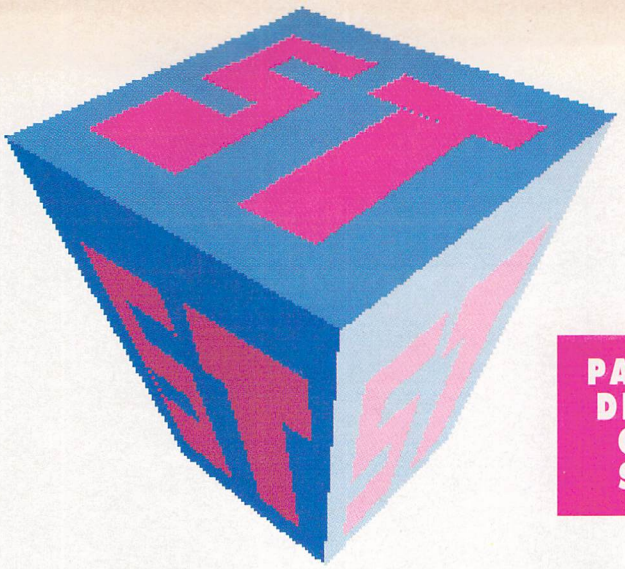
# Real-Time 3D Graphics

***Have you ever wished you could create solid 3D graphics like those you see in flight simulators? Andrew Tyler, in this series, shows you how...***



**PART ONE: DRAWING ON THE SCREEN**

In this six-part series about real-time 3D graphics on the ST, you'll learn about all those aspects of graphics which are essential to creating a solid object on the screen and making it look real: fast polygon filling, windowing, hidden surface removal, lighting, colour and rotation.

You'll be able to create and manipulate objects on the screen, just like in a favourite flight simulator or game. But one essential feature of a simulator of any kind, whether it be about flying, road racing or whatever, is that it runs in real-time.

Real-time means that the image on the screen responds immediately and smoothly to input from the joystick or keyboard.

To do this convincingly for a complex scene is quite a tall order. Even commercial films usually draw and photograph each frame separately, so that motion is only seen when the film is played back.

It's quite amazing, and a testament to the power of the ST, that such impressive games and simulators using solid vector graphics have been written for it.

In this, the first in the series, you'll learn how to draw and fill polygons – they're the building blocks. Later, you'll be able to animate and control them, in three dimensions! Everything needed to do this will be either on the CoverDisk or on these pages.

## QUICK DRAW

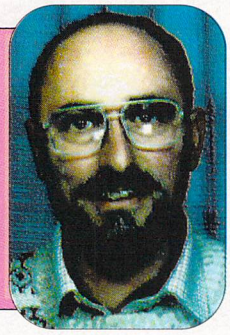
In order to produce real-time 3D graphics for a popular micro like the ST there is really only one option open – using so-called 'vector' graphics. But forget about the term vector for now – put simply, what this means is that objects on the screen are constructed from polygon meshes.

A polygon is a flat geometric shape with straight edges. An example of a four-sided polygon is a square. And a polygon mesh is a group of polygons joined at their edges.

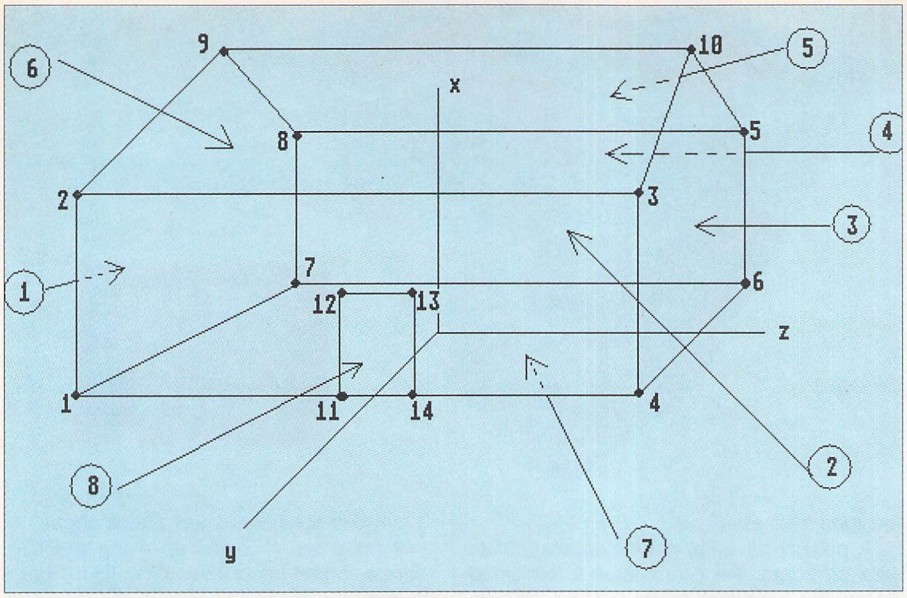
If the mesh is joined up along all of its edges so that it's closed, it's called a polyhedron. A cube is an example of a six-sided polyhedron. Houses are constructed from six-sided polyhedra, called bricks. Polyhedra are used to build houses and graphics objects for roughly the same reasons: it's easy to construct building blocks with flat faces and straight edges.

In the case of computer graphics on an ST it's particularly important that an object can be drawn quickly from a minimum of information. When polygon meshes are used, all that's required is a list of the coordinates of the vertices, or corners.

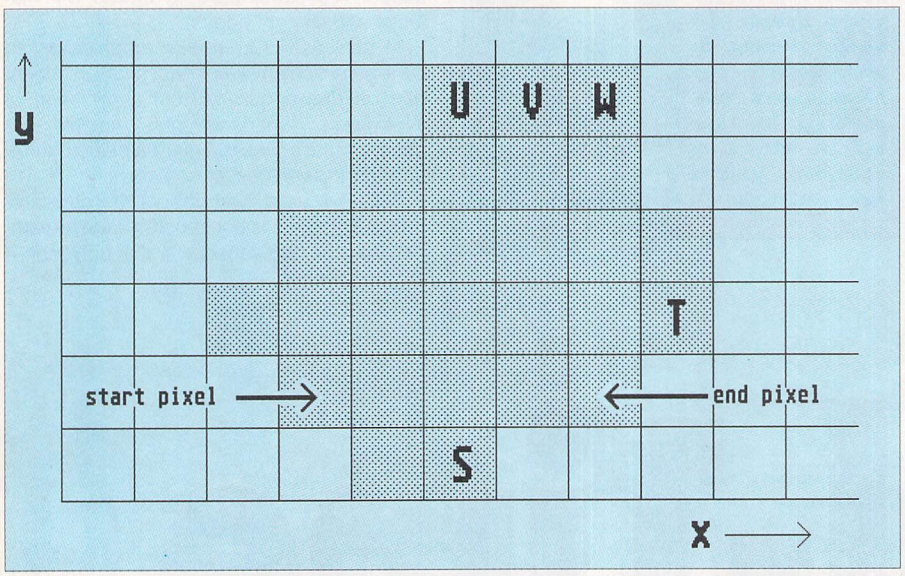
For now, don't worry about the details of how a 3D object is constructed. Just remember that the flat surfaces – the polygons –



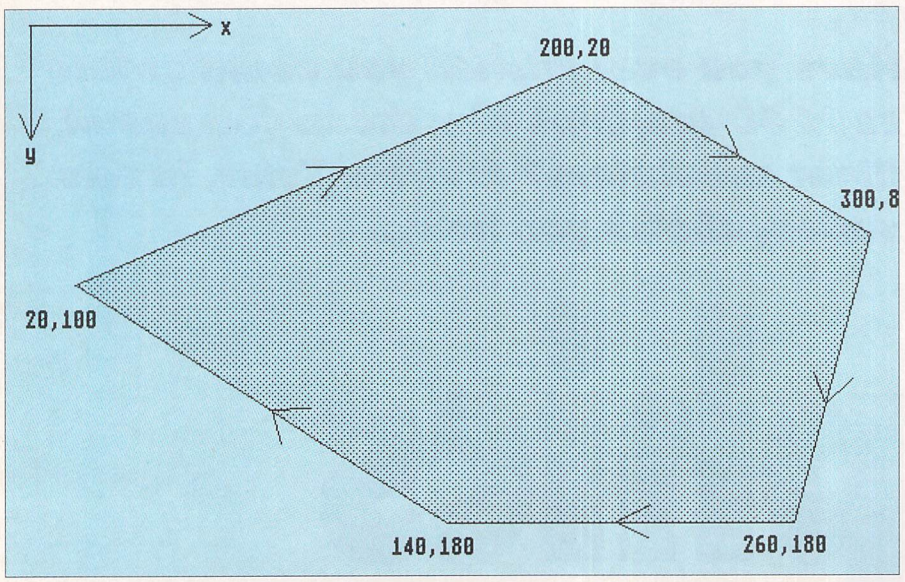
**Andrew Tyler is a university lecturer and teaches a course in microcomputer and microprocessor programming. He has had an interest in computer graphics for several years.**




**Surfaces (circled) and vertices (uncircled) of a polygon house. Bricks are examples of simple polygons. In basic computer graphics polygons are fitted together to form more complex shapes, like this house**




**A pixel polygon, magnified to show how lines are drawn using individual pixels. On a normal-sized screen the pixels merge to form a straight line**




**The numbers of the pentagon's vertices and edge connections. The pentagon is constructed from a series of straight lines joined together to form a simple polygon**

which make up its sides are of paramount importance. So we need to be able to draw and fill in these polygons quickly.

