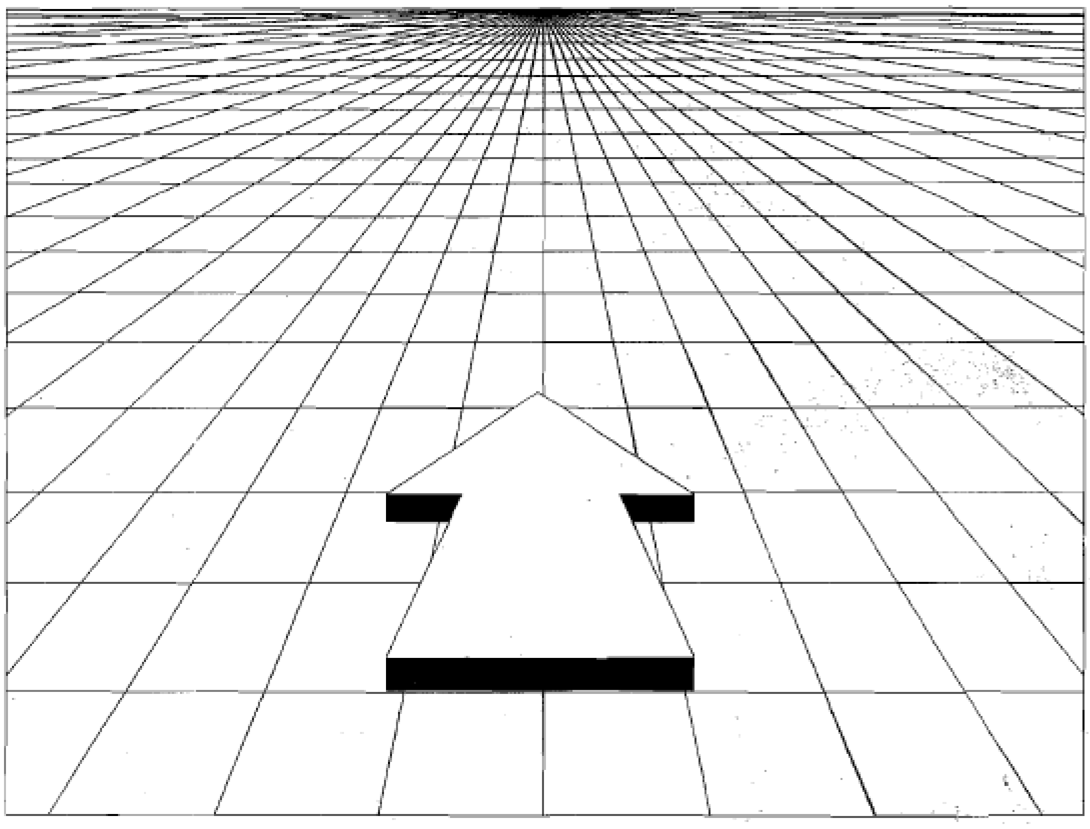
Vector Graphics on the Amiga



Af: Finn Arne Gangstød
Arnfinn Fomess & Carsten Nordenhof
Copyright Dataskolen Aps

DATASKOLEN
programmers choice

DATASKOLEN ApS. Postboks 62, Nordengen 18, DK-2980 Kokkedal, ☎ +45 49 18 00 77

Translated by Paul Klasmann using DeepL and LibreOffice (September 2022).

The original document can be found here: http://palbo.dk/dataskolen/maskinprog/

Table of Contents

Introduction

Welcome to this course in Vector Programming on Amiga.

Prerequisites

In this course 2 and 3 dimensional graphics and the so called vectors are discussed. This requires some knowledge of the mathematical concepts SINUS and COSINUS. We are not able to explain everything here - we leave that to the schools - but the most basic will be explained during the review of this topic.

Assembler

A number of the programs are written for both the Devpac2 and the K-Seka assembler. The programs written for K-Seka are in a directory of their own.

The Disk

All sample programs are on the accompanying floppy disk (see list on page 49). The explanations for the examples are in the source code.

There are also some "readme" files, which you should of course read.

Figures

At the back of the folder you will also find all the illustrations (figures). They are numbered and referred to in the course.

Assignments

You can test your knowledge by solving the exercises, on pages 50-52.

When you wish to have the exercises corrected, send the solutions together with a stamped reply envelope to Dataskolen Aps, Postboks 62, 2980 Kokkedal, Sweden, and you will have them corrected.

You are welcome to contact Dataskolen for free teacher guidance. Please send your questions together with a stamped reply envelope and we will help you. Unfortunately, we are not able to help you by phone.

We hope you enjoy the course. Let's get started:

Coordinate system

When making vector graphics on Amiga, there are a few things you need to get right. Let's therefore start by looking at how a 3-dimensional coordinate system is structured.

If you have a sheet of paper in front of you and you put a point in the middle of it, you can use this point as a reference point when you need to describe where any other points are. This point is called ORIGIN.

You use X and Y coordinates (an imaginary line horizontally on the sheet and a corresponding vertical one) to describe where other points are in relation to the reference point, ORIGIN.

For example, if you put a point 5 cm further up the sheet, you could say that the Y coordinate is 5.

If a point is below ORIGIN, the Y coordinate of the point will have negative values. A

point 5 cm below ORIGIN will have the coordinate value y = -5.

Just as the distance up or down from ORIGIN is called the Y coordinate, the distance to right and left is the X coordinate. A point 4 cm to the right of ORIGIN will thus have

X coordinate +4, and a point 4 cm to the left of ORIGIN will have X coordinate equal to -4.

Also note that the unit of measurement (e.g. cm) is not taken into account when writing the coordinate values, and that they are always mentioned in alphabetical order as follows: (X,Y).

This means that a point three centimetres to the right and four centimetres below ORIGIN is written (3,-4).

The point ORIGIN has no distance from itself (of course) and can therefore also be named (0,0).

You may wonder why we use centimetres and not millimetres or another measure of distance? Actually, you can use any unit of measurement, and because we scale for both the X and Y values, the unit of measurement is not necessary in this context.

The line that goes vertically up/down from ORIGIN is called the Y axis. Similarly, the line that goes horizontally to the right and left from ORIGIN is called the X axis.

In figure 1A you can see the X-axis and the Y-axis. Origin is the place where the two axes (lines) cross each other. The arrow on the right has its tip at a point 2 up the y-axis and 3 off the x-axis, or put another way in (3,2).

When you use only X and Y coordinates (as you would on a flat piece of paper), you have always a flat or so-called 2-dimensional coordinate system.

If you now look at figure 2, you have there an example of such a coordinate system. You have a point called P with coordinates XO and YO. The terms XO and YO stand for two numbers, which are the distance from ORIGIN horizontally (horizontally) and vertically (vertically), respectively.

The world we walk around in everyday is 3-dimensional, you could say it has depth or perspective. Therefore, when making graphics on the Amiga, it would be nice if we could get some perspective in the drawings.

If you stand in front of a mirror, it's relatively easy to imagine how you can get depth in a flat image. The height is easy enough, that's the Y axis; the width is also to deal with, that's the X axis; then you just need the distance "into" the mirror.

Now, if you imagine a line perpendicular to the mirror, so that it goes straight in and straight out from the mirror through the point ORIGIN, then we have a line from which to measure the third dimension, on an otherwise flat piece of glass. This line is called the Z axis and the measurements are called the Z coordinates. See figure 1B.

So even though the screen of a monitor is flat and 2-dimensional, you can and should use 3 dimensions in your calculations when making vector graphics.

Vector

A vector has a specific direction and a specific length, but not a specific position. In Figure 1A, the vector (3,2) is located at two different places in the coordinate system.

When you have a triangle, you can use the coordinates of the vertices to calculate two vectors or line segments.

If you look at Figure 3, vector 1 becomes equal to (X3-X1, Y3-Y1, Z3-Z1), and vector 2 becomes equal to (X2-X1, Y2-Y1, Z3-Z1).

A vector is a line segment. It is defined by three numbers or coordinates, and the three numbers represent how far the line goes in the X-direction, the Y-direction and the Z-direction

If you take the example of the sheet of paper and want to make a 3-dimensional system out of it, imagine a point hovering over the sheet. The distance above the sheet is then called the Z coordinate.

The point has a negative Z coordinate when it is closer to us than ORIGIN, and a positive Z coordinate when it is further away from us than ORIGIN (i.e. "inside the mirror" or "under the paper").

Now, when you need to describe an object (such as a cube), you give the coordinates of the vertices and then draw lines between some of the vertices to show the outline of the cube.

Radians

If, in addition to "just" being able to draw an object, you also want to be able to rotate it, you will need to use a mathematical formula to find out the new coordinates of the vertices after you have turned the object.

The first thing you need to look at when rotating an object in this way is how to describe a rotation.

Most people probably know how an angle is described or measured in degrees (°), but in mathematics it is not common to use degrees - instead, so-called RADIANS are used.

It is easiest to explain what a RADIAN is with an example:

Imagine that you have a clock of the "old-fashioned" type with hands. You place a point "three o'clock" and move it up to "twelve o'clock" along the edge of the dial, that is: without changing the distance to the centre. Now you have rotated this point 90 degrees along the edge of the dial.

And here comes the important part: The radius is the ratio of the LENGTH that you have moved the point and the DISTANCE into the center (which is the radius of the circle).

The circumference of the circle can be calculated as 2*π*radius (π, as you know, is about 3.141592654.... or 3.14 for short), so a quarter (90 degrees) is (2*π*radius) divided by 4.

The angle measured in RADIANS becomes (2*π*radius)/(4*radius)= π/2.

The number describing the angle has no designation such as metre, kilogram, degree or similar. The angle corresponding to 90° is simply written 1.571 (π/2).

In some cases, the letters RAD are added after the number to indicate that the angle is expressed in RADIANS, e.g. 1.571 RAD (which corresponds to 90 degrees).

This formula converts degrees to RAD:	RAD = (2π * number of degrees)/360
and this converts RAD to degrees:	Degrees = (360 * RAD)/(2 * π)

Sine and Cosine

When calculating a rotation, you also need to use some functions called SINE and COSINE. And what are these?

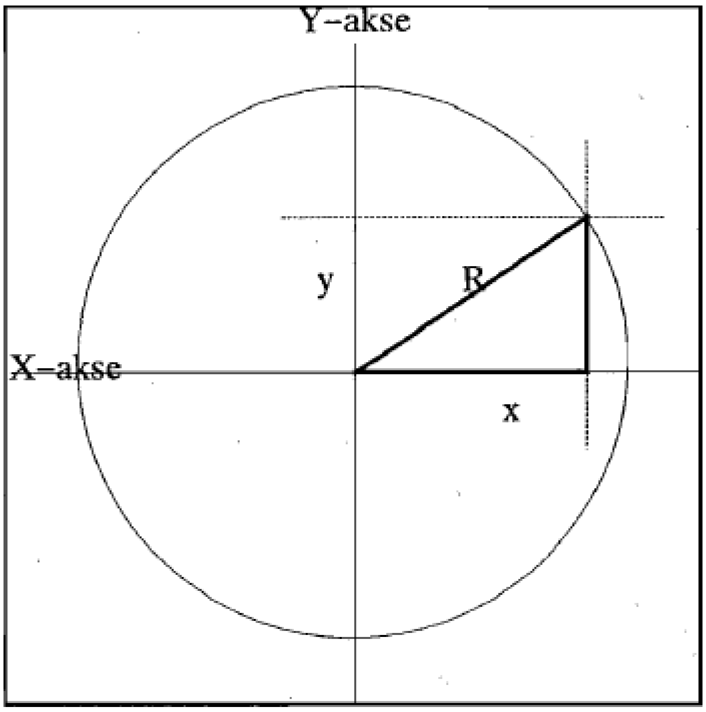
Imagine a point at a certain distance from ORIGIN, and call the distance from ORIGIN to this point R. SINE is then equal to the Y coordinate divided by R.

So the SINE of an angle is the ratio of the Y coordinate to the radius of the arc.

It is measured from the position "three o'clock" and in a counter-clockwise direction.

When the point is in the direction "three o'clock" (or nine o'clock) the Y coordinate becomes zero and thus the SINE value also becomes zero. For O/R (zero divided by R) will always be 0, regardless of the magnitude of R

Thus: SINE to an angle of 0 RAD (that's 0°) or π RAD (180°) is always zero.



SINUS og COSINUS

If you take the SINE for an angle in the direction "two o'clock" (π/6 RAD) you will get a SINE value of 0.5. So a point in the direction "two o'clock" has a Y coordinate which is half the length to the centre.

SINE for an angle of π/2 RAD (90°) is 1, because y and R are then equal.

The COSINE function is exactly the inverse of the SINE function so here the X coordinate is divided by the Radius. Therefore COSINE to 0 and π RAD is equal to 1 and COSINE to π/2 RAD is equal to 0.

SINE and COSINE are abbreviated to SIN and COS respectively.

If you look at Figure 2 again, you will see that SIN(a) (the value you get from the SINE function for the angle "a") is equal to Y0 divided by R, and that COS(a) is equal to X0 divided by R. The coordinates cannot be greater than the distance to the center, so the values you get from SIN() and COS() must be less than or equal to 1.

When the coordinates are negative, you get negative values out of SIN() and COS(). The radius is considered ONE unit of length, regardless of whether it is 0.5 mm or "fifty" light years long.

Rotation

A rotation is (in short) telling the Amiga which coordinates are associated with a certain figure and which angle (how many degrees) the figure should be rotated. The new coordinates are then calculated, the screen is updated and the rotation is complete. Sounds fairly simple doesn't it?

Let's start by looking at how points in a two-dimensional system can be rotated around the centre (ORIGIN).

Let's say you have a plane or surface in a system with only X and Y coordinates (so no depth). When points need to be moved around a centre or a point in this plane, you can use slightly different formulas depending on what you want to achieve. You can have a clockwise or anticlockwise rotation, and you can flip the figure horizontally or vertically. Try looking at this formula:

X1=COS(a0)*X0 + SIN(a0)*Y0

Y1=COS(a0)*Y0 - SIN(a0)*X0

X1 and Y1 are the new coordinates that the point gets after it is rotated around the center. The angle the point must be rotated is called a0, and the coordinates the point has before rotating it around are called X0 and Y0, respectively (Figure 4). This formula gives a counterclockwise rotation when the angle a0 (which you specify) is positive. We say that it has positive orbital direction.

For example, a point at the "two o'clock" position, when you have an angle greater than zero, is rotated counterclockwise. If the angle is equal to π/3 RAD, the point is put in the direction of 12 o'clock, if the angle is π/2 RAD, the point is put in the direction of 11 o'clock, and so on.

Mirror reversal

Mirror turning around one of the axes is achieved by changing the sign. The formula for a flip around the X axis is obtained by adding a minus sign to the first line so that the formula looks like this:

X1= - (COS(a0)*X0 + SIN(a0)*Y0)

Y1= COS(a0)*Y0 - SIN(a0)*X0

When you flip a square, you do so in a clockwise direction. The points are now rotated an angle a0 Clockwise.

That's easy to understand, but what use is the formula when you need to calculate the three dimensions?

When calculating with three dimensions, you obviously have three different planes or surfaces a point can be rotated in. First, you have the XY-plane, which is used in two-dimensional calculations.

If you imagine a clock face standing upright and looking straight ahead, this becomes the XY plane.

Then you have the YZ plane, which goes inward/outward and upward/downward. This corresponds to a dial that is upright and viewed from the side.

The last plane is the XZ plane, and it becomes like a clock face lying flat down and viewed from the side.

In the formula shown above, you can put in values for two coordinates at a time. For calculations for three planes, simply use the formula three times.

First you calculate X1 and Y1 with the formula, then you use the Y value you got out along with the original Z coordinate to calculate the coordinates after turning in the YZ plane:

Z1=COS(a1)*Z0 + SIN(a1)*Y1

Y2=COS(a1)*Y1- SIN(a1)*Z0

Now you have the final value for Y, but you need one more calculation to get the final values for X and Z. This calculation gives the rotation in the XZ plane:

Z2=COS(a2)*Z1 + SIN(a2)*X1

X2=COS(a2)*X1 - SIN(a2)*Z1

The rotation is determined by one angle for each of the planes. Here they are named a0, a1 and a2. The new position of the point is given by the numbers X2, Y2 and Z2. Now you have rotated the points in three dimensions, but the problems with 3-dimensional graphics are not over yet!

When you draw the object on the screen, you can't use the Z coordinate because the screen is flat. You then do it in such a way that you only use the X and Y coordinates.

To make everything look natural, the parts of the object that are far away (and therefore have a large Z coordinate) need to look smaller (we need to have a perspective of the object). This can be achieved by multiplying the X and Y coordinates by a number, which gets smaller the larger Z gets. Later in the course there is a routine "rotate.s" that rotates an object about all 3 axes.

Polygon routine

When you need to draw filled vectors, you need a routine that can draw surfaces in different colours. Such a routine is often called a POLYGON routine (polygon = polygon). There are built-in routines in the Amiga which take care of this, but for maximum speed it may be better to make your own routine.

Some polygon routines have the limitation that they can only draw triangles. To get the maximum range of use, in this course we will make a routine that can draw polygons with as many vertices as you want. This has several advantages, one of which is that it is faster to draw them. It does, however, impose a few limitations on the appearance of polygons. We will return to this issue later in the course.

In this course we have chosen to make a very simple routine for drawing polygons, and it therefore does not have the maximum speed. However, the advantage is that it will be easier to understand, it can be used for almost anything, and it is relatively easy to modify for your own use. It also reduces the risk of serious errors.

The Blitter

In order to create a polygon routine, it is important to be aware of how the blitter works when filling in surfaces. The first thing that may seem unusual is that the blitter works from bottom to top, and from right to left. Bit 1 of BLTCON1 (DESCending mode) should therefore be set to 1. There are three other bits which are also interesting when it comes to filling:

Bit 4: EFE (Exclusive Fill Enable)

Bit 3: IFE (Inclusive Fill Enable)

Bit 2: FCI (Fill Carry Input)

Each of these needs a more detailed explanation:

The blitter has two modes of fill, and they are selected by setting either EFE or IFE. The FCI bit must also be set sensibly (See Figure 5).

Practically it works like this:

Go pixel by pixel to the left. Each time you get to a pixel that is set, alternate between filling and not filling. The difference between IFE and EFE is that the latter also deletes the pixel that turns off the fill.

If FCI is set, the blitter will set the first pixel (and continue to do so until it gets to a 1). If, on the other hand, you turn off FCI (FCI=0), the blitter will not begin the fill until it gets to the first 1.

The blitter works like this:

It makes its own copy of the FCI bit - let's call it FCI'. When EFE is set, FCI' is inverted for every 1. In this mode, the FCI' bit is set over both the 0 and 1 bits. Since the FCI' bit also replaces them, the 1 that turns off the fill will be erased. When IFE is set, on the other hand, it will just insert the FCI' for the 0's. It inverts FCI' for every 1 as usual.

So there are many options when drawing polygons. We have chosen to concentrate on EFE filling with FCI=0. The FCI choice is completely arbitrary. As you have probably seen, changing the FCI in EFE mode will result in a negative. It doesn't matter for the blitter, it can negate anything without spending any time on it.

The advantage of using EFE mode is quite obvious - it is the simplest and fastest way to get sharp edges. If you think about it, sharp edges are impossible in IFE mode, as the width of what is being filled will necessarily be at least 2. After all, you need one pixel to start the fill, and one to stop it.

Since the blitter in EFE mode deletes the leftmost pixel, you have to compensate for this by moving all the lines on the left side of the polygon one pixel to the left. This prevents the lines from being drawn on top of each other at the top and bottom. We are now ready to proceed with the next step in the polygon routine.

Here, unfortunately, new problems appear. First, the fill requires that there be two pixels per horizontal line, and only two. If the line is more horizontal than vertical, it will sometimes be more than 2 pixels per horizontal line (Figure 6A). This will cause the fill to be performed incorrectly. But this problem can of course be solved:

Bit 1 of BLTCON1 (SINGle bit per horizontal line) puts the blitter into a special mode that only gives one pixel per horizontal line.

The fill now works fine again (figure 6B).

Line drawing routine

The first thing to do is to create a routine that takes care of line drawing. Before we start, let's see what it takes to get the blitter to draw lines:

The blitter divides its "world" into 8 octants (Figure 7). It's done this way to make it as simple as possible for the blitter (and more work for the processor...). The first thing to do, therefore, is to find out in which octant the line belongs.

Imagine that the line goes from (x1,y1) to (x2,y2) (Figure 8). Then place the center of the octant figure in (x1,y1). You must then find out in which octant (x2,y2) ends, i.e. in which direction the line goes. If the line is exactly on the dividing line between two octants, it doesn't matter which one you choose.

