghtOffs(a5,d3.w),d2     ;entry weight<pivot weight?
        ble.s   .Lower

        move.w  d6,NextOffs(a5,d3.w)     ;Insert BEFORE Nentry
        addq.w  #1,d4                     ;increase size of list 1
        move.w  d3,d6                     ;Set new Nentry

        bra.s   .Done                     ;Continue the loop...

.Lower
        move.w  NextOffs(a5,d0.w),NextOffs(a5,d3.w)
        move.w  d3,NextOffs(a5,d0.w)     ;insert AFTER first entry
        addq.w  #1,d5                     ;size of list 2

.Done
        move.w  NextOffs(a5,d3.w),d3     ;Get next entry
        dbf     d1,.permute

        move.w  d0,-(a7)                  ;save Fentry..

        move.w  NextOffs(a5,d0.w),d0     ;Sort at entry after first
        move.w  d5,d1                     ;Size of list 2

        movem.w d4/d6,-(a7)               ;Save important registers
        bsr     QuickSort                 ;and sort list 2
        movem.w (a7)+,d4/d6               ;d1 is now First Entry...
        move.w  (a7)+,d1

        move.w  d0,NextOffs(a5,d1.w)     ;Put first entry of
                                           ;list 2 after Fentry...
        move.w  d6,d0                     ;Sort at Nentry
        move.w  d4,d1                     ;size of list 1

        bsr     QuickSort                 ;no important registers
                                           ;left...

.NothingToSort
        ;Now the offset to the first entry is in d0!
        ;to get the other values in the correct order
        ;just go down the list (using nextoffs.)
        ;First object is the heaviest...

        rts

```

## 1.15 Insert Sort in 68020 assembler

A 4. The Insert Sort in 68020 assembler:

=====

```

* This isn't exactly as the algorithm described earlier,
* it begins with creating a list and then stores the ADDRESSES of the
* sorted data in 2ndBUF instead...
* This sorts all lists, just specify offset to weight (word) and
* size of each entry. You don't need any pre-formatting.
* note that you HAVE TO change a line if you want this to work
* on 68000.. I've got a scaled index at one point. replace it
* with the lines after the semicolon.
*
*      —      .
* / ( | ( ) | \ / ' ( | )
* / ) | ( | \ | / \      | )

```

```

WghtOffs=4
EntrySize=6

```

```

InsertSort

```

```

; a5=start of data
; a4=start of checklist
; a3=start of 2ndBUF
; d0 is lowest value of entries
; d1 is highest value
; d2 is number of entries

movem.l a4/a5,-(a7)

sub.w   d0,d1           ;max size of checklist this sort.
subq.w  #1,d2
subq.w  #1,d1           ;Dbf-loops...

move.w  d1,d3           ;clear used entries
.ClearChecklist clr.w  (a4)+
dbf     d3,.ClearCheckList

move.w  d2,d3           ;transform...
.Transform sub.w  d0,WghtOffs(a5)
addq.w  #EntrySize,a5
dbf     d3,.Transform

movem.l (a7),a4/a5

move.w  d2,d3           ;Insert next line instead for
.AddisList move.w  WghtOffs(a5),d0 ;68000 compatibility...
addq.w  #4,(a5,d0.w*2) ;add.w d0,d0 addq.w #4,(a5,d0.w)
addq.w  #EntrySize,a5
dbf     d3,.AddisList

moveq   #-4,d0          ; #-lwdsize
.GetMemPos add.w  d0,(a4)
move.w  (a4)+,d0
dbf     d1,.GetMemPos

movem.l (a7)+,a4/a5
.PutNewList move.w  WghtOffs(a5),d0
move.w  (a4,d0.w),d0
move.l  a5,(a3,d0.w)
addq.w  #EntrySize,a5

```

```

dbf      d2,.PutNewList

;In this case you have a list of ADDRESSES to
;each object. I made it this way to
;make it more flexible (you maybe have more
;data in each entry than me?).

rts

```

## 1.16 Further Information

### Appendix B: Further Information

#### B 1: 'Proof' of hidden-plane elimination equation