## ASSUMING ASSEMBLY

The first thing needed is assembly language. But stop! Before you decide it's too difficult – think again. It's not as unfriendly as it first appears. It's the fastest programming language and perfect for real-time graphics.

All the programs in this series are written in assembler code. To convert the assembler code into something the 68000 processor at the heart of the ST can understand requires a special program called an assembler.

There are several very good assemblers available, even some in the public domain. All of the programs in this series have been written using the very popular and powerful DevpacST assembler from HiSoft. On this month's CoverDisk you'll find Version 1 of this program, called GENST. It's a very friendly and helpful integrated package which should have you creating 3D graphics in no time.

Devpac Version 2 has the assembler and debugger – the bit that checks for errors – all within the single program. With Version 1 of Devpac you can learn the basics but Version 2 will take you even further.

## FAST FURIOUS FILLING

The objects for animation are assembled from polygons glued together at the edges. Figure 1 shows a house constructed in this way, with the circled numbers referring to polygons and the uncircled numbers referring to the vertices, or corners.

Also shown on the diagram are the axes of the (x,y and z) coordinate system in which the house is defined. In this system, y is the horizontal axis, x is the vertical axis and z makes up the depth.

In the scheme we use, the house stands in an imaginary world inside the computer, called the world reference frame. What appears on the screen is the result of a series of calculations involving rotations, perspective and windowing – or chopping – of what is outside the screen.

Don't worry, all of these will be explained later, but for the moment let's concentrate on how to draw a solid polygon, which is the most basic graphic element.

## JOINING THE DOTS

Suppose we want to draw and fill in a very small pentagon – a five-sided polygon. On-screen, with a magnifying glass, this might look like Figure 2.

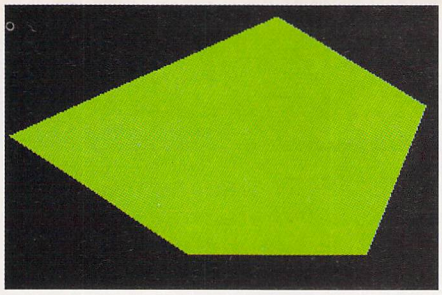
The sides of the pentagon aren't straight because the screen is really made up of a series of closely spaced dots, or pixels, which can only be either on or off.

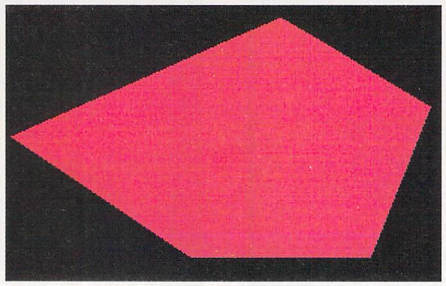
Without going into detail as to how this screen RAM is laid out for the three resolutions of the ST, it's clear that the drawing of lines is really a 'join the dots' exercise.

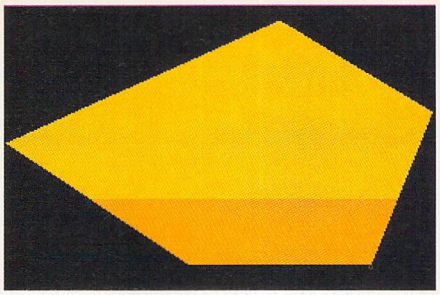
There's no need to worry here about the details of how the screen RAM is mapped to the screen since the Line A routines in the ST Operating System are used to do the actual drawing – they're fast enough for what we want here and will take care of these details.

For drawing and filling a polygon, we need to calculate the coordinates of the pixels on the left and right boundaries, and then join the corresponding pairs with horizontal lines.

The start and end pixels shown in Figure 2 illustrate this for one particular line. The strate-







**The finished 3D-01 program from the CoverDisk showing how a polygon can be filled with colour. The program cycles automatically through six different palettes of colour**

gy is simple enough. Calculating the coordinates of the boundary pixels however, isn't so easy. Fortunately this problem was solved by J E Bresenham in 1962 and has been the basis of many line drawing programs ever since.

## BUFFING UP BRESENHAM

The modified Bresenham routine generates the coordinates of the start pixels on the left-hand side and the end coordinates on the right-hand side of a polygon to be filled in with horizontal lines.

Once again, with the emphasis on speed, we want to store the data in a way which makes it immediately accessible to the Line A routine which draws the horizontal lines. This store is called the x-buffer and has an address in RAM at xbuf.

In assembly language, addresses are usually denoted by such labels, and the assembler finally converts them to real addresses. The layout of xbuf is an example of how assembly language programming takes advantage of easy access to RAM to provide speedy access to data.

Xbuf is just a block of 400 long words – the 68000 can handle bits, bytes (8 bits), words (16 bits) and long words (32 bits). Each contains the start and end x-coordinates of a horizontal line.

This is big enough to cope with the 400 possible y-values (vertical positions) on the screen in high resolution, and therefore also medium and low resolution. Each long word refers to a particular y-value which is the position of that word from the start of the block.

## ON THE RIGHT LINE

The ST comes armed with a host of fast built-in graphics routines. One of them in particular is very useful. It's the one which draws horizontal lines.

These routines are tailored to the hardware of the ST (and are therefore called device-dependent) and are accessible through a special feature of the 68000 processor called exception handling.

One 'exception' has been used by the designers of the ST to enable the user to do fast graphics. It's called the Line A Emulator. Without becoming bogged down in technicalities, it works in the following way.

Whenever the processor sees the word $A00n (the $ sign means that this is a hexadecimal number) it recognises an exception, takes control and executes the nth routine of the special graphics functions, all to be found in ROM.

The processor knows where the functions are located because it keeps the address in a special list called the exception vector table. Accessing special routines this way is an alternative to subroutines.

These routines use special variables tables to do their business, and to set these up the routine $A000 must be called before any others as part of initialization.

The horizontal line drawing routine is number four in the list. To use it, the start and end pixel coordinates must be passed to the tables together with other information, such as the colour which is required and what kind of line it is to be.

There's scope for variety here since the line can have a fill pattern – it can be dotted in many different ways. We've used a solid line but you could experiment. The block to play with is Fill, in the file data_00.s

## IN GLORIOUS COLOUR

In low resolution on the ST, it's possible to use 16 colours simultaneously out of a possible 512. What these colours are is easily seen on the Control Panel by means of the red, green and blue sliders.

There are only 16 colours available at any given time because the hardware which is responsible for generating the colour can only read a nibble-sized (4 bits) piece of data. Each of these bits is said to refer to a different colour plane. So in low resolution the ST has four colour planes. In high resolution it has only one.

To change the 16 colours in the palette a special routine in the Operating System can be used. This uses a programmable exception, which in this case is called a TRAP.

Like the Line A routines, the TRAPs are numbered. TRAP #14 accesses that part of the OS called the XBIOS. Once called, the actual routine to be executed is specified by a number pushed onto the STACK. You can see how it's done in the listings.

## TAKE A LOOK AT THESE

What has been shown here is how to draw and fill in a large polygon quickly and easily. This is an essential starting point for solid 3D graphics.

The polygon is, in fact, a pentagon and the coordinates of its vertices are listed in the data section at the end of the file 3D_01.S. It's shown in Figure 3.

To make it colourful, six colour palettes are cycled through in succession. The program is split up into several bits, called source files. The one that really matters is 3D_01.S.

This is the one which must be assembled by GENST. When this happens it pulls in all the other files by means of the INCLUDE directive. This is how you run the assembler.

From the Desktop, first load the assembler program GENST.PRG. Then load 3D_01.S from the File menu. Then pull down the Options menu and click on Assemble.

Give the binary file to be generated the filename 3D_01.PRG and assemble it with no listing. Quit the assembler and run 3D_01.PRG from the Desktop as usual.

One final point. The picture on the screen will flicker. This is a consequence of using only one screen to draw on. It can be avoided by using a technique called 'screen buffering'. We'll talk about that next time together with how to 'window', or clip, a picture to fit into the monitor screen.

All of the programs associated with this series are included on the CoverDisk. See Runtime on the CoverDisk for a full explanation of what they are and how to use them.

## IN THE BOOK

This article and programs are based on a forthcoming book, *Real Time 3D Graphics* by Andrew Tyler which is scheduled for publication soon by Sigma Press. Everything which is included here is discussed in far greater detail in the book, together with example programs and much more besides.