Before you can start drawing lines at all, the blitter must be put into line drawing mode. There are a number of registers that need to have specific contents in this mode, so the easiest thing to do is to create your own routine that initializes the blitter before you start drawing lines. It can then be executed before the first line is drawn, and you avoid thinking about it any more. We have made such a routine, and called it "initline".

init_line sets:

- bltafwm to $ffff	
- bltalwm to $ffff	The blitter will have these three registers this way in line mode.
- bltadat to $8000	
- bltbdat to $ffff:	Determines the texture of the line. $FFFF gives an unbroken line.
- a5 to $dff000:	We use A5 as the base register when addressing the hardware registers.
- bltcmod to #ln_mod	
- bltdmod to #ln_mod:	bltcmod and bltdmod contain the width of the bitplane thats drawn in. We have chosen to put the width into a predefined variable "lnmod", so that it should be easiest to change. In addition, it will be more manageable.

Now we just need to fill the other registers with something meaningful. We have chosen to use the registers d0-d3 to represent the coordinates. The line is drawn from (d0,d1) to (d2,d3). For simplicity we call abs(d2-d0) for X and abs(d3-d1) for Y. X indicates the width of the line, while Y indicates the height.

After finding out which octant the line goes to, we know whether x is greater or smaller than y, and what sign they have. This is good, because the blitter has no understanding that Y can be greater than X. This occurs in 4 of the 8 octants, and in these X and Y change importance. We must therefore swap X and Y in these octants. We then do a little trick: If Y turns out to be larger than X, we just swap the two registers representing y and x in the calculation. After this is done, the rest of the calculations can be performed.

The blitter registers must be set in this way:

`bltamod`	`= 4y-4x`	
`bltbmod`	`= 4y`	
`bltapth`	`= 2y-x`	
`bltcon0[bit15-12]`	`= x1 & $f`	
`bltcon0[bit11-0]`	`= $B5a:`	$B=use A, C & D. $5a=Minterm ->XORe line ind.
`bltconl[bit15-12]`	`= 0:`	Start bit for line texture is uninteresting.
`bltconl[bit4-2]`	`= Octant`	
`bltconl.SIGN[bit1]`	`= 1:`	This puts the blitter in the special mode which only draws one pixel per horizontal line.
`bltconl.SIGN[bit6]:`		1 if the line is at least twice as long in one direction as in the other, otherwise 0.
`bltsize[bit5-0]`	`= 2:`	Must be this way for line drawing.
`bltsize[bit15-6]`	`= X+1:`	This corresponds to the total number of pixels in the line, or the line length if you like.
`bltcpt.l`	`=`	Address of the first pixel:
`bltdpt.l`	`=`	Address of first pixel: (x1/16)*2 + y1*ln_mod + bitp1.address

The address of the first pixel is calculated in the same way as if we were to calculate the address of the coordinate (d0,dl). However, it is important to remember that the blitter must have equal addresses! The X coordinate must therefore first be divided by 16 to find the number of words, and then multiplied by two again to convert them to bytes.

The Y coordinate is multiplied by the number of bytes per line (#ln_mod). If we add the two numbers, we can see how far from the top left corner the pixel is. If we then add the start address, of the bitplane we are drawing in, we arrive at the final address.

Now everything should be ready for the routine itself.

We have called the routine "**draw_line**"

Before calling it, the arguments must be inserted as follows:

`d0`	`= first x-coordinate`
`d1`	`= first y-coordinate`
`d2`	`= last x-coordinate`
`d3`	`= last y-coordinate`

`a3`	`= address of the first byte in the bit plane to be drawn in`
`ln_mod`	`= number of bytes per horizontal line in the bit plane.`

A macro "LnModMul" must also be defined to multiply numbers by ln_mod. The reason we don't use a simple "mulu #ln_mod,Dn" is that it can be executed faster in other ways.

The next five instructions actually perform two tasks:

They calculate the value to enter in bltcon0, while calculating the contribution of the first X coordinate to the address. It works like this:

bltcon0 must be given the value $xb5a, where x is the four lowest bits of d0. To calculate the x-coordinate contribution to the address, first divide by 16 (= shift the 4 steps to the right), then multiply by 2 again.

`move.w`	`#$b5a0,d4`	d4=$****.b5a0 (*=unknown, the dot is only there to distinguish between the upper and lower half of the register.)
`swapd4`		d4=$b5a0.****
`move.w`	`d0,d4`	d4=$b5a0.xxxx (xxxx=X coordinate)
`ror.l`	`#4,d4`	d4=$xb5a.0xxx This instruction "kills two birds with one stone". While rotating the four lowest bits of the x-coordinate into the top word of d4, the x-coordinate is divided by 16.
`add.w`	`d4,d4`	This is the only thing missing - multiplying the lower half of d4 by two. d4.w=(x/16)*2, while d4.w after a "swap d4" can be moved directly into bltcon0.

We wait as long as possible to move anything into the blit registers - hoping it finishes the previous blit operation before we finish all the calculations. If data is moved into the blitter registers before it is finished, the blitter will make all sorts of errors, and in the worst case, it can cause the machine to crash. So if you are too busy, it just causes serious errors in the graphics.

Now it's time to calculate the Y coordinate part of the address. It is done like this:

`move.w`	`d1,d7`	
`LnModMul`	`d7,d6`	Calculates the Y coordinate part of the address.

Note that the contents of d1 are saved, as they will be used later. LnModMul can be something as simple as "mulu #ln_mod,\1", but once you have decided on an ln_mod, it may be better to make a routine that multiplies by ln_mod. For example, "lsl #6,\1" would be a much faster way to multiply by 64. D6 is here thought of as a register that LnModMul can use for intermediate calculations.

```
        add.w   d4,d7           D7 = Distance from top left corner.

        lea.l   0(a3,d7.w),a0   A0 = Complete address of first dot.

        sub.w   d0,d2           d2 = d2-d0 (=x2-x1 = width of the line)

        bge     xpos            jumps to xpos if x2 >= x1

xneg                            Here you only end up if x2 < x1, i.e. the line

                                goes from the right to the left

        neg.w   d2              The blitter doesn't like negative widths, so we

                                negate it and get a positive one instead.

        sub.w   d1,d3           d3 = d3-d1 = y2-y1 = height of the line

        bge     xneg_ypos       jumps to xneg_ypos if y2 >= y1

xneg_yneg                       Here you only end up if x2<x1 and y2<y1,

                                i.e. if the line goes upwards to the left.

        neg.w   d3              The blitter doesn't like negative heights either,

                                so it's negated and becomes a positive number.

        cmp.w   d2,d3           The width and height are compared.

        bge.s   octant3         If the height is greater than the width we

                                are in octant 3.

octant7                         Octant 7: The line goes up to the left and it

                                is wider than it is tall, i.e. it goes more to the left
                                than up.

        moveq   #(7<<2)+3,d6    The number that goes into bltcon1 is moved

                                into d6, so that we can wait until afterwards to

                                put it in. The number (7<<2)+3(=%00011111) turns
                                off the line drawing mode in the special mode that
                                filling with the blitter requires. This number selects
                                also octant 7.

xgty                            This macro is now executed, after we have

                                found out which octant the line belongs to, the
                                start address of the first point, and whether the
                                width or height is greatest. XGTY is the macro
                                used when the line is wider than it is tall, so that
                                then x = width and y = height. NOTE! This macro
                                is terminated by an RTS, so that it the terminates
```

			the line drawing routine! (see also explanation page 15).
`octant3`			The line goes up towards the left, and it goes more upwards than to left (x2<x1,y2<y1)
	`moveq`	`#(3<<2)+3,d6`	d6 = bltcon1 for octant 3
	`ygt`		This macro is equivalent to XGTY, but it is used when the height of the line is greater than the width.The difference is that then X = the height, while Y = the width. This macro ends also with RTS, so that it ends the line drawing routine.
`xneg_ypos`			Here you end up if x2<x1
	`cmp.w`	`d2,d3`	The width and height are compared
	`bge.s`	`octant2`	Height >= Width: Octant 2
`octant5`			The line goes down towards the left, and more to the left than down
	`moveq`	`#(5<<2)+3,d6`	d6 = bltcon1 for octant 5
	`xgty`		Macro which does the rest of the work when the width is greater than height.
`Octant2`			The line goes down towards the left, and more down than to the left.
	`moveq`	`#(2<<2)+3,d6`	d6 = bltcon1 for octant 2
`ygtx`			Macro that does the rest of the work when the height is greater than width, (see explanation on page 15)
`xpos`			Here you end up if the line goes straight up, straight down or towards right.
	`sub.w`	`d1,d3`	d3 = d3-d1 = y2-y1 = height of the line
	`bge`	`xpos_ypos`	y2=>y1 : Jump to xpos_ypos
`xpos_yneg`			x2>x1 and y2<y1, i.e. the line goes up to the right
	`neg.w`	`d3`	Converts the height of the line to a positive number.
	`cmp.w`	`d2,d3`	Compares the width (d2) and the height (d3)
	bge.s	octant1	If the height is greater than the width, the line ends in octant 1.
`octant6`			The line goes up to the right, and more to the right than up.

```
        moveq   #(6<<2)+3,d6    d6 = bltcon1 for octant 6
xgty                            Performs the rest of the job when the width is
                                greater than the height.
octant1                         The line goes up to the right, and more up than to
                                the right.
        moveq   #(1<<2)+3,d6    d6 = bltcon1 for octant 1.
 ygtx                           Does the rest of the job when the height is greater
                                than the width.
xpos_ypos                       The line goes down to the right.
        cmp.w   d2,d3           Compares the width (d2) and the height (d3).
        bge.s   octant0         Height > width: octant 0
octant4                         The line goes down to the right, and more to the
                                right than down.
        moveq   #(4<<2)+3,d6    d6 = bltcon1 for octant 4.
xgty                            Completes the line drawing when

                                the width is greater than the height.
octant0                         The line goes down to the right, and more down
                                than to the right
        moveq   #(0<<2)+3,d6    d6 = bltcon1 for octant 0
ygtx                            Completes the line when the height is greater than
                                the width.
line_end                        End of the line drawing routine.
```

Here are the definitions of the macros **xgty** and **ygtx**. These should be placed in front of the "drawline"-routine in the program, but we've included them here at the end to get them into the order in which the machine executes them.

XGTY MACRO

This macro is called up with:

- The address of the first dot in line A0

- The width of the line in d2

- The height of the line in d3

- The part of bltcon1 that determines the direction of the line in D6

(Since the width is greater than the height, x=D2 and y=D3)

YGTX MACRO

This macro is called up with:

- The address of the first dot in line A0

- The width of the line in d3

- The height of the line in d2

- The part of bltcon1 that determines the direction of the line in D6

(Since the width is greater than the height, x=D3 and y=D2)

The "ygtx" macro is almost identical to xgty, except that d2 and d3 are swapped. The is because of the blitter's "idea" that x must always be at least equal to y, and it is not the case in the octants 0,1,2 and 3. Therefore, the blitter is "clever", and swaps the meaning of x and y in these octants. Thus, x becomes larger than y anyway.

`add.w`	`d3,d3`	d3 = 2y
`move.w`	`d3,d0`	d0 = 2y
`sub.w`	`d2,d0`	d0 = 2y-x
`bgt.s`	`xgty_nslack\@`	Jumps to "xgty_nslack" if x<2y, i.e. the line is not quite twice as wide as it is tall.

This may look quite uninteresting, but it is important for the blitter. Such a line will consist of segments that are either one or two pixels wide, and the first segment must always be only one pixel wide. This is done using a 0 in bit 6 of bltcon1.

`bset #6,d6`	Sets bit 6 D6, i.e. bit 6 of bltcon1. The line is thus at least twice as wide as it is tall, and each segment in the line must therefore be at least 2 pixels wide. If this bit is set, you are telling the blitter that it first segment to be at least 2 pixels wide.
`xgty_nslack\@`	Here bit 6 of D6 is set as it should, since the line above otherwise skipped if the line is not at least twice as wide as it is tall.
`ifeq line_waitblit+1`	This is a command directly to the assembler.

The following lines should only be included in the program if "linewaitblit" is set to -1

`btst.b`	`#6,dmaconr(a5)`	Checks bit 6 of dmaconr, i.e. the "blitter finished" flag.
`bne.s`	`xgty_nslack\@`	Jumps back to "xgty_nslack" if the blitter is not finished its previous job yet.

`endc`		End of the IFEQ block.

Normally line_waitblit should be set to -1. There is only one way to avoid this test, and that is to put all the write routines into an interrupt, so that they are not called until the blitter has finished its previous operation. However, we have not done this in this program, but see anyway at the end of the course under the section "Possible improvements".

`move.w`	`d0,bltaptl(a5)`	bltapt1 = 2y-x
`sub.w`	`d2,d0`	d0 = 2y-2x
`add.w`	`d0,d0`	d0 = 4y-4x
`move.w`	`d0,bltamod(a5)`	bltamod = 4y-4x
`add.w`	`d3,d3`	d3 = 4y
`move.w`	`d3,bltbmod(a5)`	bltbmod = 4y
`addq.w`	`#1,d2`	d2 = x+1, number of pixels on the line, i.e. the length of the blitter is interested in.
`lsl.w`	`#6,d2`	Shifts the length of the line 6 steps to the left, such that it is in the right position before it is to be added into bltsize.
`addq.w`	`#2,d2`	Bit 1 must also be set in bltsize for the blitter to be satisfied.
`swap`	`d4`	Retrieves the value to put in bltcon 0 and which we calculated at the beginning of the line drawing routine. D4 contains information about which bit (from 0 to 15) the line begins in, in address a0, and how the line should drawn.
`move.w`	`d4,bltcon0(a5)`	bltcon0 = $xb5a (x is the 4 lowest bits of the x coordinate)
`move.w`	`d6,bltconl(a5)`	bltcon1 = octant selection + information if the line is double as wide as it is high or not.
`move.l`	`a0,bltcpth(a5)`	The address of the first pixels of the line is inserted in bltcpth
`move.l`	`a0,bltdpth(a5)`	and in bltdpth

`move.w`	`d2,bltsize(a5)`	Finally, the length of the line is added to bltsize. This starts the blitter, and must therefore be executed at the very end!
`rts`		Returns from the line drawing routine!
`endm`		End of the xgty macro.

Boundary of the polygon

Now the line drawing routine is finished. It can now be used to draw polygons, but first we need to think through what we really want. We need to draw a line around the entire polygon, and these lines need to be of the type that the blitter requires to fill in properly, i.e. only one pixel for each horizontal line. The line drawing routine has already taken care of that, so that matter is in order. We also need to move all the lines on the left side of the object one pixel to the left, since we are using the EFE fill. The last problem is just to get the corners right.

If we draw the polygons with what we have so far, only the top and bottom corners will be correct (Figure 9). So we need to prevent one line ending on the same line as another one begins on. The simplest way to do this is to draw a line where we remove the first dot. We have already calculated the address of this dot, so it should not cause any particular problems.

The direction in which the lines are drawn also plays an important role. The blitter puts the first pixel on each new line it comes to, and a line drawn upwards to the right will be further to the left than one drawn downwards to the left (Figure 10). This can be exploited by drawing all the faces as large as possible, so that there are no ugly gaps between them. To achieve this, the lines should be drawn towards the top and bottom as shown in Figure 11.

Only two of the lines around the polygon should be drawn normally, i.e. without removing the first dot. This is one of each of the lines starting from points A and B respectively. Note that point A is always the point furthest to the left, whether it is at the top or the bottom. Similarly, B is always the point furthest to the right.

The line drawing routine in this course is designed to draw lines without the first dot.

Now we have planned and figured out which method to use to draw the lines around the polygon. Now "just" filling, copying into the bitplanes and "clipping" is left.

Copying

The routine that draws the main objects must also contain information about where the object was drawn. When the object has to be moved, the graphic that was behind it has to be drawn again. You therefore need to know exactly which area to redraw. Of course, you can store a copy of the entire screen behind the vector objects and copy it back again between each time the vector objects are drawn. However, this takes quite a long time. It is much better to find the smallest square that is big enough to cover the whole object, and then just copy the graphics into that square.

In this way, we have created a vector routine that makes it easier to display vector objects over background graphics. You'll also save a lot of memory and some time if you can make do with a solid background. All you then need to do is delete the area where the object was drawn.

Clipping

An important part of a polygon routine is "clipping" or rather trimming the edges. If you avoid drawing a polygon that has one or more corners outside the screen, it will immediately look good enough. However, if part of the polygon ends up outside the screen, and many do, then you need to use the technique called "clipping".

If you draw the whole polygon and you don't want it to go partly or completely off-screen you need to be aware of the following:

What goes outside to the left comes back in from the right.

In addition, data that goes outside the top and bottom edges will overwrite other important data in memory.

It is therefore necessary to find a solution to the problem! The art of drawing only the part of the polygon that appears on the screen is called clipping. There are several ways to do it, but we have chosen to use the simplest one, even if it is not the fastest in all cases.

The polygon is drawn in a temporary bitplane, temp-plane, which is 512 x 512 pixels in size. It should be able to contain even the largest polygons. In order to get the best possible space for the polygons, draw in the upper left corner of the temp-plane. The polygons can then be about 500 pixels high and wide before the polygon routine refuses to draw them. However, it is easy to avoid such sizes.

If even larger polygons are needed, the temp bitplane can be expanded to 1024 x 1024 pixels. However, this takes up 128 kB of chip memory, compared to the 512 x 512 pixels which "only" take up 32 kB.

Before the polygon is drawn, it is moved as far to the left and upwards as possible to make room for as large a polygon as possible. To simplify both cutting and copying to the final bitplanes, the polygon is moved one or more whole words to the left. That is, k*16 is subtracted from the x coordinate, where k is an integer.

Before you start drawing a polygon, you need to know in advance which point is furthest to the left and which is furthest to the top. The leftmost x-coordinate is called *minx* and the topmost y-coordinate is called *miny*.

To move the polygon where you want, subtract (*minx* & 16) from all x-coordinates and *miny* from all y-coordinates. This means that the top corner ends up at the very top of the temporary bitplane, while the left corner is placed between x=0 and x=15, i.e. in the leftmost word.

When the polygon needs to be copied, the same numbers need to be added again. Here, however, we will need the cut. If the left edge of the polygon is to the left of the screen edge, do not copy that part in. Similarly, do not include the parts that are above the top, to the right of the right edge, and below the bottom. This avoids parts of the polygon ending up outside the screen.

The clipping is illustrated in figure 12. There you can see how much the different edges need to be trimmed to look good. In the example in figure 12, the polygon is so large that it goes outside the screen on all 4 sides - something you rarely encounter. However, it is easiest to see how it works with this illustration.

Vertical clipping is done by "clipping" each pixel, while horizontal clipping is done with the whole word, i.e. 16 pixels. So on the left side, subtract the polygon's left x coordinate from the screen's x_coordinate in the temporary bitplane. Then add 15 and divide the result by 16 ((min x - x4 + 15)>>4). The reason to add 15 is to ensure that you crop the image by a full word, even if it is only 1 pixel too large for the screen.

In some situations, you also avoid filling in the part of the polygon that goes below and above the screen. The same goes for the part that comes beyond the left edge. However, the right edge must be filled in because the blitter works from right to left. It needs the first stroke of the polygon to "activate" the fill. However, it stops by itself when it gets to the left edge of the screen.

If you do not start from the right with the fill, you can be sure that the polygon will not be filled. It is therefore necessary to fill in a larger part of the polygon than that shaded in Figure 12.

To illustrate how a clipping routine works in assembler, an example "poly.s" is stored on the diskette. It contains both the source code, the fill, the clipping and the copying of the polygon on the screen.

Rotation routine

You have now reached the point where you can rotate an object about all 3 axes. The rotation around the axes is defined by these angles (Figure 14):

rx : The angle rotated around the x-axis

ry : The angle rotated around the y-axis

rz : The angle rotated around the z-axis

(nx,ny,nz) is the coordinate before rotation

(X,Y,Z) is the coordinate after rotation

<u>Rotation around the x-axis</u>:

Z=nz*cos(rx)-ny*sin(rx)

Y=ny*cos(rx)+nz*sin(rx)

<u>Rotation around the y-axis</u>:

X =nx*cos(ry)-z*sin(ry)

nz=z*cos(ry)+nx*sin(ry)

<u>Rotation around the z-axis</u>:

nx=x*cos(rz)-y*sin(rz)

ny=y*cos(rz)+x*sin(rz)

Positive rx gives counter-clockwise rotation from the right Positive ry gives counter-clockwise rotation from above Positive rz gives counter-clockwise rotation from the front

A complete rotation routine in assembler using these formulas can be found in "rotate.s"

As you can see, we need the trigonometric functions Sin and Cos to calculate the rotation. These functions can of course be calculated, but it will slow down the vector routine too much. Therefore, to save time, we calculate some sine and cosine values in advance and put them in a table (this can be done in a basic program, for example).

You should think through these tables carefully before you start making them. The first problem you encounter is that Sin and Cos return values between -1 and 1. For example, Sin(1.3rad)=0.963558... The microprocessor can only process integers, and so there are only three possibilities, -1, 0 and 1. We solve this by multiplying by as large a number as possible, but still taking care that the numbers are not too large to fit into a word. A suitable large number is 16384 (214). We then get values that are between -16384 and 16384. This gives a precision that is more than good enough for the movements on the screen to be smooth. We address this issue a bit more in the chapter "Measuring angles".