```
=====
```

I presented the following equation:  
 $c=(x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)$   
 as a calculation of the normal vector of the plane  
 that the polygon in question spanned.

We had three points:  
 $p1(x1,y1)$   
 $p2(x2,y2)$   
 $p3(x3,y3)$

If we select  $p1$  as base-point, we can construct the following  
 vectors of the rest of the points:

$V1=(x3-x1,y3-y1,p)$   
 $V2=(x2-x1,y2-y1,q)$

Where  $p$  and  $q$  in the  $z$  value denotes that we are not interested in  
 this value, but we must take it in our calculations anyway.  
 (These values are NOT the same as the original  $z$ -values  
 after the 2d->3d projection)

Now, we can get the normal vector of the plane that these vectors  
 span by a simple cross-product:

$V1 \times V2 =$

$$\begin{vmatrix}
 i & j & k \\
 (x3-x1) & (y2-y1) & p \\
 (x2-x1) & (y3-y1) & q
 \end{vmatrix}$$

(if  $i=(1,0,0)$ ,  $j=(0,1,0)$ ,  $k=(0,0,1)$ )  
 ( $p$  and  $q$  are non-important)

But we are only interested in the  $Z$ -direction of the  
 result-vector of this operation, which is the same as  
 getting only the  $Z$ -coordinate out of the cross-product:

$Z \text{ of } (V1 \times V2) = (x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)$

Now if  $Z$  is positive, this means that the resultant vector  
 is aiming INTO the screen (positive  $z$ -values)

QED /Asterix

B 2. How to make a fill line out of the blitters line-drawing

=====

You can't use the blitter line-drawing as it is and draw lines around a polygon without a few special changes.

To make a fill-line routine out of a normal-lineroutine:

First, make sure it draws lines as it should, many line-drawers I've seen draws lines to wrong points  
 Make sure you use Exclusive or- instead of or-minterm  
 Always draw lines DOWNWARDS. (or UPWARDS, if you prefer that)  
 Before drawing the line and before blit-check, eor the FIRST POINT ON THE SCREEN THAT THE LINE WILL PASS.  
 Use fill-type line mode.

B 3: An alternate approach to rotations in 3-space by M. Vissers

=====

/\* This is a text supplied by Michael Vissers, and was a little longer. I removed the part about projection from 3d->2d, which was identical to parts of my other text in chapter 3. If you know some basic linear algebra, this text might be easier to melt than the longer version discussed in chapter 4. If you didn't get how you were supposed to use the result in chapter 4 then try this part instead. \*/

[ ] All you have to do is using these 3D matrices :

(A/B/G are Alpha,Beta and Gamma.) /\* A,B,C = Angles of rotation \*/

	cosA	-sinA	0			cosB	0	-sinB			1	0	0	
	sinA	cosA	0			0	1	0			0	cosG	-sinG	
	0	0	1			sinB	0	cosB			0	sinG	cosG	

These are the rotation matrices around the x,y and z axis'. If you would use these you'll get 12 muls'. 4 four for each axis. But, if you multiply these three matrices with eachother you'll get only 9 muls'. Why 9 ???  
 Simple : after multiplying you'll get a 3x3 matrice, and 3\*3=9 !

It doesn't matter if you do not know how to multiply these matrices. It's not important here so I'll just give the 3x3 matrice after multiplying :

(c = cos, s = sin, A/B/G are Alpha,Beta and Gamma.)

	ca*cb	-cb*sA	sB	
	cG*sA-sB*cA*sG	cA*cG+sG*sA*sB	cB*sG	
	-sG*sA-sB*cA*cG	-cA*sG+sA*sB*cG	cG*cB	

I hope I typed everything without errors :) Ok, how can we make some coordinates using this matrice. Again, the trick is all in multiplying. To get the new (x,y,x) we need the original points and multiply these with the matrice. I'll work with a simplified matrice. (e.g. H = cA\*cB etc...)

	x	y	z	( <= original coordinates)
New X =	H	I	J	
New Y =	K	L	M	
New Z =	N	O	P	

So...

```

New X = x * H + y * I + z * J
New Y = x * K + y * L + z * M
New Z = x * N + y * O + z * P

```

Ha ! That's a lot more than 9 muls'. Well, actually not. To use the matrice you'll have to precalculate the matrice.

Always rotate with your original points and store them somewhere else. Just change the angles to the sintable to rotate the shape. If you rotate the points rotated the previous frame you will lose all detail until nothing is left.

So, every frame looks like this :

- pre calculate new matrice with given angles.
- Calculate points with stored matrice.

[ ]

The resulting points are relative to (0,0). So they can be negative to. Just use a add to get it in the middle of the screen.

NOTE: Always use muls,divs,asl,asr etc. Data can be both positive and negative. Also, set the original coordinates as big as possible, and after rotating divide them again. This will improve the quality of the movement.

(Michael Vissers)

B 4: A little math hint for more accurate vector calculations

When doing a muls with a value and then downshifting the value, use and 'addx' to get roundoff error instead of truncated error, for example:

```

moveq #0,d7
DoMtxMul
:
muls (a0),d0 ;Do a muls with a sin value *256
asr.l #8,d0
addx.w d7,d0 ;roundoff < trunc
:

```

When you do a 'asr' the last outshifted bit goes to the x-flag. if you use an addx with source=0 => dest=dest+'x-flag'. This halves the error, and makes complicated vector objects less 'hacky'.

```

/ )
( ( / ( 1 ( ) 1 \ / ' ( 1 )
) ) / ) 1 ( 1 \ 1 / \ 1 )
( /

```