## JARGON LIST

**Assembler code** – the assembly language's 'words'

**Assembly language** – the 'lowest' programming language

**Bresenham's algorithm** – a method of drawing lines very quickly

**Exception** – how the 68000 processor handles an 'unusual' situation

**Exception vector table** – addresses of routines that handle exceptions

**Graphics primitives** – fancy name for simple objects on the screen

**Hexadecimal** – computer number system. Lumps four bits together as a unit (nibble)

**Line A emulator** – how the 68000 calls Line A routines

**Line A routines** – fast drawing routines in the ST Operating System

**Nibble** – a group of 4 bits (binary digits) with a top value of 16. Two nibbles to a byte

**Polygon** – 2D object with straight sides

**Polygon mesh** – several polygons connected together at their edges

**Solid vector graphics** – 'solid' objects drawn from polygons

**Real-time** – running now, as you watch it

**TRAP** – programmable exception built into the 68000 processor

**Vector** – separation between two points, shown as an arrow

**World reference frame** – the 'world' inside the computer

**X-buffer** – x-coordinates for starting and ending horizontal lines

# A Window on the World

***Andrew Tyler reveals that there's often more than meets the eye when you move into the third dimension...***

Last month, in the first part of this series, we looked at how to draw solid polygons. Remember, a polygon is a mathematical name for a flat surface with several straight sides – a triangle is the simplest. Now we can 'glue' them together to make the solid objects in our real-time 3D graphics programs.

In this instalment we will look at two further aspects of getting a picture onto the screen: windowing and screen buffering. The first of these is the last stage of a series of transformations, or changes, which are made to an object in bringing it from the world inside the computer onto the screen.

The second is really just a technicality to make the 'movie' look real. Both windowing and screen buffering are implemented in the example program on the CoverDisk.

## THROUGH THE WINDOW

Windowing means exactly what it says. When you look out of a window you don't see everything that's outside, only what is not cut off by the window frame. The monitor screen is similarly limited in size and there must be some way of 'clipping' off the bits of the picture which fall outside.

In fact, the outline of the visible window is called the view port, or sometimes the clip frame to emphasise its function. It isn't even necessary to confine the clip frame to the screen boundary. Often the 'active' part of the screen is much smaller, especially when the picture is complicated and much drawing has to be done.

Having a small picture speeds things up – speed is always a serious consideration in computer graphics. Sometimes the ability to vary the visible screen size can be used to special effect – like an iris opening. Figure 1 shows windowing.

Windowing of some kind is essential. This is because there is a fixed block of RAM reserved for the physical screen – there's also a logical screen – more about that later – and this defines a window.

In low resolution, as on a colour television, this window is 200 pixels high and 320 pixels wide. Any attempt to draw something outside these limits could result in disaster because the only way it can be done is to encroach on the RAM outside of screen RAM. If this bordering RAM is being used for something else, like the program for example, then it will be overwritten.

Clearly there are precautions that can be taken to ensure that windowing is kept to a minimum, such as totally rejecting objects that lie entirely outside the screen limits. But in the quest for realism it is inevitable that some objects will span the view port.

The windowing algorithm must decide what is inside and what is outside the view port and what the clipped object will look like. Since we are going to be drawing solid objects, on which you can't see the hidden sides at the rear, it isn't good enough to simply forget about lines that lie outside the clip frame.

These lines must be replaced by ones that complete the clipped shape at the window boundary. This requirement to close shapes in order to fill them in is shown in Figure 2.

An elegant solution to this problem was found many years ago by Sutherland and Hodgman. Ivan Sutherland is one of the folk heroes of computer graphics – he was the originator of many techniques in his Sketchpad system.

Before discussing Sutherland and Hodgman's work, it's worth commenting that some form of clipping is immediately available from the way data has been stored before drawing.

Remember from Part One that each polygon to be filled by a series of horizontal scan lines has the start and end coordinates of each line stored in a list called the x-buffer. The list starts at the highest y-coordinate and ends at the lowest.

To clip within a smaller window, all that has to be done is to ignore those lines which lie outside the window in the y direction, and adjust the x-coordinates when they lie outside the window in the x direction.

Right now we prefer to clip objects before they get to the x-buffer. And the Sutherland-Hodgman algorithm does this very well.

## CLIPPING ALGORITHM

The Sutherland-Hodgman algorithm is, in fact, more powerful than is really required here – it can handle polygons of any shape. It does not even require the clip frame to be rectangular.

For speed and simplicity the objects we draw will be constructed only from convex polygons in this series – all external angles greater than zero, and essentially 'round' in shape. Figure 2 is more general since it illustrates clipping of a non-convex polygon.

Looking at the diagram we can see that the general effect of windowing is to chop off vertices, such as A, and replace them with new edges connected by new vertices, such as R and S. The Sutherland Hodgman strategy is to find the intersections in turn of all the

edges of the polygon with each boundary.

Since this clip frame has four sides, this means that four complete cycles of the polygon will be made. On each cycle some of the original edges may be lost and new ones added.

As each new vertex is examined, various actions are taken which depend on its position and that of the previous vertex examined. These cases are illustrated in Figure 2 and examined below:

1. If the next vertex is outside the frame (A), check the position of the previous vertex, (C). If that was inside, find the point of intersection (S) of the edge with the clip frame, and save it. Don't save the next vertex.

2. If the next vertex is inside the frame, (B), check the position of the two previous vertex, (A). If that was out, find the point of intersection of the edge joining them with the clip frame, (R) and save it. Also save the next vertex, (B). This is the algorithm applied to all the vertices going round the polygon.

## CALCULATION BY ITERATION

At first sight, it might appear that calculating the points of intersection of sloping polygon sides with the clip frame requires a lot of nasty mathematical computation involving time-consuming multiplications and divisions (the slowest instruction). This is something we definitely want to avoid for our real-time graphics. Surprisingly, this is not so.

One of the bonuses of working directly in assembly language is that it's possible to get answers using only additions and subtractions, and where they are unavoidable, to do multiplications and divisions in powers of two by means of fast left and right shifts of the contents of registers. Are you still with us?

The secret of calculating the points of intersection is to use iteration. To illustrate how this is done look at Figure 3. Here is shown the case where the previous point (A) was outside, but the next point (B) is inside the frame boundary x=xmin.

There are two possibilities depending on whether A or B is nearer the boundary. What is wanted here is the intersection of the edge AB with the frame. This intersection will replace A in the list of polygon vertices after the clipping.

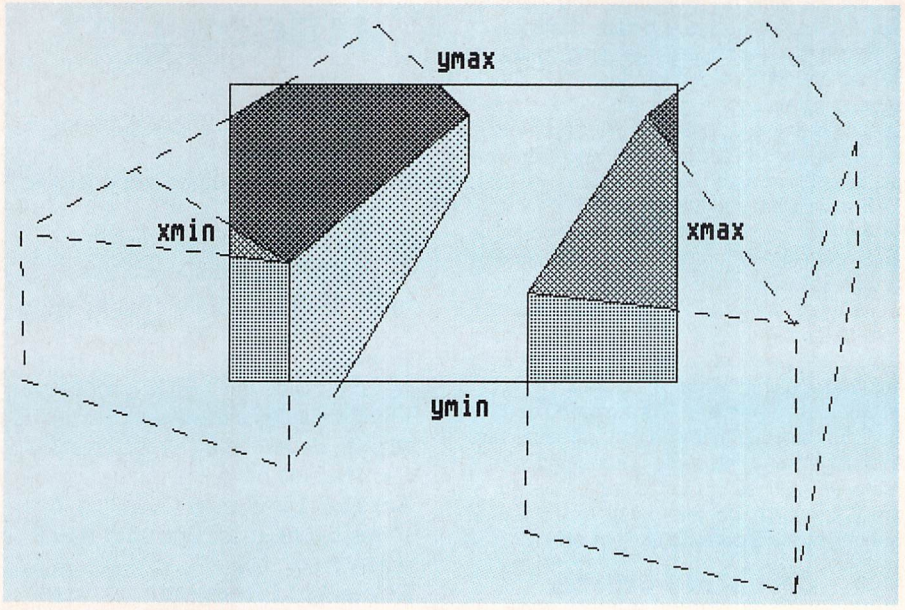
To find the point of intersection the average of the coordinates of A and B is calculated. This is T1. Then either T1 and A are averaged, as in case 1, or T1 and B are averaged, as in case 2. In either case notice how the new average, T2 is getting closer to the boundary.

This iterative process continues until the boundary itself is reached, which does not take long. The averaging can be done quickly with an addition and a right shift – a very fast division by two. To follow the complete algorithm through in all its gory details, look at the subroutine file core_01.s

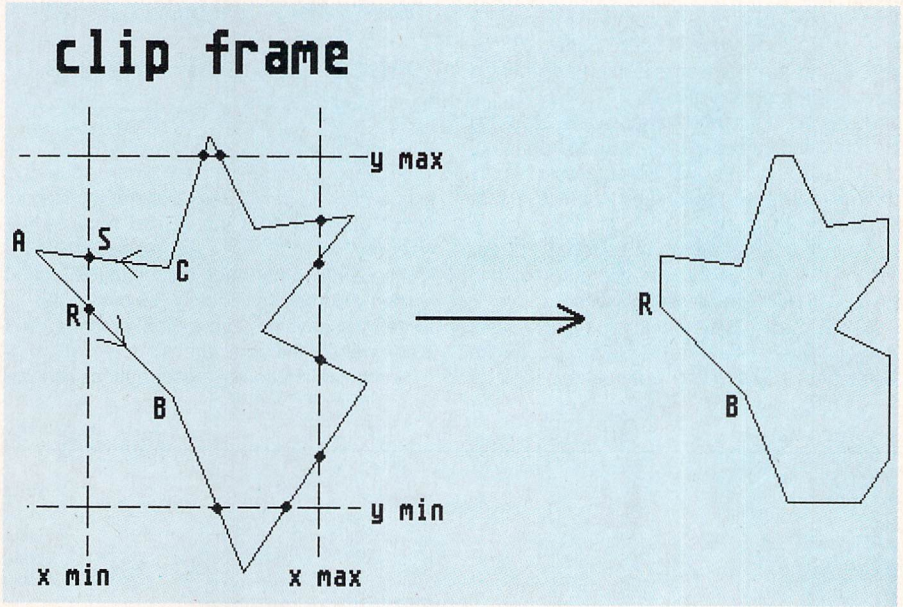
## SCREEN BUFFERING

The problem with the picture generated in Part One was that it flickered. Generating real-time computer graphics is almost the same as making a movie, except that in computer graphics the pictures are drawn just before they are displayed, often in response to changing input conditions from a joystick or some other device.