Measurement of angles

In mathematics, as mentioned before, it is common to measure angles in either degrees or radians. In radians, angular measurements lie between 0 and 2π(=6.283185...). However, this gives far too poor a resolution and it does not add up to a whole number. If you measure in degrees you solve these problems and you then have 360 different angles available. That may be enough to make useful and smooth movements, but 360 is not a very good number for computers. If you use degrees, the degree number should be between 0 and 359. Imagine that to rotate an object, you have to add or subtract a fixed number to the angle each time. If the angle is less than 0, add 360, and if it is greater than 359, subtract 360. This becomes too tedious.

Then it is much better to choose a new angle measuring system, so that the numbers can be adjusted with a single AND instruction. A favourable system is 1024 "data lines". Since each element in the table is a word, this means that the whole table will be 2048 bytes (=2 kB) long. This is so short that it has little impact on free memory, while being long enough to provide perfectly smooth movements.

In order to perform the processing as fast as possible, only data lines in even numbers between 0 and 2046 are accepted, i.e. ordinary data lines multiplied by 2. This is done to be able to address the table directly with the datagrams. Therefore, to adjust a data line in a data register, just write "AND #2046,Dn", where Dn is the current data register. This way you are sure to get even numbers that lie between 0 and 2046.

When you need to get a specific sin or cos value, you need an address register that points to the beginning of the table we call it An. You also need a data register that contains the data values we call it Dn. The answer comes out in another data register, Dm.

It's all done like this: "**MOVE.W 0(An,Dn.w),Dm**".

A concrete example: A3 points to the start of the cosine table. You want to find the cosine of the datagrid, which is in D0, and you want it put in D1. It will be like this:

"**MOVE.W 0(A3,D0.w),D1**".

Cosine and sine tables

There is both a sine and cosine table on the disk. Below is the interesting part of a basic program which can be used to calculate the cosine table:

```
FOR t = TO 1023
v = t/6.283186                    (Converts data lines to radians)
x% = INT(COS(v) * 16384 + 0.5) (x% is the value to put in the table)
NEXT
```

This formula is used to convert radians R to data degrees D:

$$D=INTEGER(\frac{R\cdot 1024}{2\Pi}+0{,}5)$$

The angle 0.7 radians gives 114 data degrees, which corresponds to 228 equal data degrees when inserted into this formula.

We can then calculate back again and see what radian angle 114 degrees corresponds to:

To convert data-words to radians:

$$R=\frac{D\cdot 2\Pi}{1024}$$

If 114 is inserted into this formula, we get 0.6994952...

So when the basic program calculates the table value for 114 data degrees, it uses the angle 0.6994952 radians.

The table value then becomes:

INT(cos(0.6994952)* 16384+0.5) = INT(0.7651672..*16384+0.5) =

INT(12537.0004...)=12537.

That is, when:

you will find Cos(0.7rad) = Cos(40.11grader) = Cos(114datagr) = Cos(2281igedatagr)

Cos(0.7rad)=0.76484...

Cos(2281igedatagr) = 12537/16384 = 0.76520.., an inaccuracy of only 0.047%. It is so small that it has no practical significance.

Perspectives

With the three-dimensional coordinate system, we can give all points a coordinate, no matter where they are. The problem with 3-dimensional graphics is that they have to be displayed on a 2-dimensional monitor. The eye must therefore be "tricked" into seeing an extra dimension. This is done by adding perspective.

Perspective itself is far from being a great mystery, and Figure 15 should give a clear picture of how it is calculated. In fact, it is nothing more than dividing all X and Y coordinates by the distance from the eye (=Z value after rotation) before plotting them.

Example of Rotation and Perspective

Now you've learned enough to do a little demonstration with dots moving in 3-dimensional patterns. In the demo "dots1" you see an example of this. The source file is called "dots1.s".

In "dots1" we have made a dot-figure, which can be controlled with a joystick. It demonstrates rotation and perspective (figure 16). Note that in addition to the Z value, we also add a fixed "imaginary" distance from the eye to the screen, so that even negative z values can be displayed on the screen. These should be imagined to lie between the screen and the viewer, i.e. in the air outside the screen. (See also the section "The coordinate system").

Filled surfaces

Dots are one thing, but it's quite another when we need filled surfaces. Here we encounter some new problems. First, let's take an object that everyone is familiar with, the cube. Take a cube in your hand and turn it. No matter how hard you try, you can never see more than 3 of the 6 sides. In some positions, you may not see more than 1 or 2 sides.

These lessons are good to keep in mind when talking about computer graphics. The pages you can't see, you don't need to draw! This way you avoid drawing about half of all the polygons in each object. We just need to find a simple and easy way to find out which sides are not visible. An incredibly simple way to do this is the "clockwise" method. It implies that all polygons should have coordinates that are clockwise when they are visible (Figure 17).

Look at this example to illustrate: take a floppy disk or a piece of paper about the same size. Choose a page that you call the front page. On it, write "1" in the upper left corner, "2" in the upper right corner, "3" in the lower right corner, and "4" in the lower left corner. Then turn the disk/paper over and write the same numbers that are in the corresponding corner on the other side. Now you will see that no matter how you turn the sheet/disk, the numbers will always be clockwise as long as you look at the front, and they will always be counter-clockwise when you look at the back.

Convex objects

DEFINITION OF CONVEX OBJECT

Any straight line from a point on an object to another point on the object must either pass along the surface or through the object.

If there is at least one line that breaks this rule, the object is not convex.

Examples of convex objects:

- CUBE

- PYRAMID

- FOOTBALL

Return to the cube again. Notice that either one side is invisible, or you see the whole side. It never happens that a page is partially covered by one of the other pages. Therefore, if you need to draw such an object, just find out which of the sides are visible, and then draw them in any order. Such objects are called CONVECT objects, i.e. objects that can never "shade" themselves. On the disk there is a program which shows a cube with 6 different colours. It is called "kube1.s" as source code, and it is also in executable form as "kube1". The source code "kube1.s" also contains explanations of the program lines.

Inconsistent objects

If all objects had been convex, it would have been no problem to make vector routines with what we have learned so far. However, the vast majority of objects are inconsistent. An example of this is an ordinary telephone pipe. The trick is to split an in-congruent object into several convex sub-objects (Figure 18A and 18B). Then each sub-object can be drawn as a single convex object, that is, only the visible sides are drawn. The only problem is what order to draw these sub-objects in. We will look at this in the next chapter.

Sorting algorithm

We use a small advanced algorithm which has several advantages: it is fast and it works without errors when certain conditions are met. However, there is one drawback: you have to think carefully when designing objects. We have therefore created some exercises for you to practice, which deal with this very issue, at the end of the chapter: "Tasks".

Sorting sub-objects

The method is best explained by some examples: see Figure 19. It consists of two blocks; I and II. Notice that when face A is visible, block II is always in front of block I. It is this phenomenon that we exploit in our plotting algorithm. When block II must be in front of block I, block I must be plotted first, since block II must be plotted on top of block I to be visible.

' On the other hand, when face A is invisible, block I will always be in front of block II. So block II must be drawn first, since block I must then be drawn on top of block II. We call this "sorting" the sub-objects, because we sort (or order) them in the order in which they are to be drawn.

This algorithm is very fast because we have already checked whether face A is visible or not. This is done as part of the drawing process, as we want to know which faces need to be drawn. This sorting algorithm is called Z-sorting.

The theory behind it is the following: If you imagine that one of the faces is extended to infinity, then all the sub-objects that are on one side of the face can be "sorted" using this algorithm. Subobjects that intersect surfaces, on the other hand, cannot be sorted.

This may be easier to understand with an example. Try to see figure 20:

If you expand surface A to an infinite size, then both objects II and III will be located completely on the right side of it. However, the surface will split object IV in two, so object IV cannot be "sorted" using surface A.

However, we have not exhausted all our possibilities yet. Block I has 5 other surfaces, and here surface E can be extended so that object IV is completely on the underside of this. We now have a "sorting basis" for all the sub-objects in relation to object I. The procedure is the same for the other 3 sub-objects. As you can see, it has already become a bit complicated - even with so few sub-objects - so more complicated objects may become even more difficult. It is therefore important to concentrate on the task and that you understand this theory. Review it several times if necessary and remember that practice makes perfect!

So we've made some objects at the back of the booklet for you to practise on. This is exactly the part of designing objects that is the most time-consuming, and it is also the part where it is easiest to make mistakes. So spend some time practising. Once you've mastered the art, you can make big, beautiful objects that - without any bugs - can move in all sorts of directions.

Main objects

There are a number of things that are important to be aware of when "designing" the main objects:

You need to make a coordinate table for each main object. The coordinate table must contain the X, Y and Z coordinates of all vertices used in the main object. The number of coordinates minus 1 must also be included in the main object description. For example, if a main object needs 37 coordinates, enter the number 36 in the main object description.

In the main object description, enter the address of the coordinate table. In addition, you give the address of a "drawing coordinate table" and a "calculation table". When all the coordinates have been rotated, the new coordinates are added to the calculation table. Then perspective is added, the screen offset is added so that the centre point is in the middle of the screen, and the coordinates are added to the plotting table.

There they are as (x,y)-coordinates, the z-coordinate is dropped out. It is these coordinates that are used directly in plotting the screen - hence the name.

Calculation table

You need to allocate (number of coordinates * 6) bytes to the calculation table.

The plotting table.

You must allocate (number of coordinates * 4) bytes to the plotting table.

The main objects must be divided into several sub-objects.

Each of these sub-objects MUST be convex, otherwise the plotting will work properly.

Each of the sub-objects must be assigned a number, and the first sub-object is given number 0. The next one gets the number 1, the next 2 and so on...

There is no limit to how many sub-objects you can have, but be careful don't get too complicated! It is useful to divide the main objects into as few sub-objects as possible.

It is a great advantage that the first sub-object can be used to sort ALL the other sub-objects of the main object.

If you cannot sort all the other sub-objects in relation to the first sub-object, there are two other ways to solve the problem:

1. One is to add extra invisible faces to the sub-object that looks the most promising. These surfaces are called "phantom surfaces". Phantom surfaces are created by adding the colour number of both the front and the back to -1, how to create sub-objects).

2. The second method is to create an additional "phantom sub-object". This "phantom sub-object" must have the property that all side faces are invisible, i.e. both the front colour and the back colour are set to 1, on all faces of the sub-object. If you do this cleverly, you may end up sorting all the other sub-objects using it.

If you want to display multiple main objects on the screen, it's fairly simple. You just draw the main objects furthest away first, and the ones closest to last.

Sub-objects

Here is an overview of what you need to be aware of when creating sub-objects:

All sub-objects must be convex.

However, this is a truth with some modifications. In Figure 21 we have illustrated 2 different, perfectly legal sub-objects. As you can see, sub-objects do not have to be closed. And so, strictly speaking, they cannot be said to be convex, but the term takes on meaning nonetheless.

If you expand the faces of the object in Figure 21B, you get the same object as in Figure 21A, and it is convex. So, if you have a convex object, you can "shrink" one or more of the side faces and still call it convex.

Each side surface can have a different color, with the color also being different on the front and back sides.

How many colours you have available depends - as you may know - on the number of Bitplanes.

1 Bitplane: 2 colours (0-1)

2 Bitplanes: 4 colours (0-3)

3 Bitplanes: 8 colours (0-7)

4 Bitplane: 16 colours (0-15)

5 Bitplane: 32 colours (0-31)

6 Bitplane: Extra halfbrite: 64 colours (0-63)

Note that the colour numbers start with 0, not 1.

Remember that if you use background graphics, the colours must be shared between the vector objects and the background graphics!

If you specify too high a colour number, the colour number will automatically be adjusted down to an acceptable level. This is because the colors are selected from as many of the lower bits as there are bitplanes:

- At 1 bitplane, only the bottom bit of the color number is taken into account.

- For 2 Bitplanes, only the 2 lowest bits of the colour number are taken into account.

- For 3 Bitplane, only the 3 lowest bits are taken into account, etc.

- LIMITATION: The colour number must be positive, i.e. less than 32768 ($8000).

In most cases, Hold and Modify mode is not very well suited for vector graphics.

The front and/or back can be made invisible by giving them the colour number -1.

In closed sub-objects, the "back" or "inside" of the faces will never be visible. If you want to significantly increase the speed of the drawing, set the back colour to -1 on closed sub-objects.

In open sub-objects, remember to set the color of the back of the faces to something other than -1. It looks silly if a surface suddenly disappears into thin air. Often a slightly darker colour on the inside than on the outside looks good.

The coordinates of the sub-object are common to the whole main object. So you need to make a coordinate table for each main object - not for each sub-object. This is because many of the sub-objects are related and if each sub-object had its own coordinate table, many of the coordinates would appear in several of the tables.

The coordinates are given by index to the coordinate table. The first coordinate has the number 0, the second coordinate has the number 1 and so on.

The coordinate numbers must be multiplied by 4 before being inserted into the sub-object as each coordinate always occupies 4 bytes:

1 word for X coordinate and 1 word for Y coordinate. We do it this way in order to address them directly with the (An,Dn.w) addressing method, where An points to coordinate 0 and Dn is the coordinate index, after multiplying by 4.

For each of the side faces of the sub-objects, you need to devise the sub-object sorting. That is, you need to find out how many of the other objects lie WHOLLY on the front (outside) of each single surface if the surface is expanded to infinity. This number minus 1 must be entered in the sub-object description. In addition, the address of a table of the addresses of the current sub-objects must be given. If there are 3 sub-objects that are entirely on the face of a surface, give the number 2. If there are none, give -1. If -1 is given, the value of the table address does not matter, but it must be included anyway! This is because the length of all sub-object structures must be the same.

It sounds worse than it actually is. See the previous example "poly.s", where we review the design of a complete object. There is a complete program listing on the disk.

Description of main and sub-objects using structures

Now we have come to how the main objects and sub-objects are to be described using structures.

Structures are actually lists which can contain different types of values, while

Tables are used if it is a list with elements of the same type.

We use a special way of writing them here:

For each line, a size is specified, e.g. byte, word, long word..., and how many of them there should be.

"B:"	Means 1 byte
"<2>B:"	Means 2 bytes
"<3>B:"	Means 3 bytes etc...
"W:"	Means 1 word
"<2>W:"	Means 2 words etc...
"L:"	Means 1 longword
"<2>L:"	Means 2 longwords etc.
"<n>B:"	Means n number of bytes

Each element is assigned a name, so that you can address each element by its name. So you don't have to remember the offset all the time. In addition, each element of a structure is preceded by how many bytes from the beginning that element is located. This is what we call the "offset". The way of writing is like this:

name:(offset) <Number>Size: Explanation

If the name starts with an exclamation mark, it means that this variable is only used internally and that you should avoid writing to it. However, you can read the values, as they may be interesting from time to time. The names are written without exclamation marks in the programs.

If the explanation is followed by an asterisk (*), it means that it is an address of a new structure/table, as explained below.

Main object structures:

!objStatus(0)	W: Status word.
!objMinX(2)	W: Minimum X coordinate when drawing.
!objMinY(4)	W: Minimum Y coordinate at record.
!objMaxX(6)	W: Largest X coordinate when plotting.
!objMaxY(8)	W: Largest Y coordinate when plotting.
objX:(10)	W: X coordinate of the main object (positive to the right).
objY:(12)	W: Y coordinate of the main object (positive down).
objZ:(14)	W: Z coordinate of the main object (positive into the screen).
objRX(16)	W: Rotation around the X axis.
objRY(18)	W: Rotation around Y axis.
objRZ(20)	W: Rotation around Z axis.
objNoCrd(22)	W: Total number of coordinates in the coordinate table minus 1.
objCrd(24)	L: Address of coordinate table*.
objClCrd(28)	L: Address of the calculation coordinate table*.
objPlCrd(32)	L: Address of the plotting coordinate table*.
objSubs(36)	L: Address of the sub-object table. * There must always be at least one sub-object!
objNoSubs(40)	W: Number of sub-objects - 1.

The values to set when designing the object are:

NoCrd, Crd, ClCrd, PlCrd, Subs and NoSubs (i.e. the bottom 6).

The values that are set in the program and that create movement are:

X, Y, Z, RX, RY and RZ.

Values that you should never set are the top 5:

Status, MinX, MinY, MaxX and MaxY.

They are used by the plotting routines to store temporary results etc, and if you change them, there can be nasty consequences.

The coordinate table is simply structured. It simply consists of the coordinates organised in this way:

x0, y0, z0, x1, y1, x1, x2, y2, x2 etc, where all sub-coordinates are words. Each point is described by the x,y and z coordinates, and therefore requires 6 bytes per coordinate. The number of elements in this table is equal to objNoCrd+1. The length is therefore (objNoCrd+1)*6 bytes.

The coordinates must be pre-set. They determine the appearance of the object.

The calculation coordinate table is constructed in exactly the same way as the coordinate table, and is exactly the same size. The coordinates in this table do not need to be set to any particular value, since the plotting routines themselves insert values into this table.

The plotting coordinate table, on the other hand, is a bit special. Since the screen is 2-dimensional, we only need X and Y coordinates to describe the points on the screen. Therefore, after adding perspective, the 3 dimensions in the calculation coordinate table are reduced to 2 in the plotting coordinate table. It is structured as follows:

x0, y0, x1, y1, x2, y2, etc....

Again, each sub-coordinate is a word, so a complete plotting coordinate takes up 1 long word (=4 bytes). It is also not necessary to pre-enter values in this table; the plotting routines take care of that.

The sub-object structure describes how the individual parts of the object are related. It looks like this:

IsubStatus(0)	W: Status-word. Used by the drawing routine - for internal use only!
subNoPoly(2)	W: Number of faces in the sub-object minus 1.
subPolys(4)	<subNoPoly+1>L: Address of each page face that makes up the sub-object.*

As you can see, it is impossible to say how long the sub-object structure is. The length varies depending on how many faces the sub-object has.

The side faces are described by a "poly" structure. It is quite complex, and contains, among other things, complete information about the sort order of the sub-objects.

However, the structure is quite simple:

IpolyStatus(0)	W:	Status-word. For internal use only!
polyFCol(2)	B:	Color of the front (outside). -1 means invisible front. NOTE! EXCHANGE!
polyBCol(3)	B:	Color of the back (inside). -1 means invisible back. NOTE! BYTE!
polySame(4)	L:	Address of a surface in the same plane (Figure 22). Set to 0 if the is none. Used by the visibility check routine for faster processing.
polyNoFrn(8)	W:	Number of sub-objects in front of this surface minus 1
polyFrnTb(10)	L:	Address of a table of sub-objects that precede this surface.
polyNoCrd(14)	W:	Number of vertices in the polygon minus 1. A polygon must have at least 2 vertices.
polyCords(16)	<polyNoCrd+l>W:	Coordinate index. Remember! These must be

coordinate number * 4!

The table of sub-objects is also quite simple. It is built from <polyNoFrn+1> long words with addresses of the sub-object structures that are in front of the current surface. All values in this table must be set in advance as part of the object design.

Selecting coordinates for the polygon

When choosing coordinates for each polygon, it is important to remember to get them in the right order. Figure 23 shows a simple pyramid, which is a good example to illustrate what the correct order is. The coordinates are listed in order so that you can draw a line after them, and for example for the bottom polygon it would not be correct to say that the order is 1-3-2-4.

In addition, all coordinates must be clockwise when looking from the outside of the sub-object towards the surface.

The correct order for each of the faces of the pyramid in Figure 23 would then be: 1-5-2,2-5-3,3-5-4,4-5-1 and 1-2-3-4.

It doesn't matter which of the vertices you start with, so the first face could just as easily have had the coordinate order 5-2-1 or 2-1-5.

If the coordinates are not clockwise, the routine to check if the face is visible or not will report an error. This will cause errors in the drawing of both the sub-object and the order in which the sub-objects are drawn. It is therefore important to get this right from the start.

Cube/pyramid routine

Now we're going to use everything we've learned to make a single filled vector object that rotates on the screen.

The coordinates of the vertices are given in Figure 24, so it's natural to start with the coordinate table:

```
CoordTable

     dc.w -100,-50,-50

     dc.w 0,-50,-50

     dc.w 0,50,-50

     dc.w -100,50,-50

     dc.w -100,-50,50

     dc.w 0,-50,50

     dc.w 0,50,50

     dc.w -100,50,50

     dc.w 50,-50,-50

     dc.w 50,-50,50

     dc.w 50,50,50

     dc.w 50,50,-50

     dc.w 100,0,0
```

That should be it, 13 coordinates in total. In addition, we need to make two other coordinate tables, one for calculation and one for plotting:

```
CalcCoords

     ds.w 13*3          ;Sets aside 39 words, which is enough for 13 calculation
                        coordinates.

DrawCoords

     ds.l 13            ;Sets 13 longwords, which is enough for 13 drawing coordinates
                        coordinates.
```

We can now go directly to the main object structure:

```
object
     dc.w 0,0,0,0,0          ;The first 5 words are only used by the internal drawing
                              routine.
                              We set them to the initial value 0.
     dc.w 0,0,0,0,0,0        ;x,y and z coordinate and rotation are set by the rotation
                              routine.
     dc.w 12                 ;13 coordinates in total, (number -1 must be given here!)
     del  CoordTable         ;The address of the coordinate table.
     del  CalcCoords         ;The address of the calculation coordinates table
     del  DrawCoords         ;Address of the drawing coordinates table
     del  SubTable           ;Address of the list of sub-objects
     dc.w 1                  ;Number of sub-objects -1 (object consists of 2
                                  sub-objects)
```

Now it's the turn of the sub-objects. As you can see in the figure, the object consists of two clearly defined sub-objects. We call the cube sub-object "Cube" and the pyramid sub-object "Pyramid".