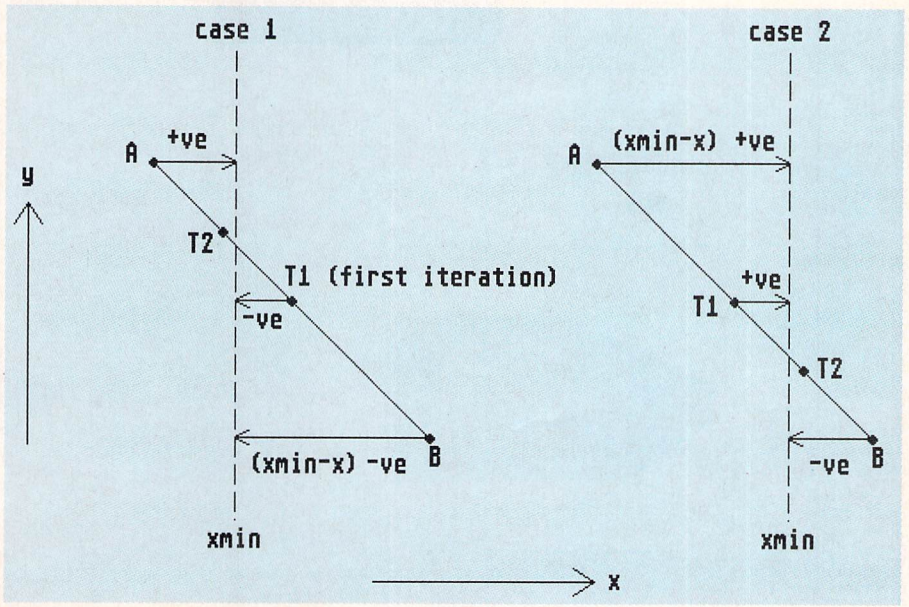
This is where computer graphics really scores over movies, which are totally predictable. The thing they both have in common is that the illusion of motion is created by pro-




**Windowing – the ability to view a part of an image, such as a polygon, is essential**




**Clipping a polygon – the polygon must be closed at the boundary**




**Finding a point of intersection by iteration – where the polygon gets the chop**

jecting still pictures, or frames, in a progression too fast for the eye to detect the flicker.

For this to happen, the time between frames should be less than about 1/20th of a second. Of course, in games this time is often stretched to the limit, and any real program is a trade-off between complexity and the slowest tolerable speed.

The problem for us is that with a single screen it's not possible to display one frame and generate the next. The solution is straightforward – have two or more screens.

We'll take the simplest option, two screens. One to display the frame which has just been drawn (the physical screen) and one to draw the next one on (the logical screen).

When the next frame is drawn and needs to be displayed, these two screens are exchanged and the cycle is repeated. There are even routines in the Operating System of the ST to help do this. First, a few words about how the Operating System works.

## OPERATING SYSTEM

The Operating Sysytem (OS) as a whole is called TOS, but it consists of several parts. There are the device-independent parts (with abbreviations BDOS, XBDOS, VDI and AES) which would work on any computer with TOS, and the device-dependent parts (BIOS, XBIOS and line A routines) which depend on the hardware details of the ST.

These all constitute 'legal' ways of using the OS. Illegal ways meddle with the contents of system registers whose addresses may change between machines manufactured at different times.

We are definitely going to be legal. For our purposes, there are several useful routines in the XBIOS concerned with the screens.

XBIOS means eXtension of the Basic Input Output System. It's a large piece of software residing in ROM which controls the flow of information to and from such externals as the keyboard, mouse, joystick and the monitor.

For our purposes the XBIOS routines we want are: #2 – tell us the address of the physical screen (we can then calculate the logical screen address by adding 32kbytes); #5 – switch the physical and logical screens; and #37 – wait for the vertical blank interrupt.

Some explanation of these routines is helpful. They are listed in the file system_02.s. All the XBIOS routines work by pushing all relevant information onto the STACK which is a temporary storage area for the 68000 processor, then declaring the TRAP #14 instruction.

This is another of those exception instructions which the system supervisor performs privately. Routine #2 tells us where the OS has located the physical screen and, by adding 32kbytes, determines where to place the logical screen. The distinction between these two screens is lost as soon as they have been switched once so in the routines they are simply called screen 1 and screen 2.

Routine #5 causes the system to switch the identity of the two screens and #37 delays this switch until the electron beam in the monitor screen reaches the bottom and is ready to fly back to the top. The time for this to occur is called the vertical blank interrupt and is a good time to make a flicker-free switch.

There is one other requirement of course. The program must also be synchronised to the swap of the two screens. The swap is called at the end of drawing each frame.

## JARGON LIST

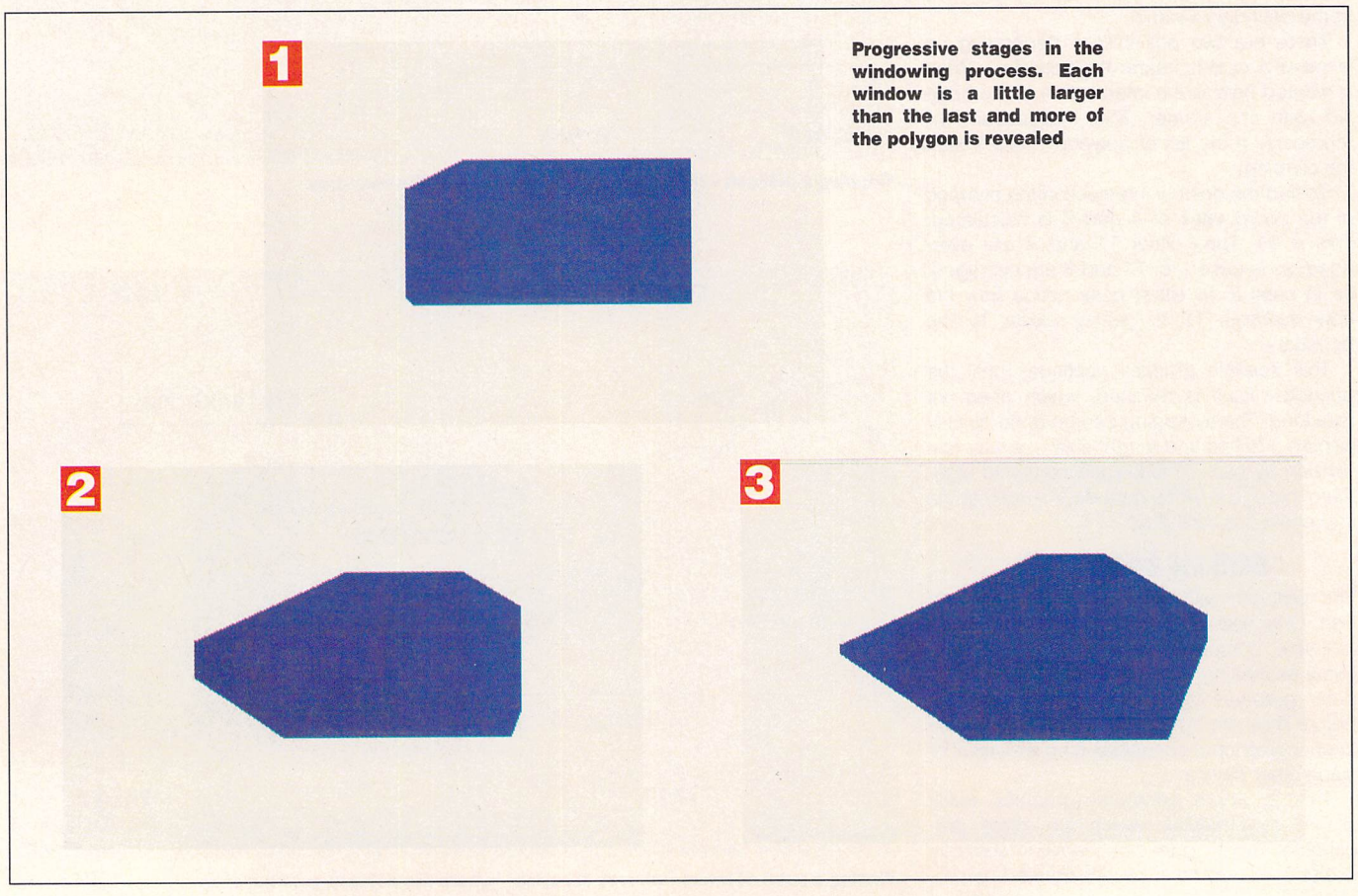
**BIOS** – the part of the ST's operating system concerned with input and output
**Clip frame** – another name for the window
**Iteration** – going around in a circle and getting better each time
**Left shift** – moving a register's contents one bit to the left
**Register** – one of the special places inside the 68000 processor where calculations are done
**Right shift** – moving a register's contents one bit to the right
**Screen buffering** – having two screens; one to display and one to draw on
**Sutherland-Hodgman algorithm** – a way of windowing
**Vertex** – the sharp corner of a polygon
**Windowing** – chopping out what's not in view
**XBIOS** – an extension to the BIOS

## THE EXAMPLE PROGRAM

The example program draws a pentagon inside an opening rectangular window. The control program to do this is in the file 3D_01.S and should be assembled with the DevpacST1 assembler on the CoverDisk.

As indicated by the Include directives in this file, all other files will be pulled in at assembly. Since these include files from Part One it would be a good idea to dedicate a disk to this series and copy all the files onto it.

● *This is one in a series of articles which are based on a forthcoming publication:* ***Real Time 3D Graphics*** *by Andrew Tyler. The book is to be published soon by Sigma Press.*




**Progressive stages in the windowing process. Each window is a little larger than the last and more of the polygon is revealed**

# A point of view

***Andrew Tyler continues his series on bringing 3D graphics to the ST screen***

It's a curious fact that distant objects look smaller than those which are close. They aren't smaller, but they subtend a smaller angle at the eye. So for any world we create inside the computer to look real, the size of objects must diminish as they recede into the distance.

In real life all of this is done by the brain and the eye. On the computer screen we have to get the same effect with the help of geometry. That's what this month's article is all about – the perspective transform.

But before we launch into the perspective transform it's time to face up to an unfriendly little topic that's been lurking in the background since Part 1, when we considered coordinate systems and frames of reference.

It's a good idea to deal with it here, because from now on we'll have to think in 3D most of the time, rather than the 2D world of the monitor screen.

One of the most confusing aspects of computer graphics results from the way geometry is used in the various stages of bringing an object to life on-screen.

It helps a lot to visualise these stages from particular points of view which are called frames of reference.

For example, suppose you are going to build a model railway and have marked out the positions of each section of track, buildings and any other structures on a rectangular base, where x is along the long side and y is along the short side and zero is at one corner.

This way of fixing the positions of everything is called the world reference frame, and the way x and y are set up is called the world coordinate system.

But each individual object in the layout has its own detailed structure ,and we must have a separate way of specifying this, independent of where it is on the layout.

We could imagine listing the dimensions of each part of the object as measured from its centre. This is called the object reference frame.

Then, since the train itself may be in continuous motion, we need only specify its current position on the track by a single pair of coordinates – perhaps the coordinates of its centre would do – in the world frame. We can then refer to its object frame for the details.

But hold on, we haven't finished yet. Since this layout has to be drawn on the ST's screen, it has to be seen from a particular viewpoint, that of you, the observer. The frame of reference attached to the observer is called the view reference frame.

## Jargon list

**Subtend:** Defines an angle of view

**Column vector:** The components of a vector written as a list in a column

**FRAME:** A space opened on the STACK by the LINK instruction

**Homogenous co-ordinates:** 4D mathematical space

**LINK:** A 68000 instruction that lets you reserve space on the STACK

**Matrices:** A way of showing how one vector changes into another – as in rotation, for example

**Perspective transform:** Projecting the 3D world inside the computer on to the 2D screen

**STACK:** A temporary storage place used by the 68000 processor

**World coordinate system:** The grid layout for the 3D world inside the computer

**World reference frame:** The space occupied by the 3D world inside the computer

## CONFUSED?

Furthermore, the outline of each visible object when projected on to the screen for display is then a set of polygons whose vertices are specified in screen coordinates. The relation between these various reference frames is shown in Figure 1.

One other thing. Confusion abounds when it comes to setting up the coordinate axes which accompany these frames of reference. This is because there is no unique way of labelling the directions up, forwards and sideways, and different people have different preferences.

The first option is to decide whether to use left or right-handed coordinates. Figure 2 shows these alternatives. We have chosen to use the right-handed system, since that's what is comon in science and engineering.

The decision to point x upwards results from another convention in computer graphics – that of pointing the z-axis into the picture. This convention is adhered to consistently throughout.

There is one last coordinate system to deal with – that of the screen. The origin (0,0) of the screen coordinates (xs,ys) is at the top left-hand corner of the screen, so a conversion to this system has to be made before the picture can be displayed.

Figure 3 shows an object, in this case a cube, defined inside the computer in the world frame and seen from the viewpoint – the observer's eye, also called the centre of projection – which lies some distance (-100 in this case) along the negative axis of the view frame.

The monitor screen lies in the xv, yv plane of the view frame and is called the view plane. Complicated, isn't it! What we want on the monitor is the outline marked by the intersection on the view plane of the 'rays' from the object to the viewpoint.

That's really all there is to it. This is a particularly simple type of perspective projection. Draughtsmen use many other types, but this one works well.

To see how the perspective projection is worked out, look at Figure 4, where the line ED produces the perspective projection CB. Without going into details, you can see that the triangle ABC is similar to the triangle AED, so that the ratio CB/ED is in the ratio d/(d+zv). This, therefore, is the factor by which DE must be multiplied to get the projection CB.

There only remains an adjustment to express CB in terms of screen coordinates xs and ys, which have their origin at the top left-hand corner of the screen.

It's pretty clear that drawing in 3D gets complicated unless you have a shorthand notation. That's what vectors are – they allow you to specify a distance and direction in 3D

easily and quickly. It's not surprising then that many of the complicated calculations in computer graphics are done in terms of vectors. In fact, it turns out that these calculations can themselves also be written in a special shorthand notation – called matrices.

Vectors and matrices go hand in hand. Whatever convention is chosen for the vectors affects what the matrices look like.

Let's look at how vectors and matrices can be used to perform calculations in computer graphics. This is one of the areas of mystique of the subject. If you can understand what's going on here you can really impress your friends.

To be really tricky, we'll do the perspective projection as a matrix calculation. It isn't really necessary to do it this way, but it will provide an opportunity to introduce another buzz word – homogeneous coordinates – and show off a clever assembler instruction, LINK, which allows us to safely meddle with the STACK.

You might remember from your GCSE maths that rotations can be done by matrices. This is certainly one application that is very useful and straightforward.

Figure 5 shows a 2D vector which stretches from the origin to the point 1,0 being rotated to the point 0,1. We want to have some way of saying this using geometry. A matrix is the answer.

Here's the way it's written down:

$$\begin{pmatrix}0\\1\end{pmatrix}=\begin{pmatrix}0 & -1\\1 & 0\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix}$$

What does this mean? The object on the left hand side is the final vector, written in a column and the object on the extreme right is the original vector, also written as a column. The thing in the middle is called a 2x2 matrix transform for a 90° rotation in the x-y plane.

To be thorough we should have also included the z-axis – which points out of the paper – in the figure even though no z values are changed by this particular rotation. Then the matrix would have had 3 rows and 3 columns making it 3x3. All rotations are 3x3 matrices.

The product as a whole is a piece of mathematical machinery. The good news is that having worked it out for this one case, it will work on any vector we care to try, not just the one at 0,1.

Well, we'll meet rotations again later. For the moment, just note that matrix transforms like to multiply a column vector on the right to produce an answer on the left.

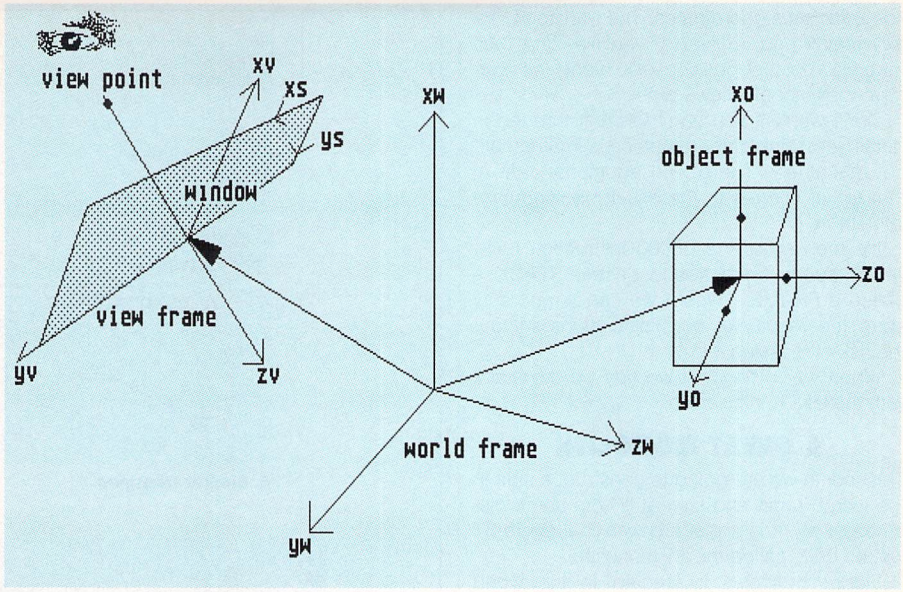
The perspective transform just can't be done using regular 3x3 matrices in x-y-z space, but it can if we move into the fourth dimension and treat it as a 4x4 transform.