The sub-object list, which is called "SubTable" in the main object structure, becomes like this:

```
SubTable
        del   Cube,Pyramid.
```

The cube is now sub-object 0, and the pyramid is sub-object 1.

```
Cube
        dc.w 0      ;the status word is only used by the internal routines.
        dc.w 5      ;6 faces in the cube sub-object. The number -1 must be given here.
        del  poly0,poly1,poly2,poly3,poly4,poly5      ;Addresses of the
                                                       polygons that make up
                                                       the sub-object.
```

So that was the cube, and here is the sub-object structure for the pyramid:

```
Pyramid
        dc.w 0
        dc.w 4      ;5 faces of the pyramid
        del  poly6,poly7,poly8,poly9,poly10
```

Now we just need the polygon structures. As you probably remember, the sorting of the sub-objects is done in the polygon descriptions. As you can see, the pyramid is right in front of face 2, and the cube is right in front of face 10. We cannot say anything about the other faces, but they are not interesting either. One face is enough to determine the sorting.

To get more colors on the object, we give each side face the color number 10 + the number of the side face. Page 0 gets colour 10, page 1 colour 11 and so on...

```
poly0
        dc.w 0
        dc.b 10,-1;Front colour: 10. Back colour: invisible (object is closed)
        del  0      ; No faces in the same plane
        dc.w -1     ;No sub-objects in front of this face
        del  0      ;We specify the address 0 when there are no objects in front of
                    this faceplate.
        dc.w 3      ;This polygon has 4 vertices. Here, the number of vertices -1 must
```

```
                  be given.
    dc.w 0,1*4,2*4,3*4          ;The coordinate indices for this polygon multiplied
                                 by 4. To multiply the coordinate indices by 4 is
                                 quite important.
poly1
    dc.w 0
    dc.b 11,-1              ;Front: colour 11. Back: invisible
    del  0                  ;No surfaces in the same plane
    dc.w -1                 ;No sub-object in front of the surface
    del  0
    dc.w 3                  ;4 corners
    dc.w 4*4,5*4,1*4,0      ;Coordinates 4, 5, 1 and 0.

poly2
    dc.w 0
    dc.b 12,-1          ;Colour 12, reverse side invisible
    del  0
    dc.w 0              ;A sub-object is in front of this surface (pyramid)
    del  FrontPol2Table
                        ;Address of a list of sub-objects that are in front of this surface.
    dc.w 3                  ;4 vertices
    dc.w 1*4,5*4,6*4,2*4        ;Corners: 1-5-6-2

FrontPol2Table
    del  Pyramid            ;pyramid is in front of polygon 2.

poly3
    dc.w 0
    dc.b 13,-1                  ;Color 13, back side invisible
    del  0
    dc.w -1
    del  0
    dc.w 3                       ;4 vertices
    dc.w 5*4,4*4,7*4,6*4         ;Coordinate 5-4-7-6

poly4
    dc.w 0
    dc.b 14,-1                   ;Colour 14, reverse side invisible
    del  0
    dc.w -1
    del  0
    dc.w 3                       ;4 vertices
    dc.w 3*4,2*4,6*4,7*4         ;Coordinate 3-2-6-7

poly5
    dc.w 0
    dc.b 15,-1              ;Colour 15, backside invisible
    del  0
    dc.w -1
    del  0
    dc.w 3                  ;4 vertices
    dc.w 4*4,0,3*4,7*4      ;Coordinate 4-0-3-7
```

These were the polygon structures for the cube. Note especially **poly2**, where there is a reference to the pyramid. If poly2 is visible, the pyramid should be drawn after the cube, otherwise it should be drawn before. Study this example carefully so that you are sure how it works.

Now comes the rest of the polygon structures:

```
poly6
    dc.w 0
    dc.b 16,-1              ;Front: 16, back invisible
    del  0
    dc.w -1
```

```
     del  0
     dc.w 2                 ;3 corners
     dc.w 8*4,12*4,11*4     ;coordinates 8-12-11

poly7
     dc.w 0
     dc.b 17,-1             ;Front 17, back invisible
     del  0
     dc.w -1
     del  0
     dc.w 2                 ;3 corners
     dc.w 9*4,12*4,8*4      ;coordinates 9-12-8

poly8
     dc.w 0
     dc.b 18,-1             ;Colour 18, back side invisible
     del  0
     dc.w -1
     del 0
     dc.w 2                 ;3 corners
     dc.w 12*4,9*4,10*4     ;Coordinates 12-9-10

poly9
     dc.w 0
     dc.b 19,-1             ;Colour 19, backside invisible
     del  0
     dc.w -1
     del  0
     dc.w 2                 ;3 corners
     dc.w 11*4,12*4,10*4 ;Coordinates 11-12-10
```

```
poly10

     dc.w 0

     dc.b 20,-1          ;Colour 20, backside invisible

     del  0

     dc.w -1             ;No front faces is not quite true - see explanation below

     del  0

     dc.w 3              ;4 corners

     dc.w 9*4,8*4,11*4,10*4    ;Coordinate 9-8-11-10
```

When we say that the cube is not completely in front of face 10, this is actually wrong, but it is still not necessary to give this up in the poly10 structure.

It is possible to determine which of the two sub-objects should be drawn first completely by using surface 2. It is therefore of no consequence to determine the same thing a second time using a different face. A golden rule is therefore that it has no meaning to have such references back to sub-object 0, i.e. the start sub-object. Since all sub-objects must be sorted with respect to the first sub-object, it just causes extra work to figure this out for each sub-object you get to.

Now we've actually made the whole object! That wasn't so hard, was it? Anyway: study this example carefully make sure you understand everything before moving on to the next example.

The above example is in two versions on the disk, named "obj1nb.s" and "obj1b.s" respectively. The first is an object on a solid background, while the second loads a 32-color image and puts it behind it.

Star routine

Now is the time to tackle a slightly more advanced object. In Figure 25 you can see which object we are going to make. It's not that much more difficult, but it contains 7 sub-objects, one of which is a phantom sub-object.
As usual, we start with the coordinate table:

```
StarCoords
     dc.w 0,-200,0
     dc.w -50,-50,-50
     dc.w 50,-50,-50
     dc.w 50,-50,50
     dc.w -50,-50,50
     dc.w 0,0,-200
     dc.w 200,0,0
     dc.w 0,0,200
     dc.w -200,0,0
     dc.w -50,50,-50
     dc.w 50,50,-50
     dc.w 50,50,50
     dc.w -50,50,50
     dc.w 0,200,0
```

In total there are 14 coordinates. As usual, we set aside space for calculation coordinates and plotting coordinates:

```
StarCalcCoords
     ds.w 14*3
StarDrawCoords
     ds.l 14
```

Before we start on the main object structure, we need to figure out how many sub-objects there should be. There are 6 that immediately stand out: Each star branch is a sub-object. The problem is that the stars do not immediately contain any side faces that allow them to be sorted in relation to each other. We then have two options; either to include an additional phantom sub-object or to include phantom faces in the sub-objects.
Both methods work, and here we will look at phantom surfaces. We therefore get 6 sub-objects in total. On the disk you will also see how this can be done with a phantom sub-object - more on this later.
In Figure 25 we have illustrated how to place a phantom tile on one of the vertices - This should be done in a similar way for all the other sides, as the object is completely symmetrical. It is therefore also irrelevant which of the roofs is chosen as object 0 - we have chosen the top one. Each of the sub-objects gets the phantom face on a total of 5 faces. We are now ready to start on the main object structure:

```
star
     dc.w 0,0,0,0,0        ;The first 5 words are only used internally in the
                           recording routines
     dc.w 0,0,0,0,0,0      ;x, y and z coordinates and rotation angles...
     dc.w 13               ;14 coordinates in total
     del  StarCoords                ;Address of coordinate table
     del  StarCalcCoords            :Address of the calculation coordinate table
     del  StarDrawCoords            ;Address of the drawing coordinate table
     del  StarSubTable              ;Address of the list of all sub-objects
     dc.w 5                ;6 sub-objects (number minus 1 must be given here!)
```

Now it's the turn of the sub-objects. First, however, we need a table of all the sub-objects:

```
StarSubTable

        del  StarUp,StarLf,StarFr,StarRg,StarBk,StarDn
```

In the figure you can see that we have given the sub-objects the following name and number:

```
        0:Up, 1:Lf, 2:Fr, 3:Rg, 4:Bk, 5:Dn
```

Here we have also added "Star" in front of all the names so that it is clear that they belong to the "Star" main object.

```
StarUp

        dc.w 0

        dc.w 4              ;5 faces in each "branch"

        del  upPoly0,upPoly1,upPoly2,upPoly3,upPolyBlack
```

As you can see here, you can give the polygons any names you want. It is often best to give them a name so you can see which sub-object they belong to.

```
StarLf
        dc.w 0
        dc.w 4
        del  lfPoly0,lfPoly1.lfPoly2,lfPoly3,lfPolySort
StarFr
        dc.w 0
        dc.w 4
        del  frPoly0,frPoly1,frPoly2,frPoly3,frPolySort
StarRg
        dc.w 0
        dc.w 4
        del  rgPoly0,rgPoly1,rgPoly2,rgPoly3,rgPolySort
StarBk
        dc.w 0
        dc.w 4
        del  bkPoly0,bkPoly1,bkPoly2,bkPoly3,bkPolySort
StarDn
        dc.w 0
        dc.w 4
        del  dnPoly0.dnPoly1,dnPoly2,dnPoly3,dnPolySort
```

It was that simple. However, now we are missing all the polygon structures. For this object, this is where the most work lies: there are 6*4 = 24 different faces in this figure. To get the maximum number of colors, we give them different colors.

```
upPoly0
     dc.w 0
     dc.b 1-1          ;Front colour 1, back invisible.
     del  0            ;No surfaces in the same plane
     dc.w -1           ;No sub-objects in front of this page
     del  0
     dc.w 2            ;triangle
     dc.w 1*4,0,2*4 ;Coordinates 1-0-2

upPoly1
     dc.w 0
     dc.b 2-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 2*4,0,3*4 ;2-0-3


upPoly2
     dc.w 0
     dc.b VI
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 3*4,0,4*4 ;3-0-4


upPoIy3
     dc.w 0
     dc.b 4-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 4*4,0,1*4 ;4-0-l

upPolySort
     dc.w 0
     dc.b -1,-1
     del  0
     dc.w 4            ;5 sub-objects in front of this surface (all the others actually!)
     del  FrontUpTable
     dc.w 3      ;4 vertices
     dc.w 1*4,2*4,3*4,4*4

FrontUpTable
     del  StarLf,StarFr,StarRg,StarBk,StarDn
```

```
lfPoly0
     dc.w 0
     dc.b 5,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 8*4,1*4,9*4           ;Coordinates 8-1-9

lfPoly1
     dc.w 0
     dc.b 6-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 8*4,4*4,1*4           ;8-4-l

lfPoly2
     dc.w 0
     dc.b 7,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 4*4,8*4,12*4          ;4-8-12

lfPoly3
     dc.w 0
     dc.b 8,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 8*4,9*4,12*4          ;8-9-12

lfPolySort
     dc.w 0
     dc.b -1,-1
     del  0
     dc.w 3                     ;4 sub-objects in front of this surface.
     del  FrontLfTable
     dc.w 3                     ;4 vertices
     dc.w 1*4,4*4,12*4,9*4      ;l-4-12-9
```

```
FrontLfTable
     del  StarFr,StarRg,StarBk,StarDn

frPoly0
     dc.w 0
     dc.b 9,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 1*4,2*4,5*4              ;Coordinates 1-2-5

frPoly1
     dc.w 0
     dc.b 10,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 2*4,10*4,5*4             ;2-10-5

frPoly2
     dc.w 0
     dc.b 11,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 5*4,10*4,9*4             ;5-10-9

frPoly3
     dc.w 0
     dc.b 12,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 1*4,5*4,9*4              ;l-5-9

frPolySort
     dc.w 0
     dc.b -1,-1
     del  0
     dc.w 3                             ;4 sub-objects in front of this surface.
     del  FrontFrTable
     dc.w 3                             ;4 vertices
     dc.w 2*4,1*4,9*4,10*4              ;2-1-9-10
```

```
FrontFrTable
     del  StarLf,StarRg,StarBk,StarDn

rgPoly0
     dc.w 0
     dc.b 13,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 2*4,7*4,6*4              ;coordinates 2-7-6

rgPoly1
     dc.w 0
     dc.b 14,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 6*4,7*4,11*4             ;6-7-ll

rgPoly2
     dc.w 0
     dc.b 15,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 10*4,6*4,11*4            ;10-6-ll

rgPoly3
     dc.w 0
     dc.b 16,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 2*4,6*4,10*4             ;2-6-10

rgPolySort
     dc.w 0
     dc.b -1,-1
     del  0
     dc.w 3                        ;4 sub-objects in front of this surface.
     del  FrontRgTable
     dc.w 3                        ;4 vertices
     dc.w 3*4,2*4,10*4,11*4        ;3-2-10-11
```

```
FrontRgTable
     del  StarFr,StarLf,StarBk,StarDn

bkPoly0
     dc.w 0
     dc.b 17,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 3*4,4*4,7*4      ;coordinates 3-4-7

bkPoly1
     dc.w 0
     dc.b 18,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 7*4,4*4,12*4    ;7-4-12

bkPoly2
     dc.w 0
     dc.b 19,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 11*4,7*4,12*4   ;11-7-12

bkPoly3
     dc.w 0
     dc.b 20,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 3*4,7*4,11*4    ;3-7-11

bkPolySort
     dc.w 0
     dc.b -1,-1
     del  0
     dc.w 3                ;4 sub-objects in front of this surface.
     del  FrontBkTable
     dc.w 3                ;4 vertices
     dc.w 4*4,3*4,11*4,12*4     ;4-3-11-12
```

```
FrontBkTable
     del  StarFr,StarRg,StarLf,StarDn

dnPoly0
     dc.w 0
     dc.b 21,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 9*4,10*4,13*4  ;Coordinates 9-10-13

dnPoly1
     dc.w 0
     dc.b 22,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 10*4,11*4,13*4 ;10-11-13

dnPoly2
     dc.w 0
     dc.b 23,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 11*4,12*4,13*4 ;11-12-13

dnPoly3
     dc.w 0
     dc.b 24,-1
     del  0
     dc.w -1
     del  0
     dc.w 2
     dc.w 12*4,9*4,13*4  ;12-9-13

dnPolySort
     dc.w 0
     dc.b -1-1
     del  0
     dc.w 3              ;4 sub-objects in front of this surface.
     del  FrontdnTable
     dc.w 3              ;4 vertices
     dc.w 12*4,11*4,10*4,9*4  ;12-ll-10-9
```

```
FrontdnTable
     del  StarFr.StarRg.StarBk.StarLf
```

Notice that StarUp is never mentioned in the Front tables of the other sub-objects, although it could have been. This is what we explained earlier, it makes no sense to include it, since the drawing routine always starts at sub-object 0.

This was all that was needed to create the object in Figure 25.

On the disk there are two versions of this sample program. Both versions show the star on an image, but the star is made in two different ways.

In "startsides" is the example we just reviewed, where we used "phantom sides" to determine the sub-object sorting.

In "starfsub.s" we have used a "phantom sub-object". In this case it is actually the easiest. Except for the way the sub-objects are made, the two program examples are completely identical.

Double and triple buffering

So far, we have not talked about how to get smooth moves, although we have included it in most of the program examples. The method is called "double-buffering". In some of the examples, "triple-buffering" is even used.

This means that you have more than one screen in memory at the same time. While one screen is displayed, another is drawn in a buffer. This means that you never see the screen refresh, and you therefore avoid the annoying flickering. If you only use 2 screens, you may have to wait a while before you can start drawing. For maximum ' speed when drawing, you need to use 3 screens. This means: The first is displayed, the second is ready to be displayed while drawing on the third. If you have too little RAM, it can be a bit problematic with so many screens. 3 screens with 32 colours in 320x256 resolution take up 150 kB. If you then also need background graphics, an additional 50 kB is needed. That's a total of 200 kB - and all this is CHIPMEM, so it can be a serious problem.

When you're done drawing a screen, set the bitplane pointer registers the way you want them. What can cause delays is that the previous screen is displayed, and the electron beam can be anywhere on the new screen in the meantime. To achieve a smooth transition, wait until the electron beam has traversed the entire screen before switching to the screen now to be displayed. That is, if you have only two screens, wait for the electron beam to finish before you can start drawing again. With 3 screens, this is avoided.

Figure 26 shows how triple buffering works and why double buffering is not as fast.

To switch the screen that is displayed while the electron beam is not drawing the image on the screen, an interrupt is used: the "vertical blank" interrupt. It is called every time an image is finished on the monitor, and if you are quick you can change the screen registers before the screen recording starts again.

This interrupt also solves another problem for us. If you have a fast machine, A3000 etc..., you may run into the problem that the machine reaches to record the vector objects too fast; that is more than one object per screen refresh. The machine must therefore be forced to wait until the screen refresh is complete.

It's not as difficult as it sounds. If the objects are drawn that fast, you get smooth movements on the screen. It is impossible to make the movements look better. This can only be achieved for relatively simple objects when they are "far away" and therefore need to look small. By inserting this pause, the screen is never cluttered, no matter how fast the machine is.

We have created a program that shows such a movement. The source file is called "obj2ef.s", and in assembled form "obj2ef".

Animation example

So far we have only discussed how to draw simple objects, and in principle how to draw multiple objects simultaneously on the screen. However, it is a big step from this and to making a small "movie" or similar.

On the disk is the program "anim1.s", in executable form "anim1". It is a relatively short animation, which shows some of the possibilities you have.

Possible Improvements

There are a number of things that you can do to improve the routines that have been covered in this course. Most of the programs are made as simple as possible, so that they should be both clear and easy to understand. However, where necessary, we have optimised the routines to improve speed. However, there is still plenty of scope for improving the routines.

The vector routines on Interrupt

The biggest improvement comes from putting the vector routines on Interrupt. This saves you a lot of time by not having to wait for the blitter to finish.

You need to start by making your own "Interrupt handler". Each time the blitter finishes a task, it sets a bit in the INTREQ register. If the corresponding bit in the INTENA register is set, this will give an Interrupt that you can use. Instead of writing directly to the blitter registers in the line drawing routine and in the other drawing routines, you can store these values in a table. The interrupt handler should then read these values from the table and provide them to the blitter as soon as a new interrupt occurs.

This way you ensure that the blitter has something to do all the time while the processor is also working full time.

Create a routine for drawing triangles, squares, etc.

You can also optimise the routines by creating routines to draw triangles, squares, etc. A routine designed exclusively to draw triangles will draw a triangle somewhat faster than our general routine, which can handle polygons with over 50 vertices.

The disadvantage of this method is that the program will be somewhat larger and you will have to specify in the polygon structure which routine will draw the current polygon.

Simulation of light source on 3D objects

In many contexts it can be nice to simulate a light source on 3D objects. It should be relatively easy to add a light source to the routines in this course - they are prepared for it.

In the "calculation coordinates table" you have the coordinates by rotation, and these can be used to calculate the brightness of the individual surfaces. There are several good "light models" to choose from; the best known is probably "Phong-shading". However, it requires quite good mathematical skills to be able to program light sources.

Animation module

In order to make nice presentations, demos and the like, it is important to get movement in the objects. It pays to develop your own animation module, so you don't have to make hundreds of tables to add a little movement. Often you see effects where a block turns into a spaceship or similar. There may be 50 frames in such a "transformation", and the smartest thing is just to drop the block and the spaceship, and then make a routine that can output all the "intermediate shapes".

Animation structures

You should also consider making your own "animation structures", so that the animated objects are easy to keep track of. It should then not be an impossible task to make, for example, a robot that walks in a natural way.

Editor

Once you have designed a few 3D objects, you will soon find that you can use an editor specifically for 3D objects. That is, a program in which you can draw the objects, after which the program (and not you) will find the coordinates of the corners. This can be done in assembler, but it is also possible to do one in Basic. If you have an editor, it is much faster to make objects - especially if they are also to be animated. There is currently no such editor on the market, so it's something you can take as a challenge to make yourself.

Overview of programs on the floppy disk

Background.Raw	Background graphics for Anim1, StarFSide, StarFSub
LesMeg	Information about the disk not included in the disk itself course material.
MakeSinCosTable.bas	Basic program used to make SinCosTable.
MakeSinCosTable.bas.info	
MakeSRCSinCosTable.bas	Basic program used to make SinCosTable.s
MakeSRCSinCosTable.bas.info	
C (dir)	
Copy	
Delete	
Rename	
List	
Dir	
Type	
S (dir)	
startup-sequence	Can be booted directly from the floppy.
DEVS (dir)	
system-configuration	This sequence creates 80-character screen widths, to provide a nice screen.
Execs (dir)	Contains assembled files.
Anim1	Animation example.
Dots1	3D demo with joystick control.
Kube1	Building a cube.
Obj1B	Cube and pyramid on 32-colour background.
Obj1Nb	Same as Obj1b, but on a plain background.
Obj2Ef	Movement with smooth transitions.
StarFSide	Star with phantom face.
StarFSub	Star with phantom object.