Doesn't that sound crazy? But mathematicians do this sort of thing all the time. Inventing an extra dimension provides a way round the problem – it gives more 'space' to move in.

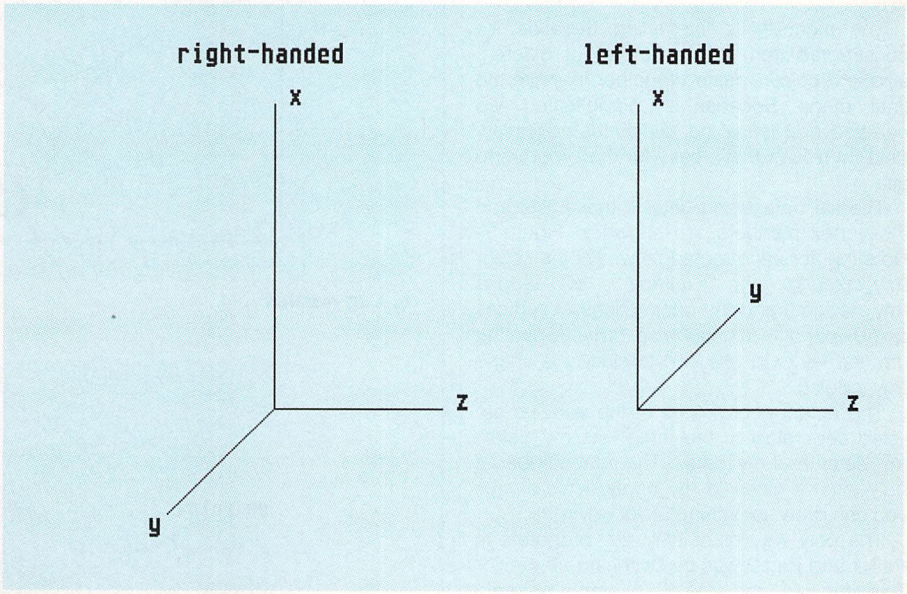
Funnily enough, the value of this dimension is always 1 and so a point in ordinary space with coordinates x,y,z has coordinates x,y,z,1 in this 4D space.

These four dimensions are called homogeneous coordinates. They are very popular with graphics programmers because with them, all manipulations of an object can be done as a matrix product.

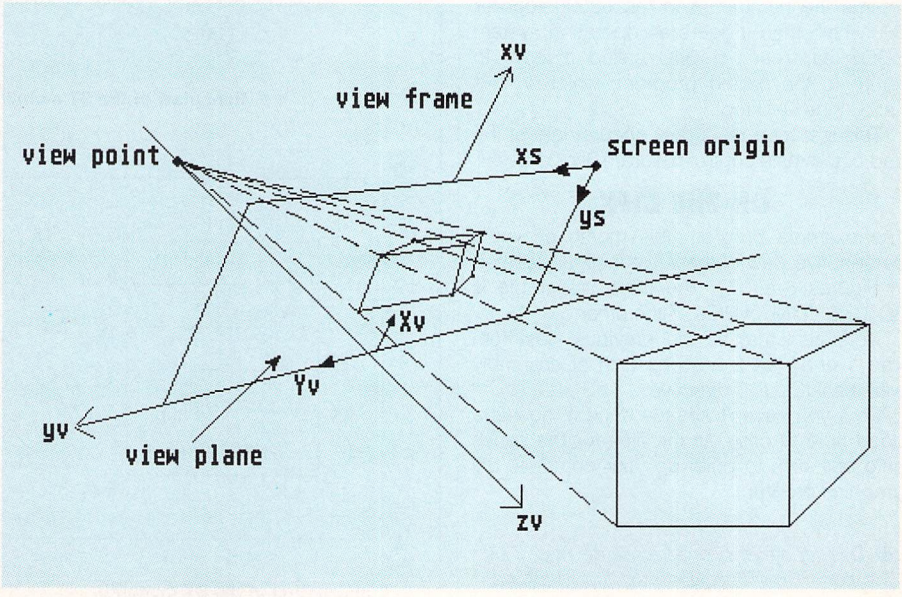
In the program file on the CoverDisk called core_02.s you can see how the perspective transform is done as a matrix product in




**1. Frames of Reference**




**2. Right-handed and left-handed coordinates**




**3. Perspective projection**

homogeneous coordinates. The transform is a 4x4 matrix and is listed in the file data_02.s for a viewpoint at position -100 along the negative z-axis of the view frame.

Don't worry if you can't follow it through in detail, just think of it as a piece of mathematical machinery. There's an additional item in that file of interest to the assembly language 'athlete'.

It's the use of the 68000 instruction LINK which opens up a space on the STACK – called a FRAME – where we can temporarily store the results of our calculations without the risk of messing it up.

When we've finished we tidy up the stack with the UNLK instruction.

## A GREAT MONOLITH

This month we are going to construct a sign in the world frame and look at it from some distance away. It is a placard with the letters ST written on it, called the ST monolith.

Figure 6 shows it erected in the world frame and seen from behind. In later instalments we'll rotate it in various ways and show it illuminated by a light source. Right now all we want to do is build it and look at it.

The monolith is interesting because it's constructed from six rectangles of different size and colour, pasted together to make the final image. Because it's complicated, we need several lists to contain all the necessary data, in a form that's easy for the program to use.

The file data_01.s contains these lists and the vertex numbers are shown in Figure 7. Note that for each rectangle we list the colour (my_colour), the number of edges (my_nedges) and the connections of vertices going round in a clockwise order repeating the first vertex at the end to close the shape (my_edglst).

The actual coordinates of the vertices are listed separately in the three lists my_datax, my_datay and my_dataz. The total number of polygons is given in my_npoly. If you want, you can draw something of your own.

The only way of quitting the programs in Part 1 and Part 2 was by switching off the ST. This time, you can stop the example program running from the keyboard.

This is done using first an operating system BIOS call – number $1, called bconstat – to see if a key has been pressed. If one has, this returns the number -1 in the 68000 register D0, which then triggers an operating system BDOS call (number $4C called p_term) to return to the calling program, which in this case is the Desktop.

This test for a key press occurs right at the end of the main control program 3D_03.S

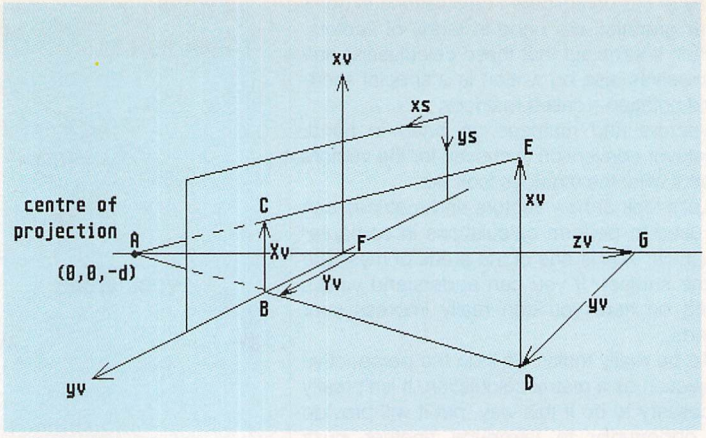
## ON THE DISK

The example program this month shows a perspective view of the ST monolith illustrated in Figures 6 and 7. The file to assemble is 3D_03.S. It INCLUDEs all the others.

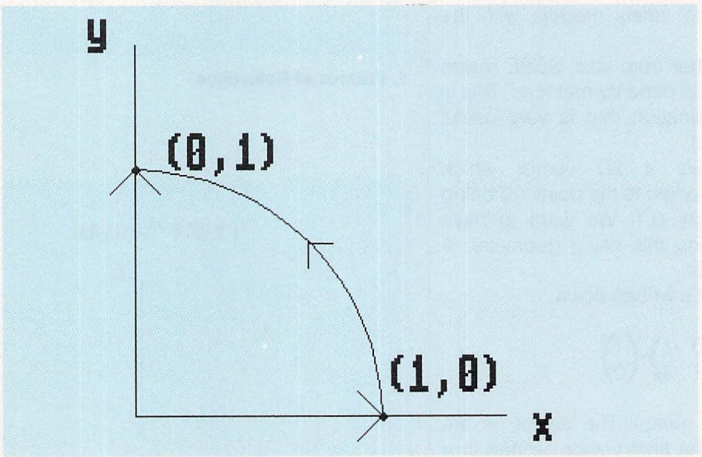
Note also that all the previous files from Part 1 and Part 2 are pulled in at assembly with the INCLUDE directive.

So if you haven't already done it, now is a good time to copy all the files for the series onto one disk to build up your complete 3D graphics program.