Sources (dir)	The sources of the assembled files in execs.
Anim1.s	
Dots1.s	
Kube1.s	
Obj.s	Source code for routines used in other programs.
Obj1B.s	
Obj1Nb.s	
Obj2Ef.s	
Poly.s	Source code for filling, cropping and copying.
Skeleton.s	Source code with a basic structure like others program examples are built over.
StarFSide.s	
StarFSub.s	
Includes (dir)	
Custom.i	List of hardware registers.
Line.s	Line drawing routines. Contains the routines: draw_line Draws a line draw_Xline Draws a line without the first dot init_line Puts the blitter into line drawing mode.
Rotate.s	Rotate about 3 axes (source code).
sincostable	Sine/Cosine table (binary file).
SinCosTable.s	Source code for sincostable.
SEKA (dir)	Contains the program examples written for K-SEKA.
Sources (dir)	
Anim1.s	
Dots1.s	
Kube1.s	
Obj.s	
Obj1B.s	
Obj1Nb.s	

Obj2Ef.s

Poly.s

Skeleton.s

StarFSide.s

StarFSub.s

Includes (dir)

Custom.i

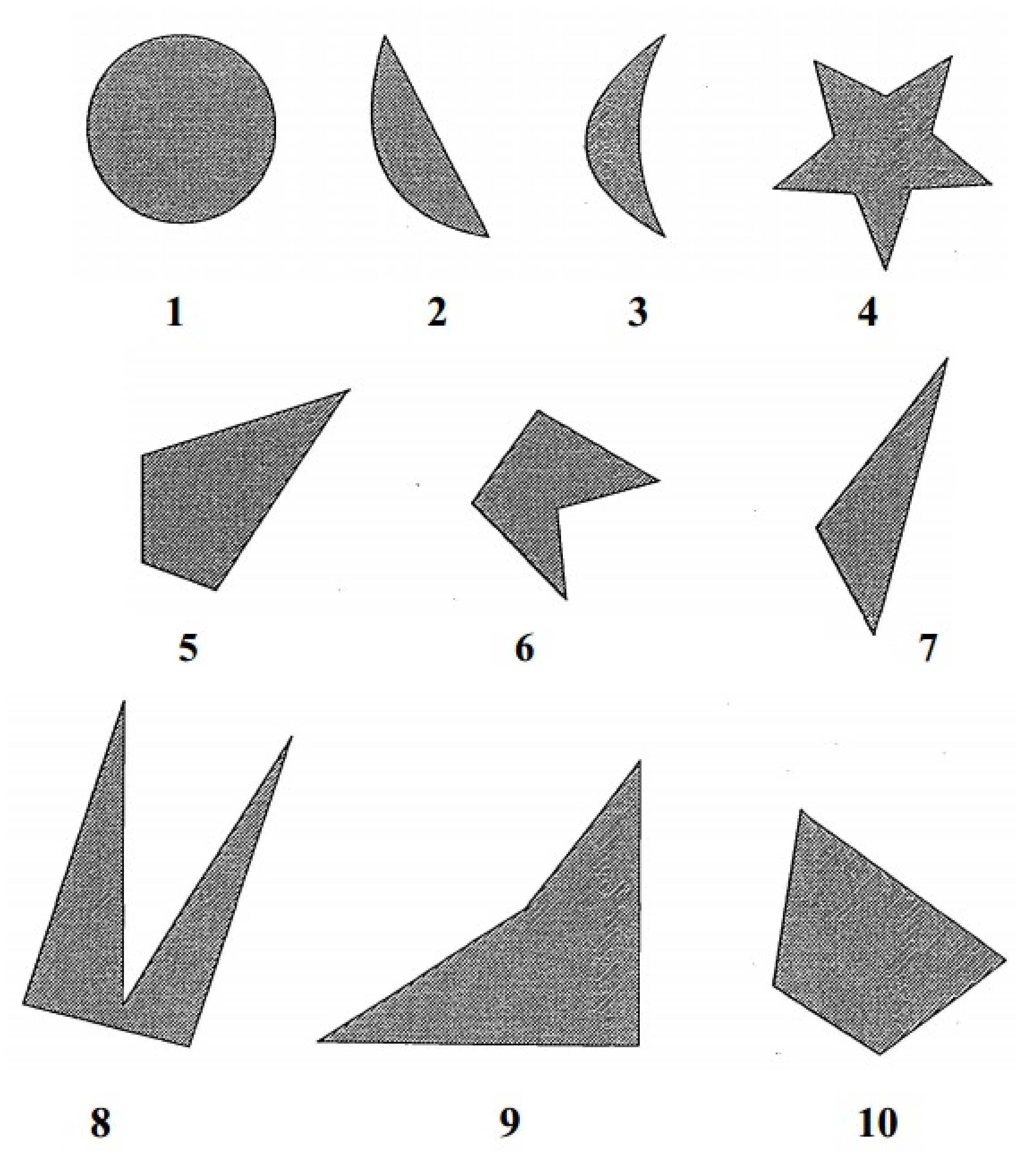
Line.s

Rotate.s

SinCosTable

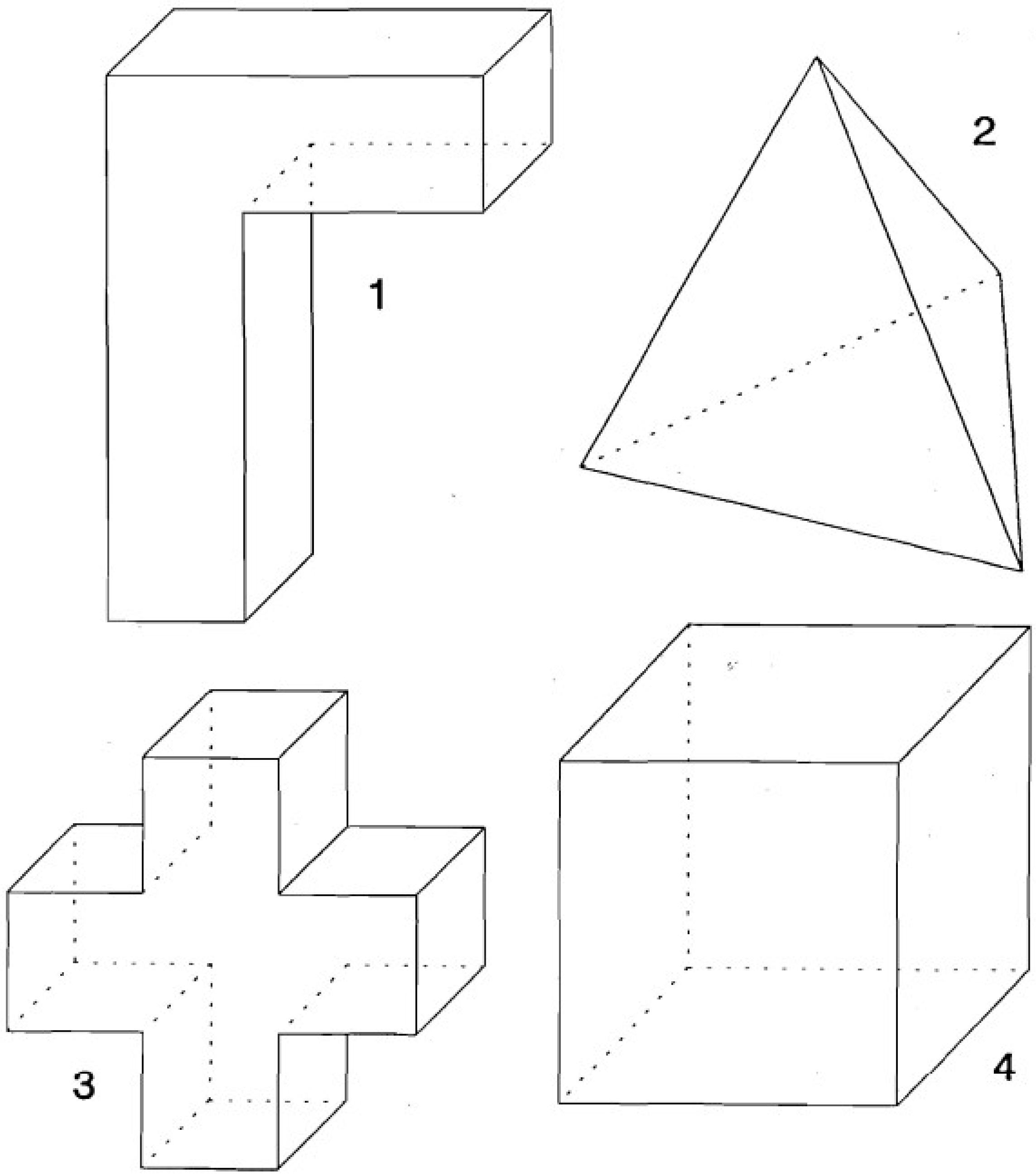
SinCosTable.s

Task1



Which of these object files are concave?

Task 2

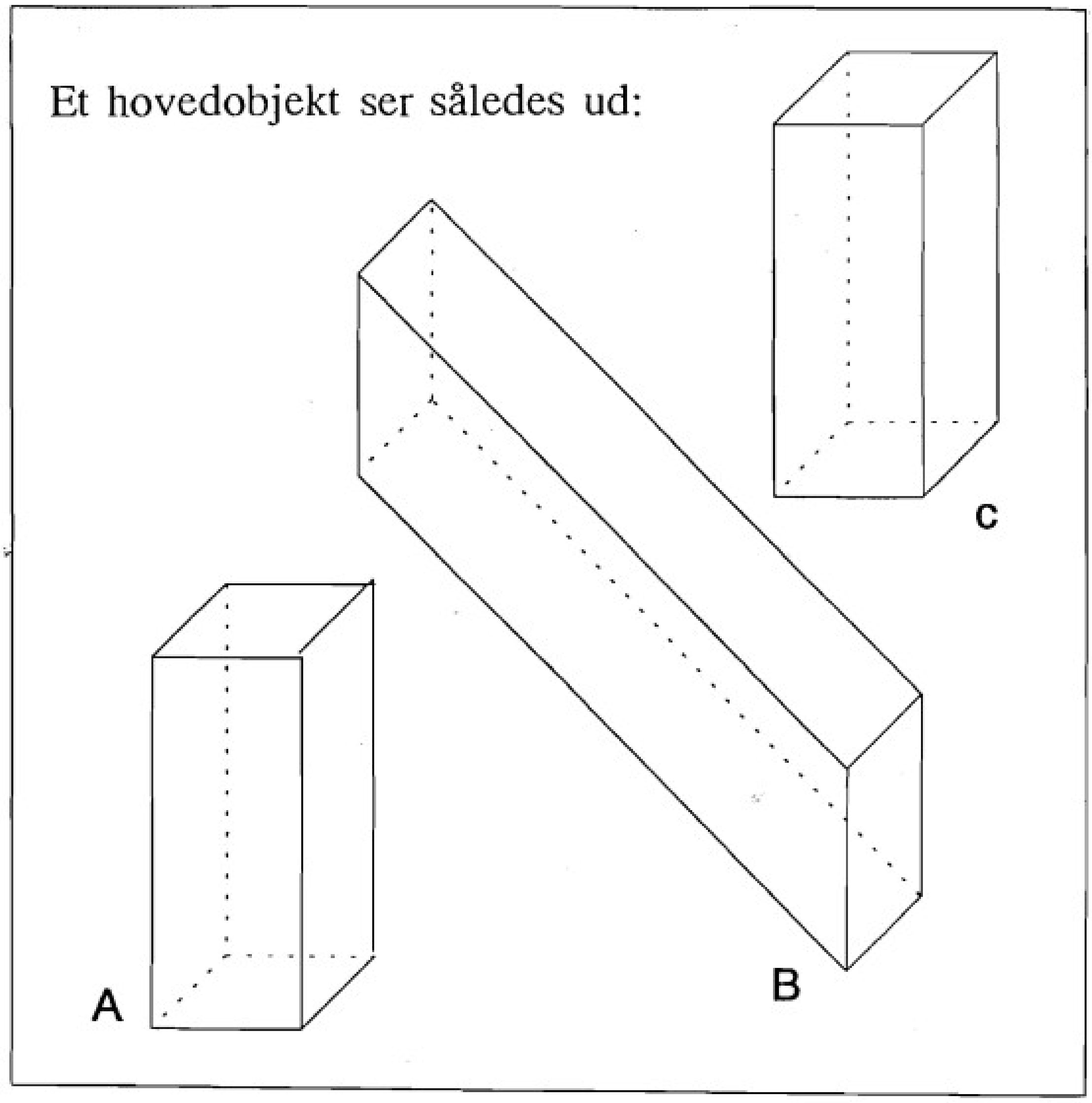


Which of these objects can be a sub-object?

How will you divide the other objects to get as few objects as possible?

Task 3

A main object looks like this:



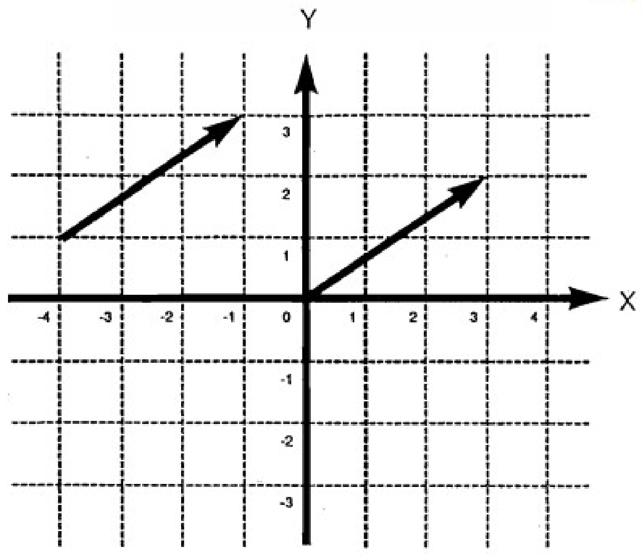
The objects are named A, B and C.

Which of the sub-objects can be sub-object 0 in the main body. Think carefully!

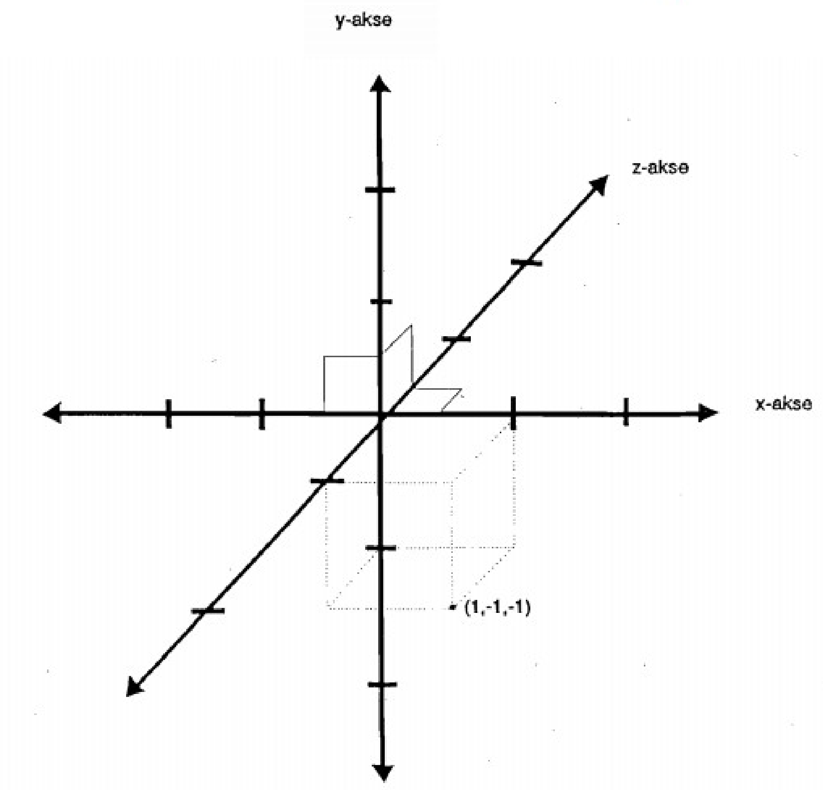
Review the checklists, both for the main and the sub-objects.

Give a reason for your answer.