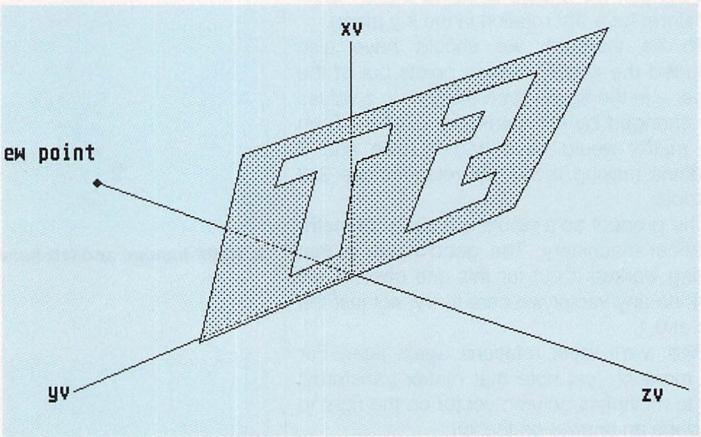
● *This article and programs are based on the book RealTime 3D Graphics which has been published recently.*



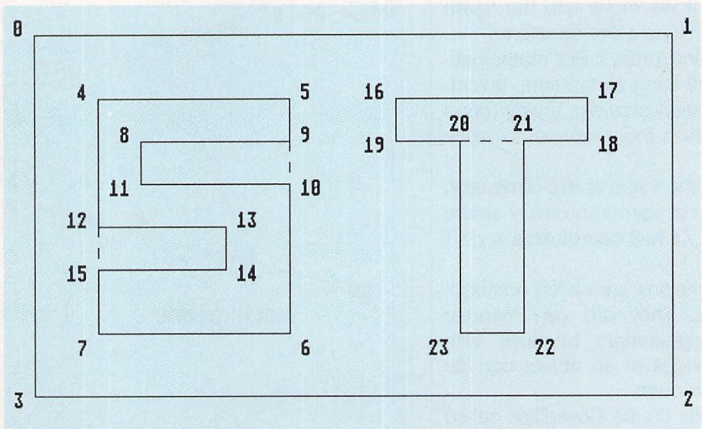

**4. Similar triangles**




**5. A 2D rotation**




**6. Rear view of the ST monolith**




**7. Construction of the ST monolith.**

# Going round in circles

***Andrew Tyler twists and turns objects on the screen***

Static pictures on the screen can be dead boring, but using what we have picked up so far in this series it is possible to achieve quite a lot in the way of animation. So now let's get things moving along.

Rotating an object is a rather strange process, particularly when several rotations are done in succession.

In general, calculating the positions of the vertices of an object after a rotation is a complicated business.

However, we can simplify the task as much as possible by doing rotations about the x, y or z axes. That is what we will look at here.

Figure I *(see page 111)* shows the rotation of a point p to a new position at p' about each of the principal axes.

Different symbols are chosen for the angles for different axes to avoid confusion when dealing with compound rotations about several axes in succession.

One important convention to stick with is the direction of rotation with respect to the axes. Looking at the figure you can see that the direction of rotation is clockwise looking into the axis of rotation.

This is said to be a positive rotation. Naturally, going the other way is negative. When an object is rotated about an axis the vertices which lie far from the axis move in large circles while those which lie close move in smaller ones.

At first sight it might seem difficult to calculate where the vertices move to, but fortunately matrices are ideally suited to this task. Remember, matrices were introduced in Part 3 of the series as a way of carrying out the perspective transform.

The matrices for rotations are rather simpler, being 3x3 (3 rows and 3 columns) in size.

One interesting property of a rotation is that it doesn't change the shape of an object, and this explains why the rotation matrices are so simple.

To illustrate how the rotation matrices are constructed look at the 2D rotation about the x axis in Figure II. The x coordinate of point p is not changed by the rotation, but the y and z coordinates are.

You can see that the y component is reduced but the z component is increased and – because of the circular motion – the sine and cosine of the rotation angle are the important maths. The exact relations between the coordinates of p(x,y,z) and of p'(x'y',z') are:

$$\begin{aligned} x' &= x \\ y' &= \cos\theta \,.y - \sin\theta \,.z \\ z' &= \sin\theta \,.y + \cos\theta \,.z \end{aligned}$$

which can be written as a matrix product

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Rotations about the y and z axes look rather similar except that the order of the rows and columns of the matrix are interchanged. They are rotation about the y axis:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\Phi & 0 & \sin\Phi \\ 0 & 1 & 0 \\ -\sin\Phi & 0 & \cos\Phi \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Rotation about the z axis:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

## TRIG TABLES

In the bad old days before electronic calculators were invented it was usual to use tables to look up precalculated values of complicated mathematical functions.

Instead of a calculator, every student carried a book of tables. This is how we will determine the sines and cosines of rotation angles for the rotation matrix.

It might appear that there is a problem here, since sines and cosines of all angles lie between 0 and 1 and yet the smallest two values in binary arithmetic are 0 (lowest bit clear) or 1 (lowest bit set). Remember that for speed we want to stick with binary integer arithmetic. How then can we handle, for example, the angle 60, for which the cosine equals 0.5?

One solution is to multiply all trig functions by a large factor – rounding off what is left behind the decimal point – while calculations are being done, and then divide out at the factor at the end.

The factor 16,384, which in binary is 2 14, is quite suitable for this, and confines all trig functions to word size. When it comes to dividing out the factor 2 14 this can easily be done in binary by 14 right shifts without the need for time-consuming divisions.

It is very convenient to make sines and cosines word length since they easily fit in the microprocessor registers, especially when mutiplications are being carried out.

This strategy works fine providing certain precautions are taken. Most important is to remember that there is still an error in each cosine and sine, since it is only accurate to that last bit, which is 1 part in 16,384.

This may be a problem in any calculation where an object is progressively rotated, using the calculated vertices from the previous orientation to get the next so that accumulating errors give rise to visible distortion. This is something to avoid.

## A SINE FOR ALL ANGLES

Have a look at the program file DATA_03.S. which contains the sines of angles from 0 to 90 in increments of 1, multiplied by the factor 16,384 as explained above.

Surprisingly, this is all that's required to calculate the sine or cosine – at degree intervals – of any angle between 0 and 360, and is the look-up table we will use.

Why does this work? Put in a nutshell, going from 0 to 360 is like going all the way round a circle. But a circle is an object of high symmetry and it is possible to construct the whole circle from just one quarter.

The first routine in CORE_03.S shows how this is done. For speed, it really makes more sense to have a longer look-up table with the sines and cosines for the complete range 0 to 360, but to illustrate the similarity of sines and cosines it has been done in the way

described. Rotation about a single axis is fine but in real life, rotations are likely to occur about all three axes. This is where the curious nature of combined rotations becomes apparent.

Figure III shows this. In 1 a point p is first rotated about the x axis by 90 and then about the z axis by 90 to end up along the x axis. In 2 the order of rotations is reversed so that it ends up along the negative y axis. Clearly the order of rotations is important.

We need a consistent scheme to deal with combined rotations about the three axes. It isn't sufficient just to list the three separate rotation angles since there is no record of the order of the rotations.

## INTO THE MATRIX

There are several ways of dealing with this problem: Some simple and others complicated. We needn't worry about complicated solutions here, instead we'll take the simplest option.

The simple scheme is to keep a running total of the angles of rotation about the separate axes and then do the rotations in a fixed order.

The advantage is that the three rotation matrices can be multiplied – concatenated, if you want to impress your friends – together beforehand to form a single matrix and then the transform done all in one go.

That is what happens in the example program. What happens on screen as the angles are changed is clearly unique to this order of the matrices but, providing the order is remembered, the end result is predictable.

We will use this transform in a later instalment to construct a quite complicated object.

Look at the file CORE_03.S. You will find in the subroutine a calculation of the nine elements of the transform of a combined rotation which consists of a first rotation about the z axis followed by a second rotation about the y axis and then a third rotation about the x axis.

It doesn't matter if some of the rotations are zero. In that case the appropriate matrix elements are zero the important thing is that we have on hand a transform which can handle combined rotations, which will be needed later.

## GETTING INTO POSITION

If you look at the overall picture described last time you'll see that objects are moved into their positions in the imaginary world inside the computer with an object-to-world transform and then the world is projected on to the screen with the viewing transform followed by the perspective transform.

In this part, life is going to be made simple by assuming that the observer, that's you, is fixed in position at the origin of the world reference frame.

So the world-to-view transform isn't needed since these two frames are coincident. What is left then is the object-to-world transform which places the object at its current position in the world frame.

The overall transform which does this is a combination of a rotation to get it in the correct orientation and a translation to put it in the right place.

Translation here doesn't mean converting from French to English, but is a bit of mathematical jargon meaning move it from here to here.

In our case we take the current position of the object to have the coordinates (Oox,Ooy,Ooz) in the world frame so the overall transform is:

$$\begin{pmatrix} x' \\ y' \\ z \end{pmatrix} = R' \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} Oox \\ Ooy \\ Ooz \end{pmatrix}$$

where R' is the rotation to get it pointed the right way. We can use the rotational transform that has already been worked out. In particular when the rotation angle is changed a bit each frame, the object will rotate.

The translation appears as a separate addition on the right-hand side but it could be converted to a product if we were to go into homogeneous coordinates.

But that would be making work for the sake of it. In the program it is left as a simple addition. This idea of a translation is quite important. If the object were to have a life of its own it would alter Oox, Ooy and Ooz continously from frame to frame.