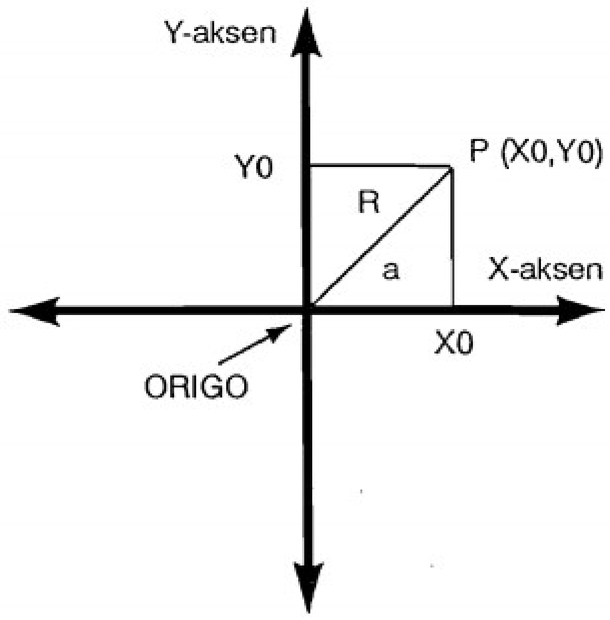
Figur 1A



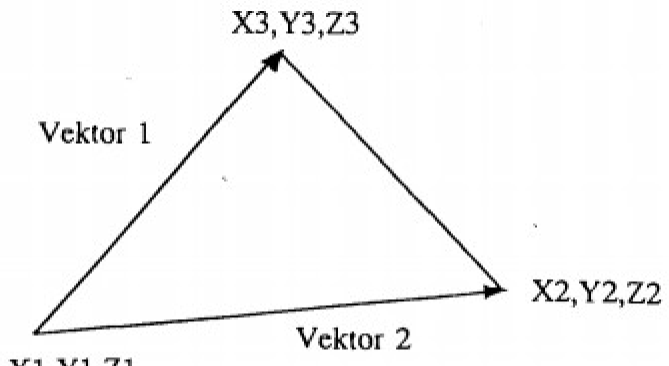
Figur 1B



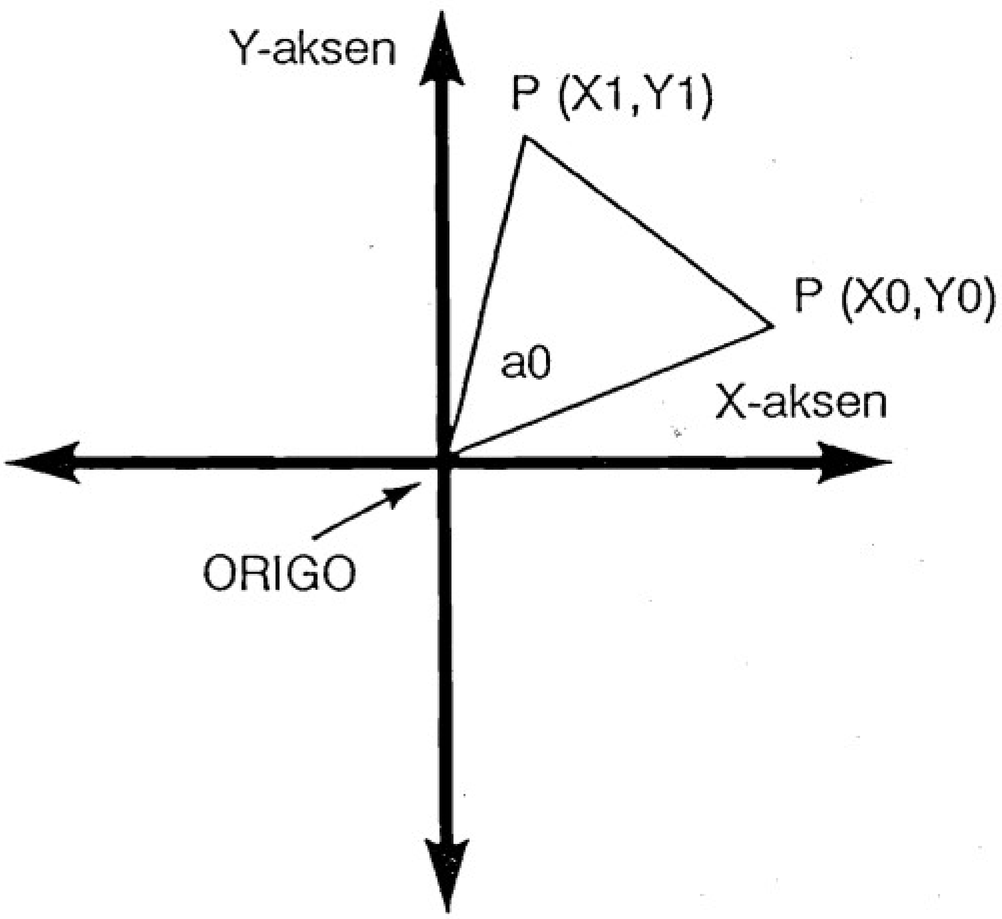
Figur 2



Figur 3

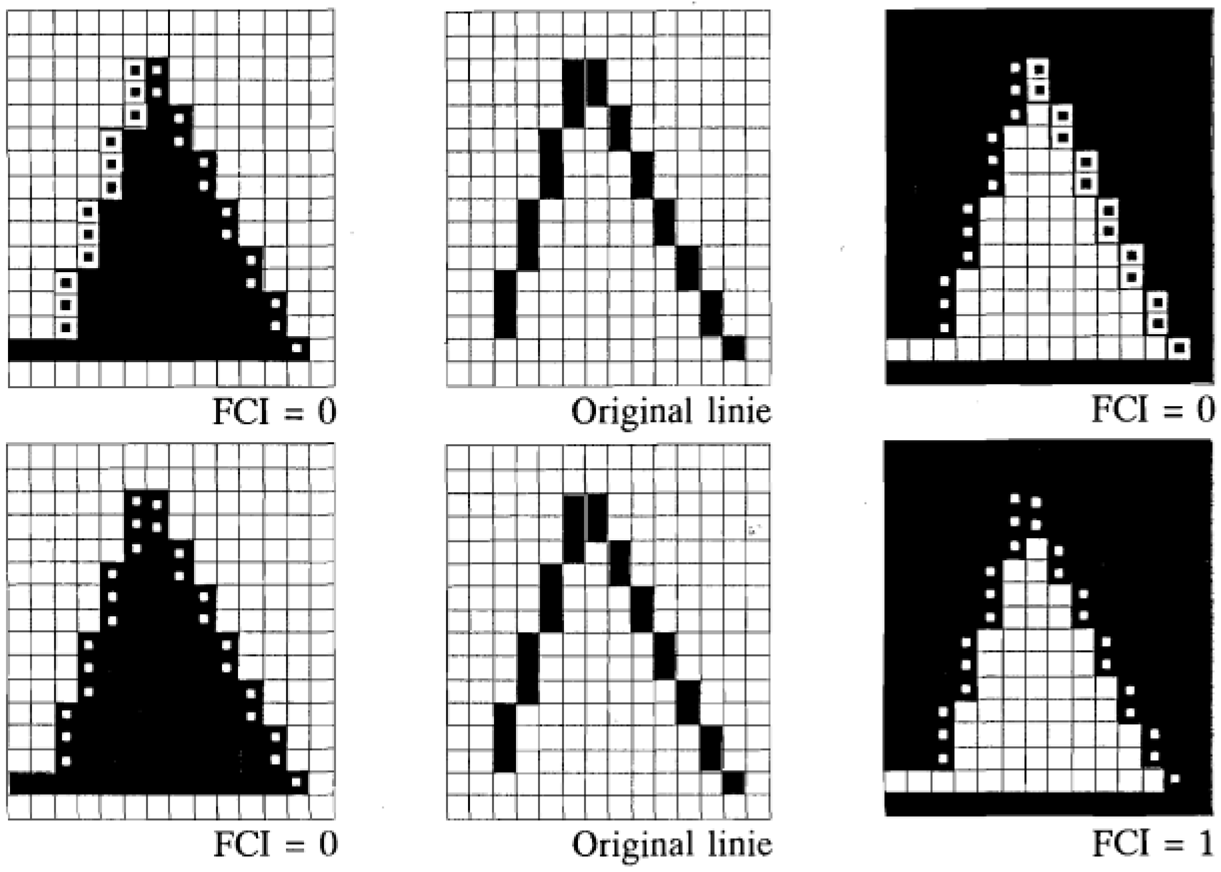


Figur 4



Figur 5

Blitterens to udfyldnings-modes



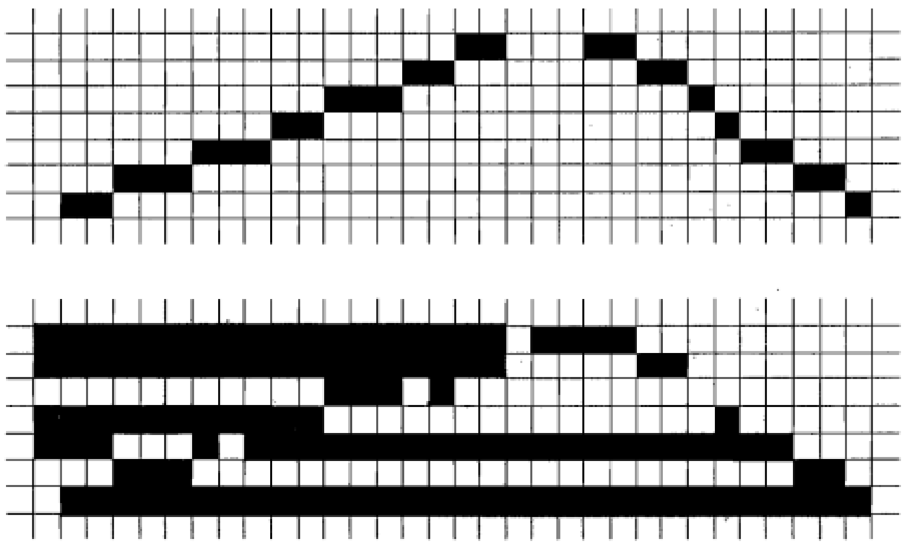
The top three figures are in EFE and the bottom three are in IFE.

Here you see the 4 different ways that the shape can be filled in.

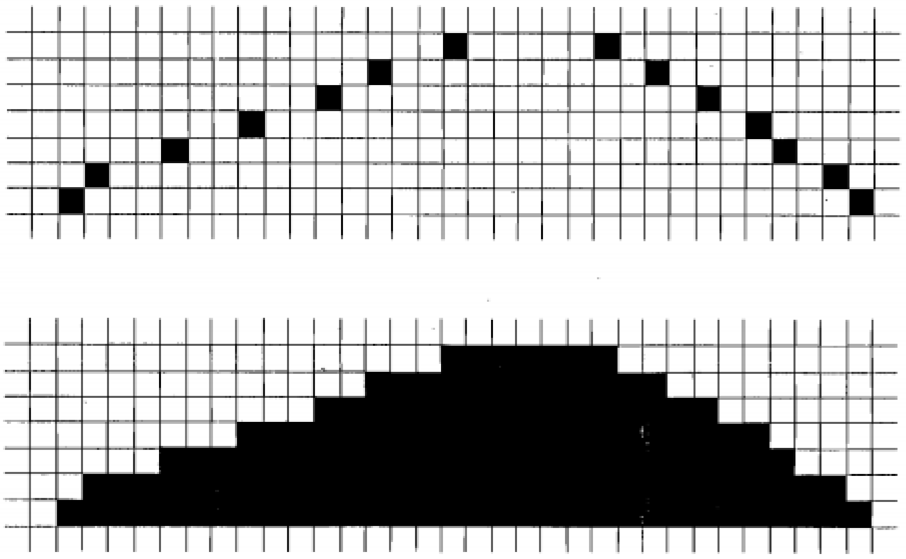
Each square represents a pixel. The small squares inside some of the panes shows where the lines were before the fill.

Also notice that the FCI is reloaded for each line, so that the error caused by missing pixels is minimised.

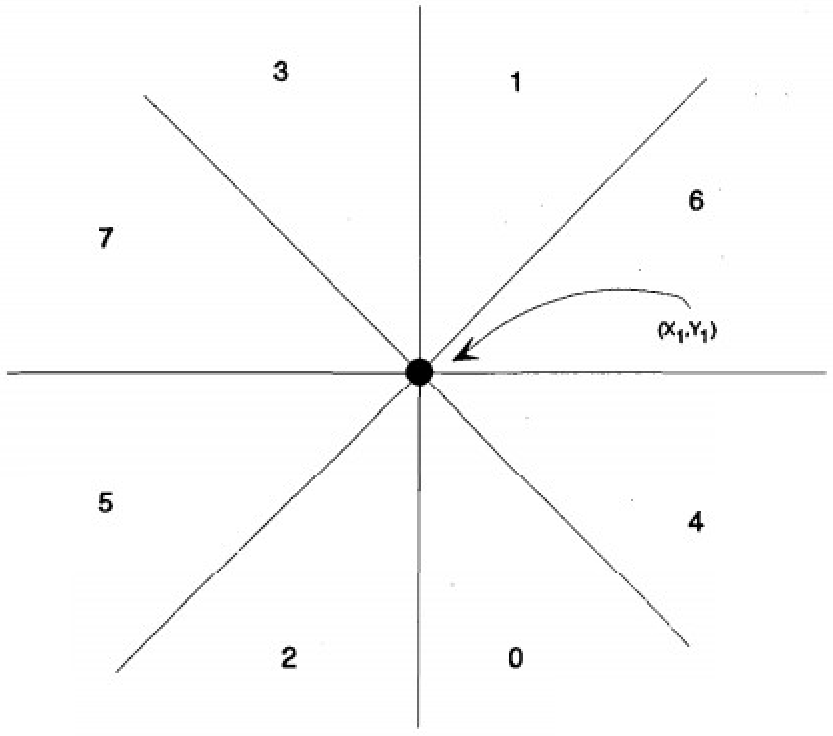
Blitter's two Linetypes


Figur 6A

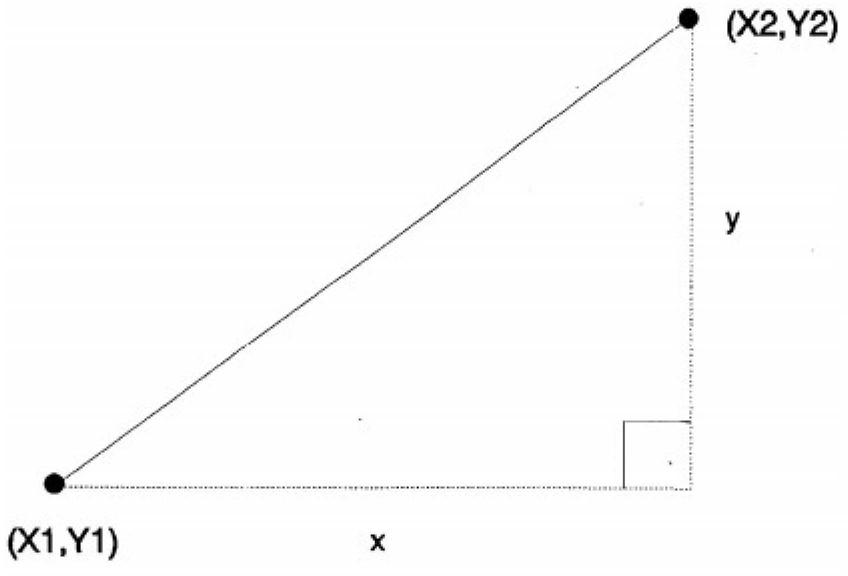
This is a normal continuous line, as you can see it gives the ugly error when the object is filled.


Figur 6B

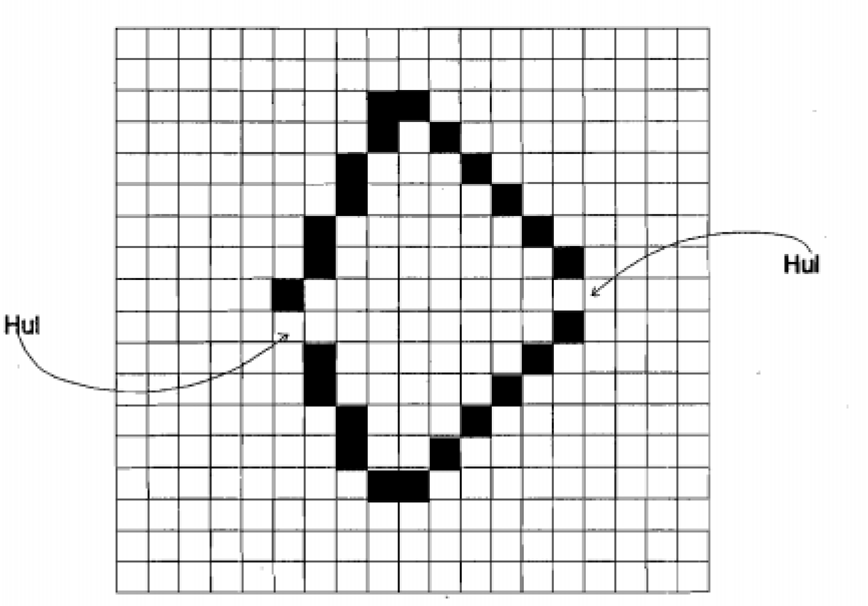
This is a special type of line where you only have to draw one pixel per horizontal line. The the filling works perfect. This line type is only used for lines around one object to be filled with colour.



Figur 7

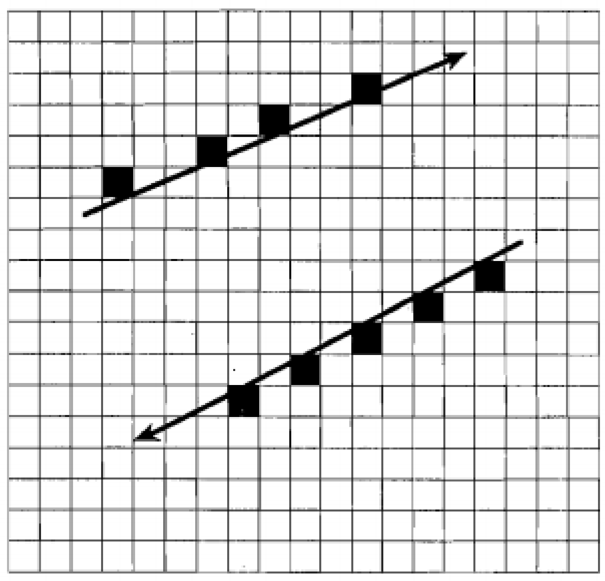


Figur 8



Figur 9

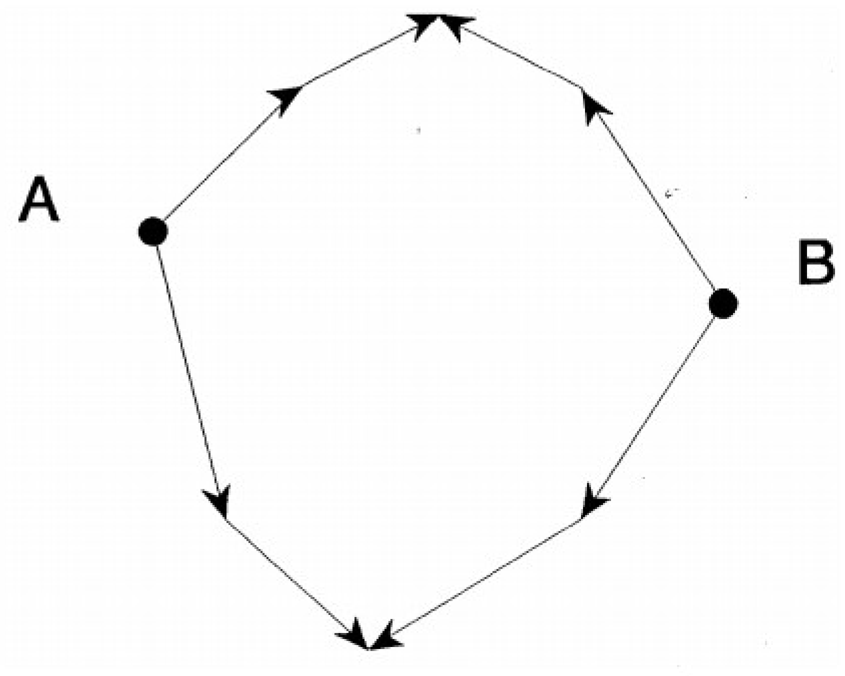
Ugly holes appear in some of the corners if no special lines are used.



Figur 10

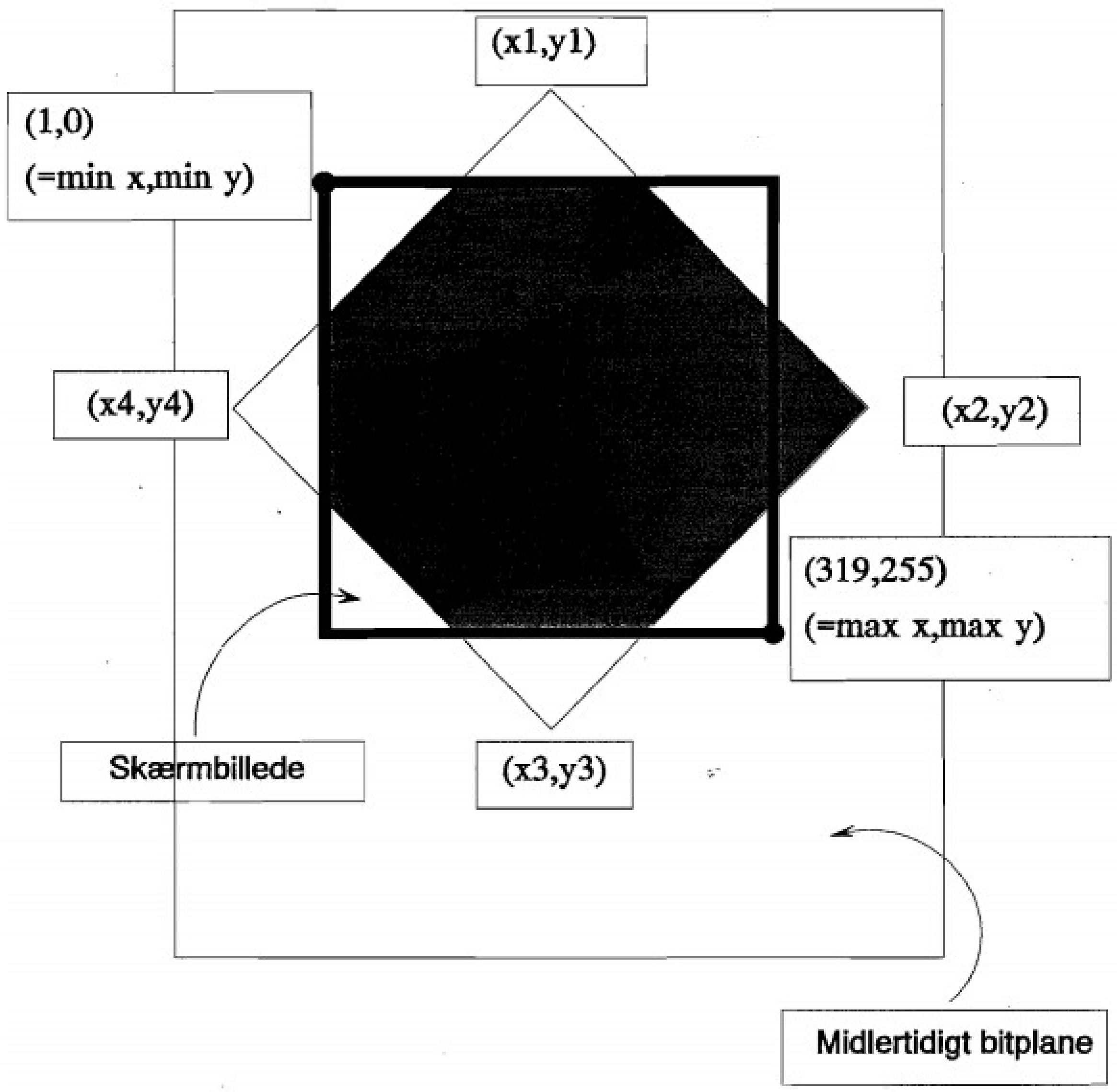
A line drawn upwards to the right will appear further to the left than one drawn downwards to the left.

Figur 11



If the lines are drawn around the object in this way, the polygon becomes as large as possible. This avoids holes between the surfaces.

KLIPNING **Figur 12**



Temporary bitplane

Here we set min x, min y and max y for a lo-res screen (320 x 256 pixels).

On the left side, (min x – x4 + 15)>>4 words must be cut away.

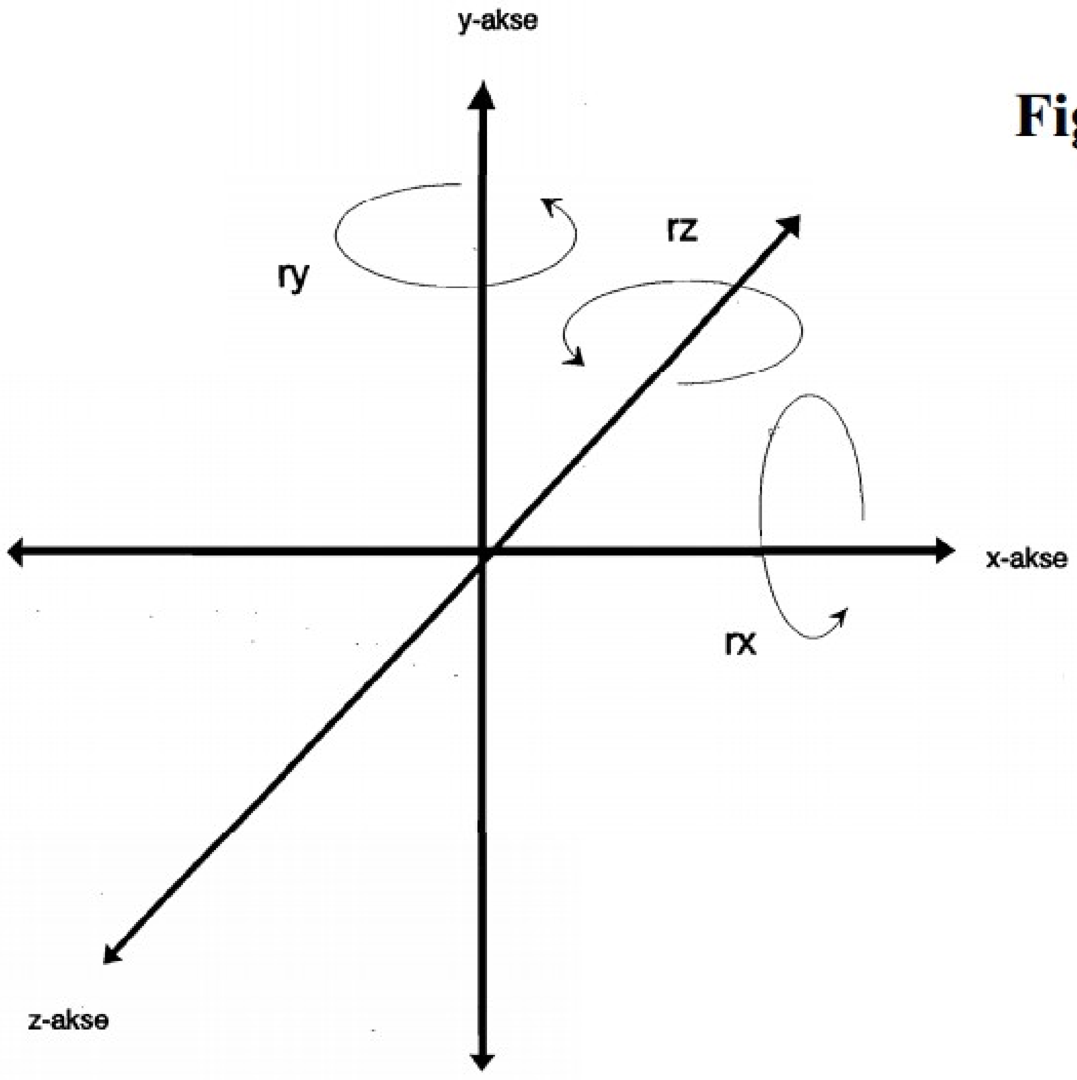
At the top (min y - y1) lines must be cut away.

At the bottom (y3 – max y) lines must be cut away.

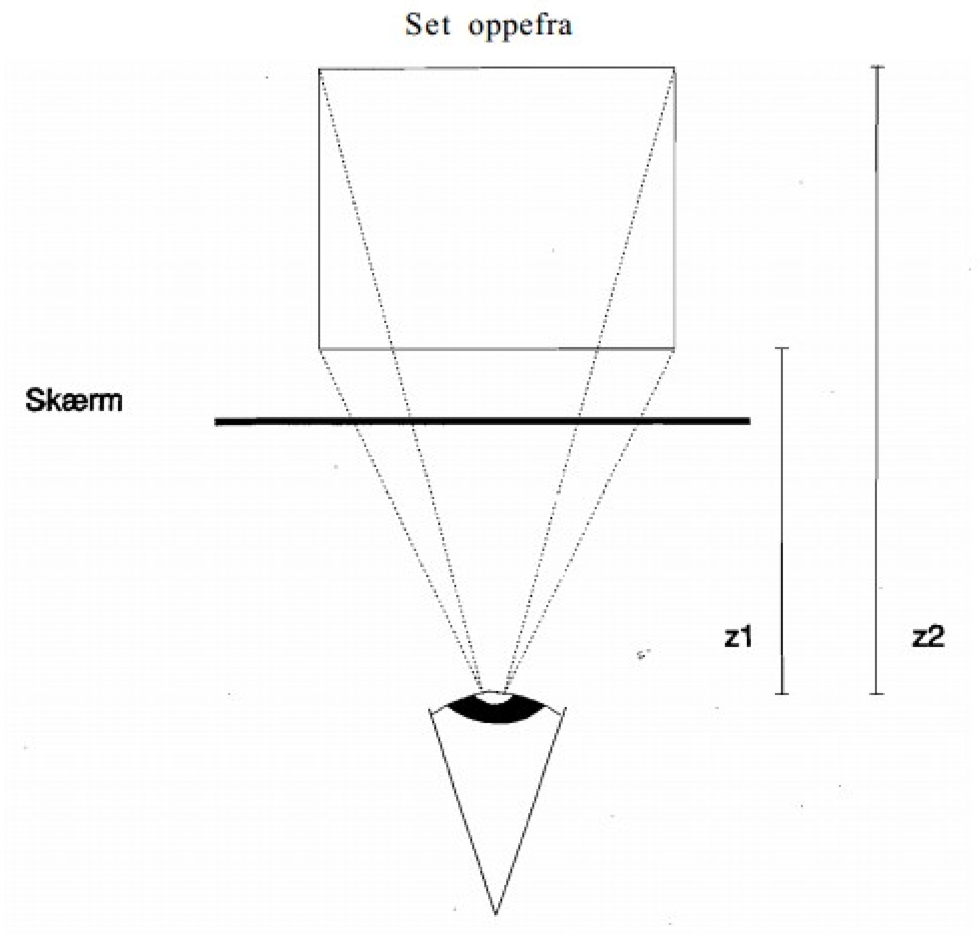
On the right side (x2 – max x + 15) >>4 words must be cut away.

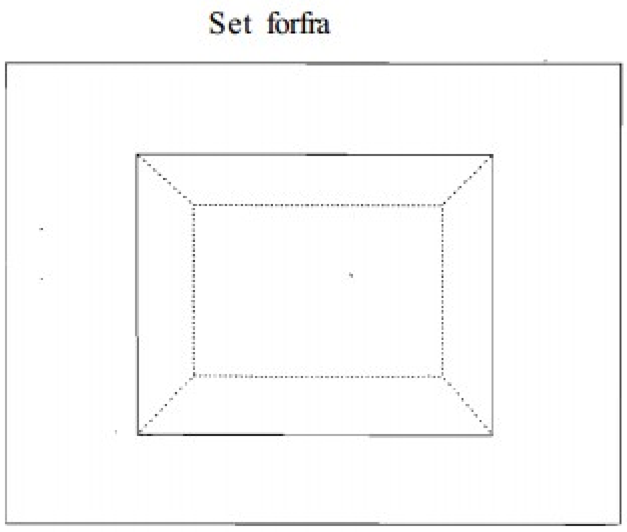
Only the shaded area of the figure needs to be filled. The area that is not filled will not be copied to the screen.

Figur 14

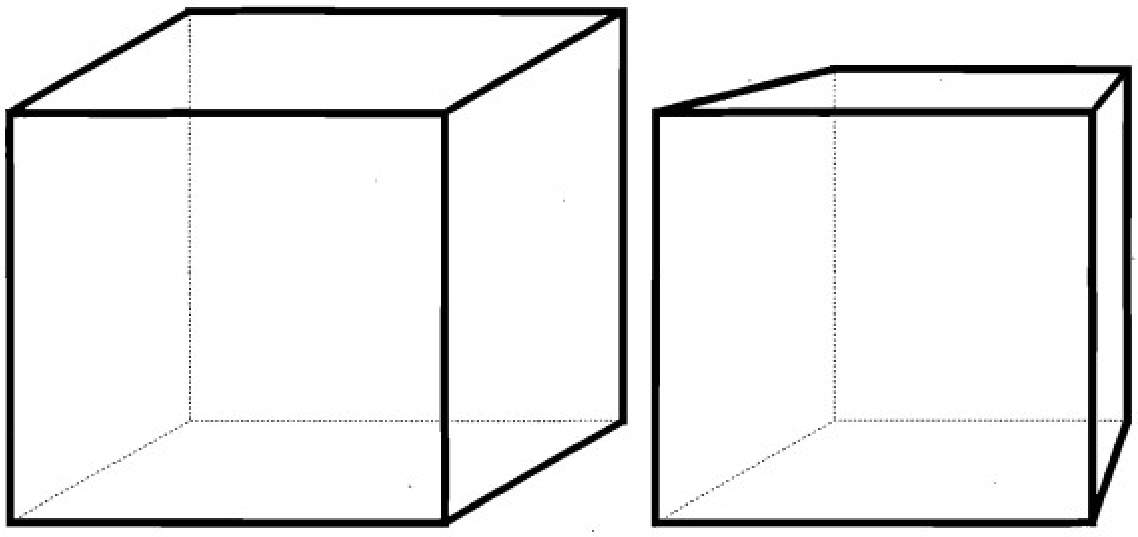


Figur 15



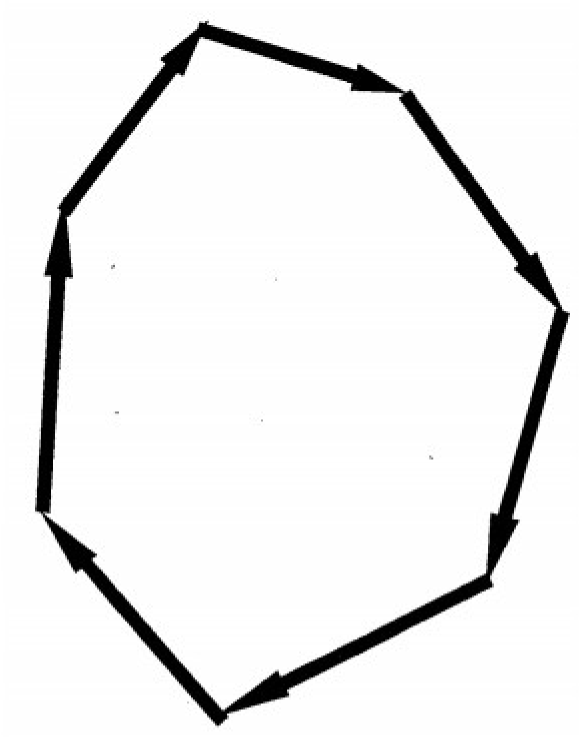


Figur 16



Cube drawn without and with perspective.

Figur 17



Clockwise

Figur 18A

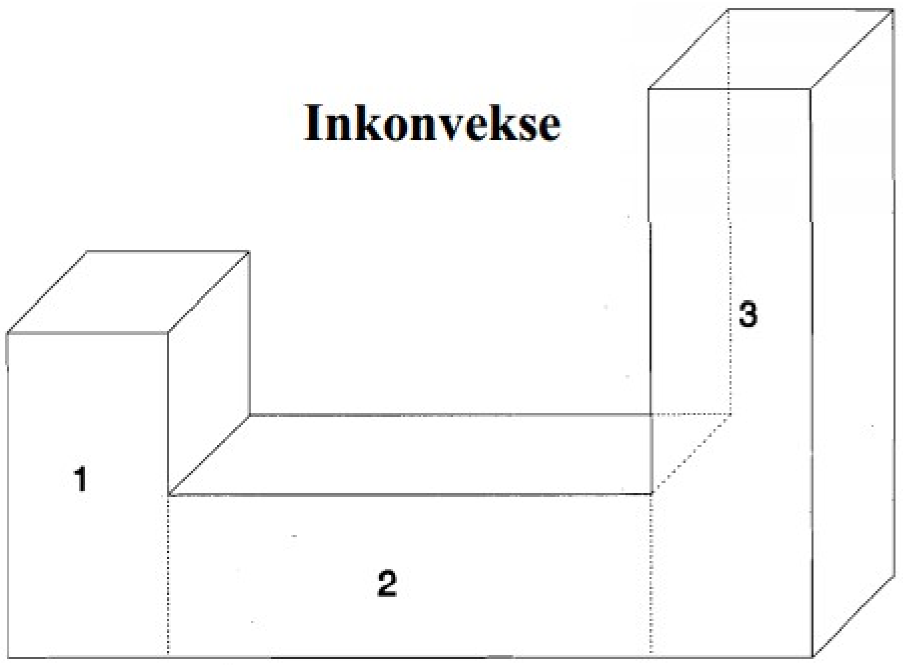
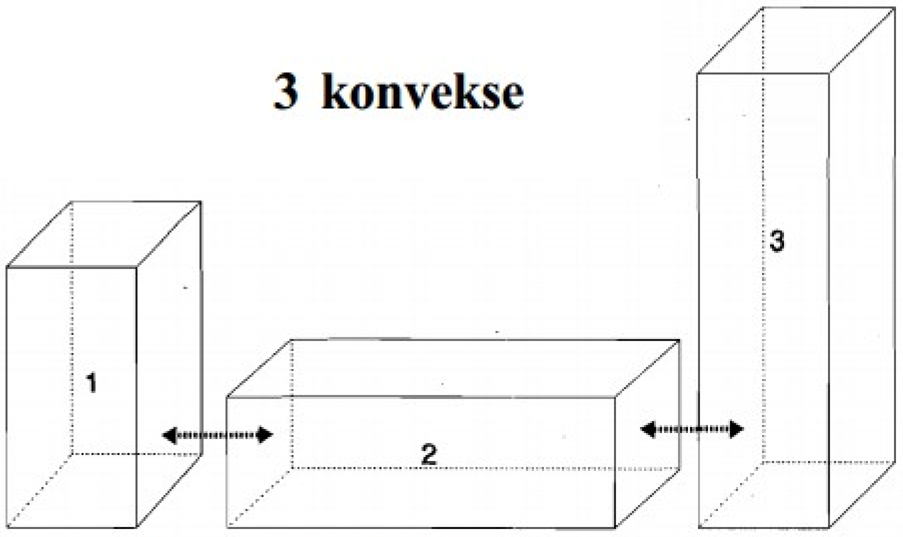


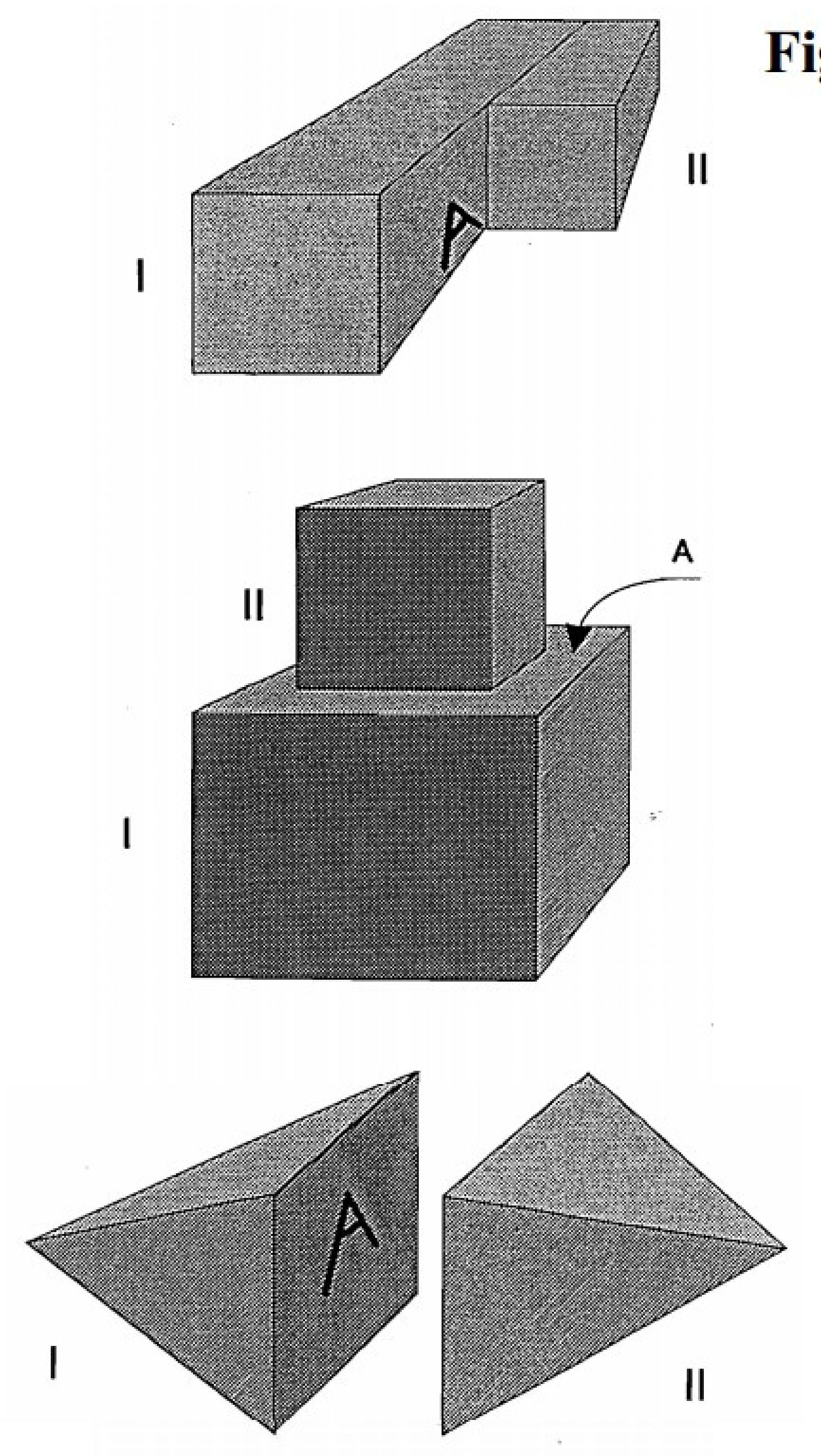
Figure 18b is assembled into figure 18a.

Figur 18B



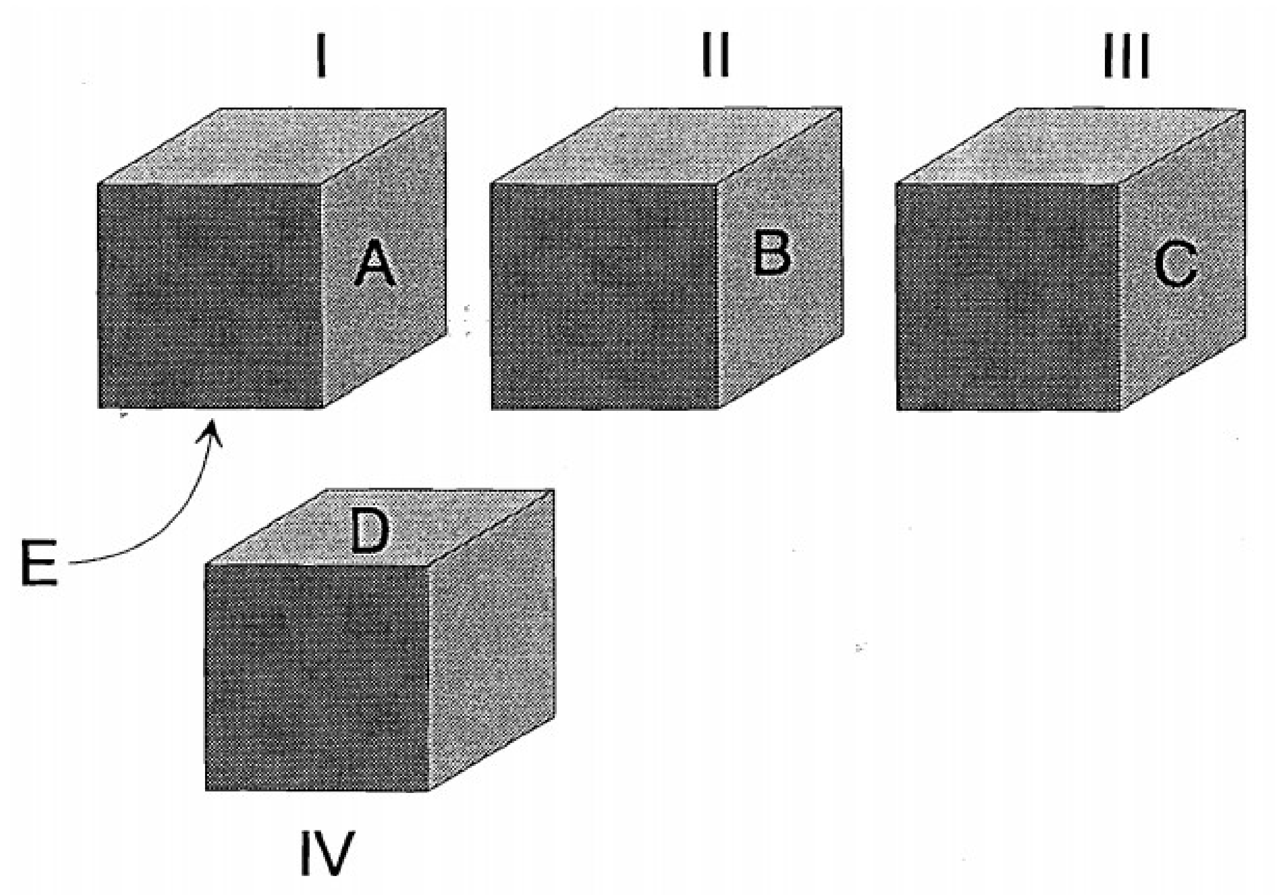
figur b samles til figur a

Figur 19



Here are three different objects. For each one, when surface A is visible, the object I comes in front of object II.

Figur 20



Above is shown an object which consists of 4 sub-objects.

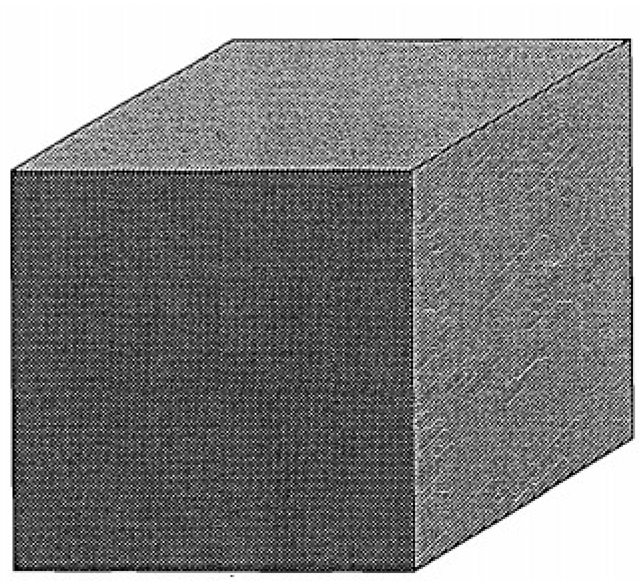
Sub-objects II and III lie to the far right of surface A.

Sub-object III is also to the far right of surface B.

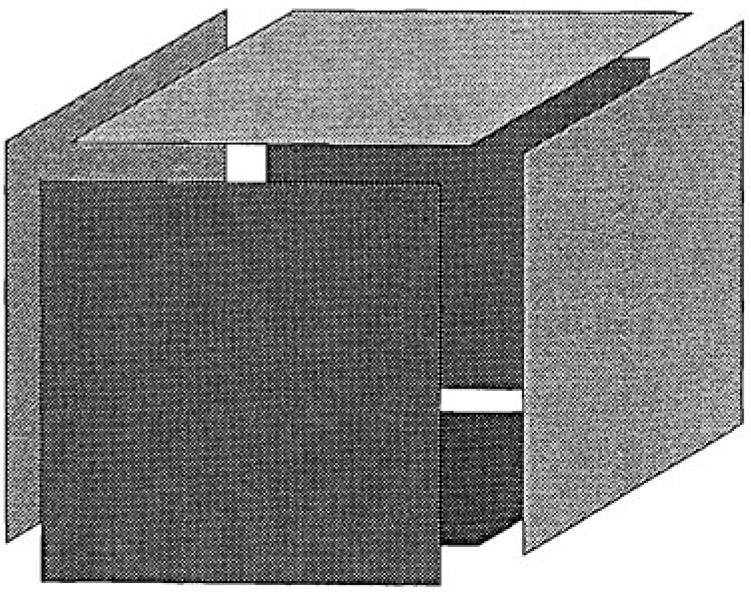
Sub-object IV cannot be sorted using these surfaces in contrast, the sub-objects I, II and III lie completely on the upper side of surface D.

Correspondingly, sub-object IV lies entirely on the underside of surface E.

Figur 21A



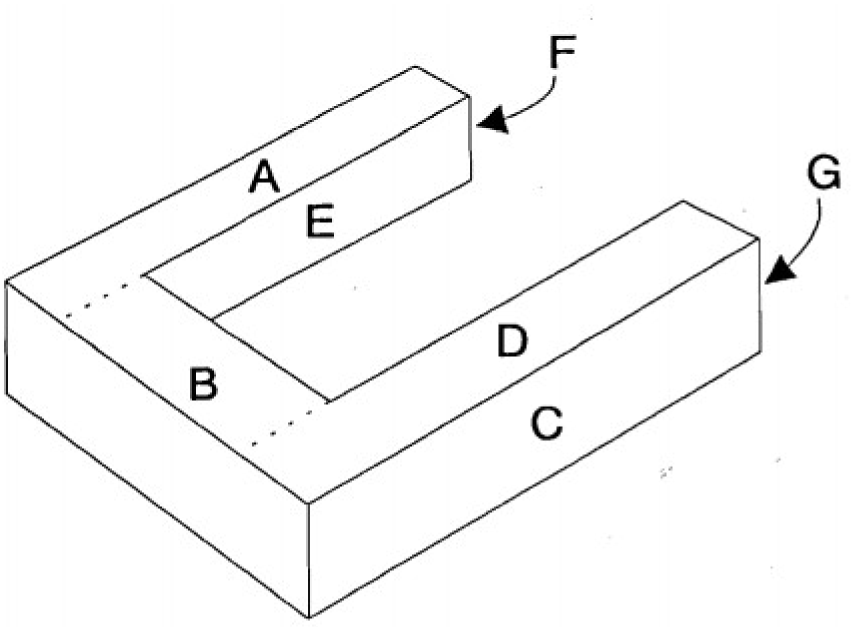
Figur 21B



Polygon in the same plane

Figur 22

POLYGON I SAMME PLAN

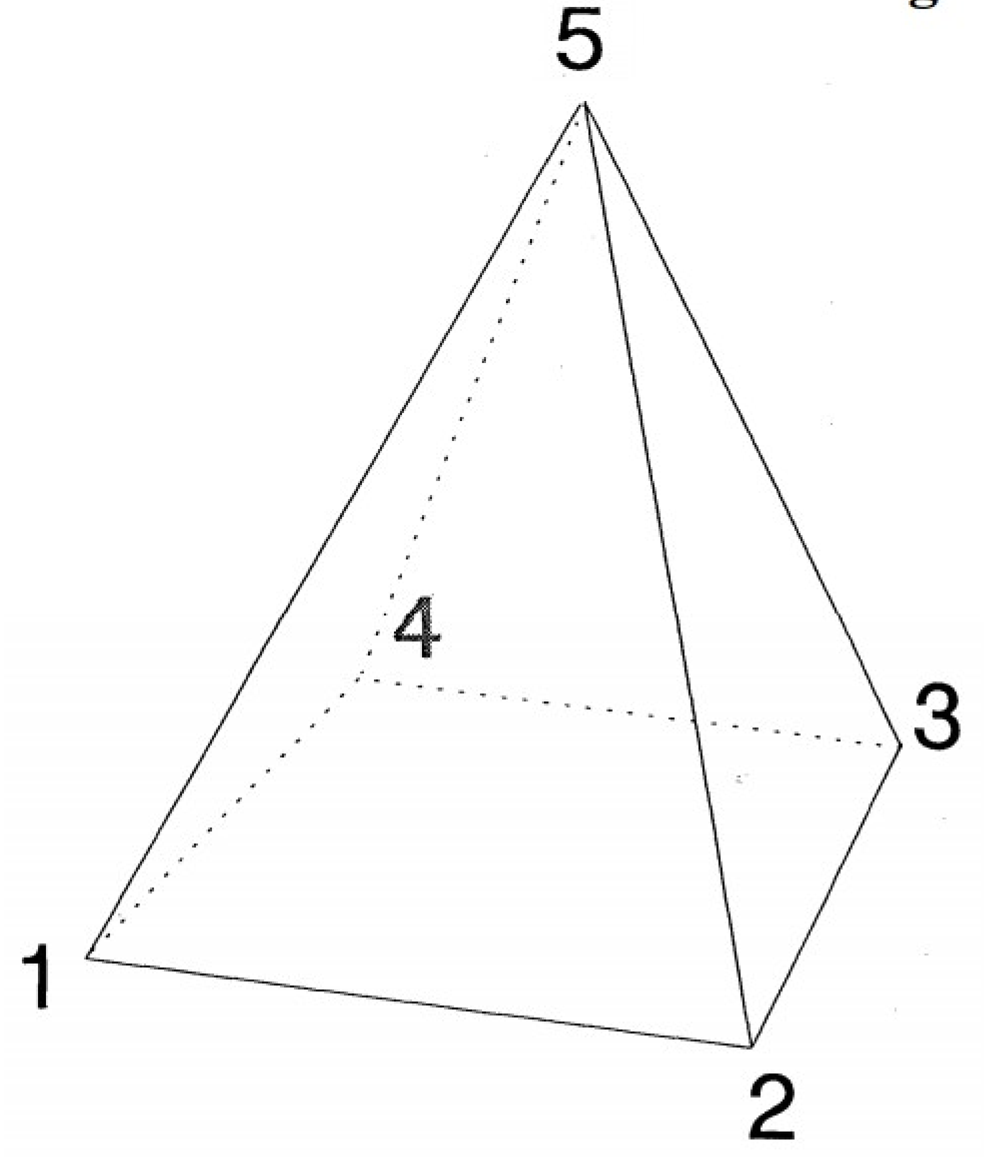


A, B and D are in the same plane.

F and G are in the same plane.

E and C are parallel but not in the same plane.

Figur 23

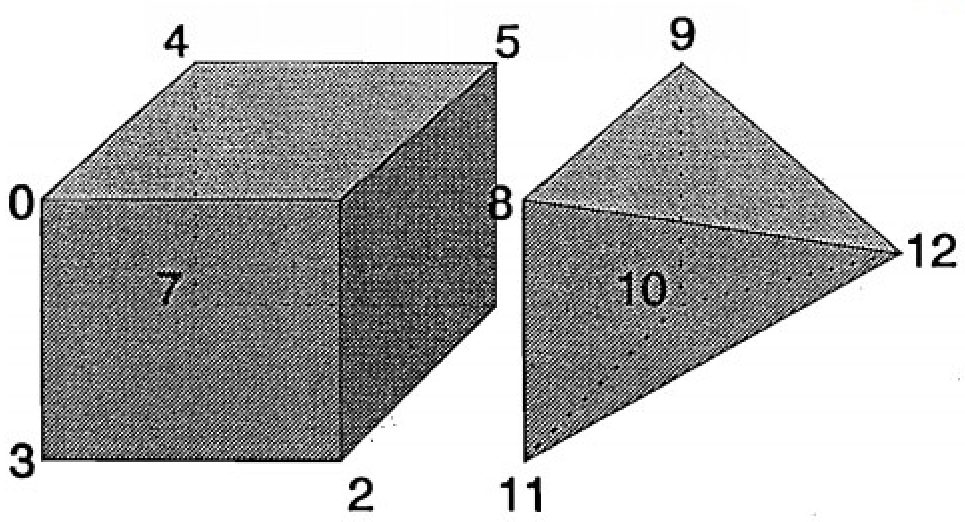


When you only provide the coordinates for each surface, it is very important to list them in the correct order.

The corners of the figure must have its vertices labelled in the correct order, eg clockwise.

The coordinates of the corners must be entered in a clockwise direction when you look at each surface from the outside of the object. In this pyramid, the right row becomes the following corners: 1-5-2, 2-5-3, 3-4-5, 4-5-1 and 1-2-3-4.

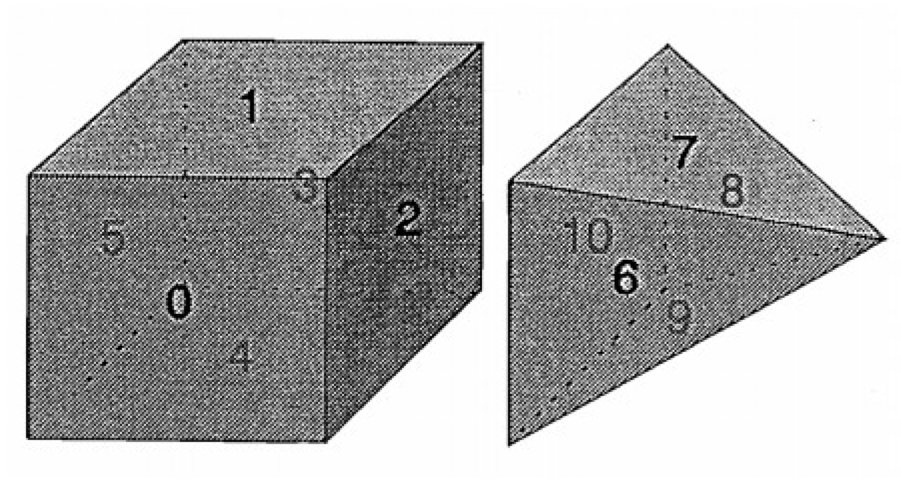
Figur 24



Coordinates of the corners

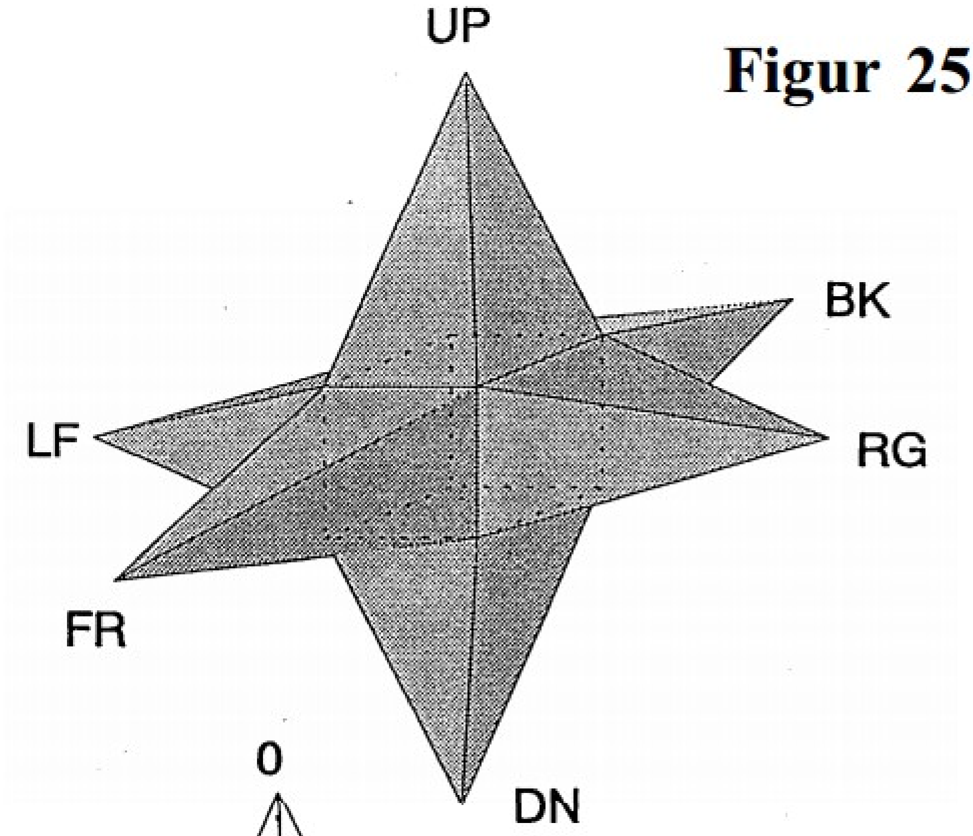
HJØRNERNES KOORDINATER

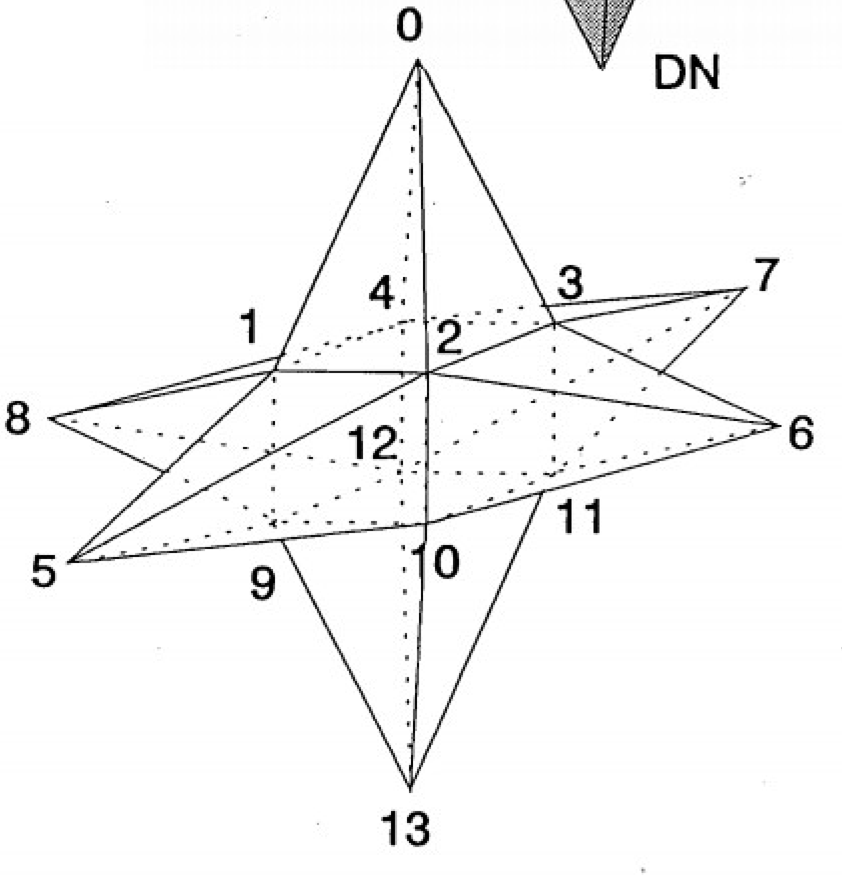
0	(-100,-50,-50)	7:	(-100,50,50)
1	(0,-50,-50)	8:	(50,-50,-50)
2	(0,50,-50)	9:	(50,-50,50)
3	(-100,50,-50)	10:	(50,50,50)
4.	(-100,-50,50)	11:	(50,50,-50)
5	(0,-50,50)	12:	(100,0,0)
6:	(0,50,50)		



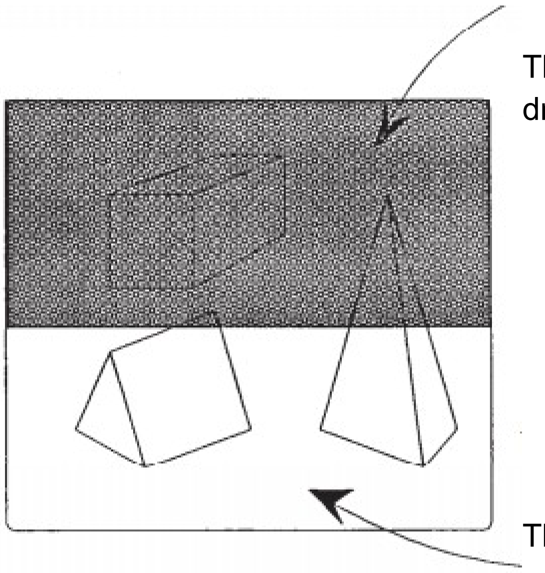
In this figure, the numbers represent the odd one, mutual number of surfaces.

Figur 25



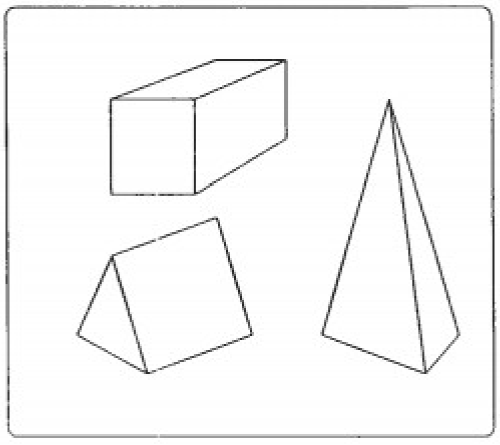


The top figure shows the names of the individual sub-objects. Each constellation is a sub-object. The lower figure shows the coordinate numbers of the individual corners.

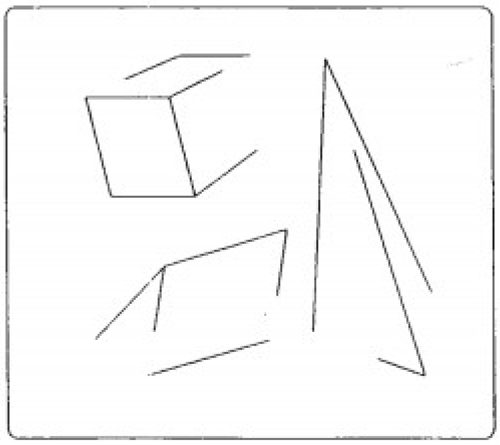

1

This region has the electron beam already drawn on the screen.

This area has not yet been drawn.


2

This screen is ready to be displayed and will be drawn on the monitor, as soon as picture 1 is drawn.


3

Here is the next screen about to be drawn.

For double-buffering, screen 2 will be removed. It is therefore necessary to wait for the electron beam to finish drawing image 1 before we can tackle image 1 again. It delays the refresh of the screen.