## THE PROGRAM

Let's have a look at the example program for this part which, taken together with what has gone before, is getting quite long. Remember the program for this part uses the program files from the earlier parts by INCLUDEing them during assembly.

The INCLUDE directive is an instruction to the assembler to add in all the earlier files, which should all be on one disk. The example program shows the ST monolith – a sign with ST written on it – which we showed in perspective last time, rotating about the z axis, which points into the screen.

The rotation is done by decreasing the angle each time by 10 from its initial value of 360 in the control program 3D_04.S.

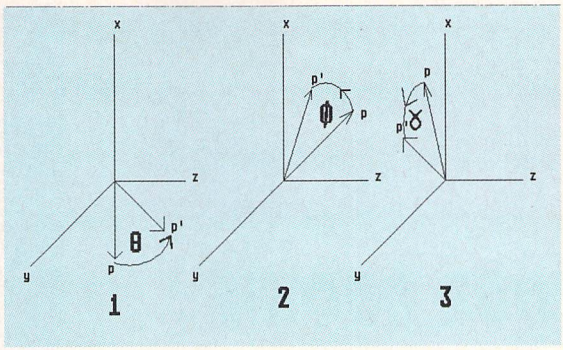
When it gets to zero the angle is reset to 360 and the cycle repeated. Screen buffering is used to give flicker-free motion. Rotating about either the x or y axis would be a problem because that would attempt to show the back of the object.

Since the back is hidden it should not be visible. Removing surfaces which are not visible is called hidden surface removal. That's one of the topics we are going to cover next time.

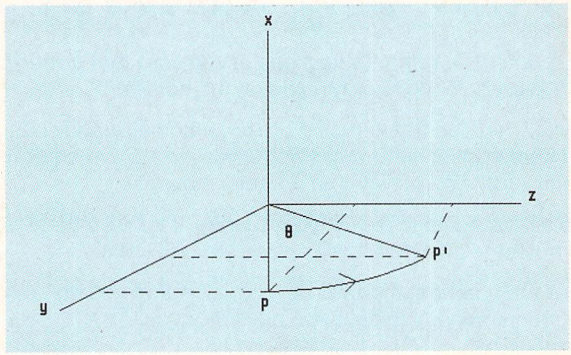
If we really wanted to see the back of the ST monolith it would be necessary to draw the back as a second object, 'pasted' on to the front.

Apart from the new program sections there is no new data this time, since it is possible to use the static ST monolith shown in 3D_03 in perspective. Now it is being rotated.

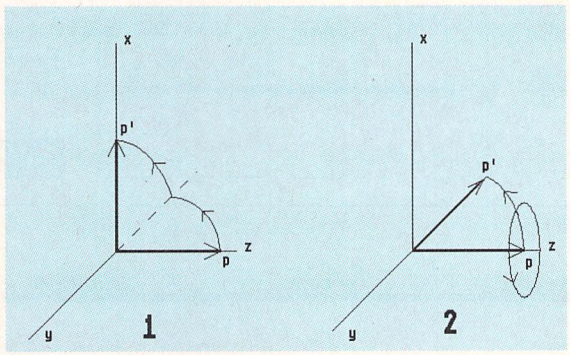
*• This article and programs are based on Andrew Tyler's book RealTime 3D Graphics, published recently by Sigma Press.*




**1: Rotations about the principal axes**




**2: A rotated object**




**3: A rotation in two dimensions**

# Hidden surfaces and illumination

***Andrew Tyler looks at how light and movement affect the appearance of three dimensional objects***

Here we are, poised for a big breakthrough in our graphics "pipeline". Last time, the capability for rotation was put into place. Now we'll add a couple of very important new stages which will add dramatically to the illusion on the screen.

We don't want to draw surfaces which shouldn't be seen – for example on the far side of an object – and we want to add a vital visual clue to the "solid" appearance of an object by illuminating more brightly those surfaces which face a light source.

It turns out that both of these can be done conveniently together.

## HIDDEN SURFACE REMOVAL

A computer is a fast number cruncher, but it doesn't know anything about the real world. When it comes to conveying simple everyday experiences – like not being able to see through solid opaque objects – the computer is a real loser.

Making the computer show this simple fact of life is hard work. It's called the hidden-surface problem and is the basis of some very time-consuming algorithms in computer graphics. In our programs we can make the problem as simple as possible by ensuring that polygon meshes (the joined-together polygons which make objects) are convex, that is, each polygon looks outward and not towards another polygon.

Figure I shows a polygon mesh which is completely closed and therefore forms a polyhedron. The procedure for deciding whether a surface is visible is straightforward in principle: it is visible if it faces the viewer. The problem is to convert the word "faces" into a mathematical expression.

This is done in the following way. Each surface has associated with it a vector which points out at right angles so that the polyhedron as a whole looks like a porcupine, as shown in Figure I. Each of these vectors has the same length, which is chosen to be unity (1). They are called surface normal unit vectors. The only difference between vectors therefore is their direction, which tells us which way they face.

## DOT THE VECTOR

For practical purposes in binary arithmetic, 1 is not a useful size for a vector and so, like sines and cosines, it is multiplied by $2^{14}$ (16384). This keeps quantities within word size and makes multiplication and division simple.

The test to establish whether a surface is visible from the view point consists of seeing whether its unit vector is in the opposite direction to a vector (the view vector) drawn from the view point to the surface. Figure II shows this.

The actual calculation which is done for visibility is called the vector dot product **(V.n)** of the view vector **V** and the normal vector **n**. That sounds very mathematical but what it amounts to is taking the products of corresponding components and adding them:

**V.n = Vx.nx + Vy.ny + Vz.nz**

This curious product has the useful property that it's positive if the two vectors point the same way (both to the left or both to the right), but negative if they point in opposite directions. For a surface to be visible the product must be negative, which means the surface is pointing back towards the viewer.

There's only one thing missing: the surface normal unit vector. It has to be worked out. But the effort is worthwhile since the normal vector is also required to calculate the degree of illumination of the surface forming a light source, which we want to do as well.

## CROSS THE VECTOR

To calculate the unit vector you need to do a square root which, while not being a great problem, does require additional work. For the object we're going to display – the ST monolith where each polygon is a rectangle – it is possible to take a short cut, as discussed below.

The first part of the task is to find the normal vector which points outwards at right angles from a surface. Figure III shows a nor- ➤

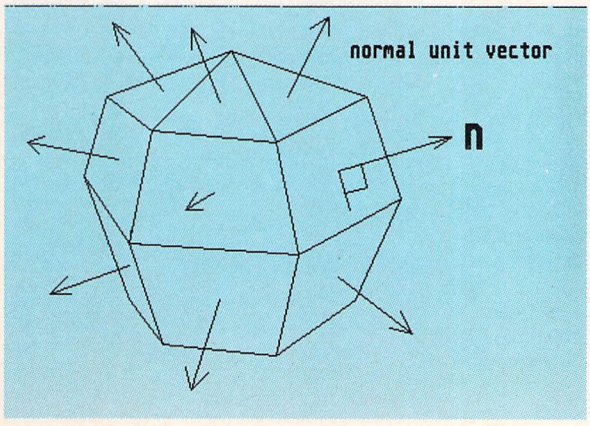



***Figure I A closed polygon mesh, or polyhedron***

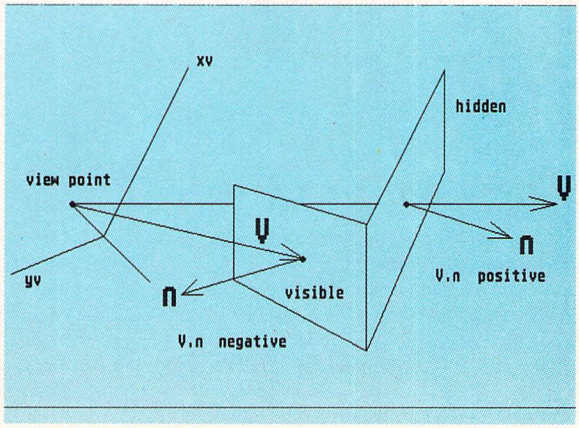



***Figure II The test to establish whether a surface is visible from the view point***

➤mal vector **B** between two vectors A12 and A23 lying along adjacent edges of a polygon.

The way in which B is calculated from the two edge vectors is called a vector cross product; the second kind of product you can do with vectors. Again, this sounds like another complicated piece of maths, but what it amounts to is an odd combination of products of vector components:

**Bx = A12z.A23y - A12y.A23z**
**By = A12x.A23z - A12z.A23x**
**Bz = A12y.A23x - A12x.A23y**

Now to get the unit vector from **B** it is necessary to divide by the size of **B** to end up with a unit vector of length 1. This all sounds a bit daft but remember the unit vector still has x, y and z components which is all we need to see which way it's pointing.

Here's the short cut. It turns out that for rectangular polygons the size of **B** is just the area of the rectangle. Since all our polygons are rectangles we can calculate their areas beforehand and store them in a list, ready to use.

Therefore, having calculated the components of **B** for a given rectangle, a simple division by its area taken from the list yields the components of the surface normal unit vector.

## ALL LIT UP

In 3D, one of the easiest and most dramatic improvements to add realism to a model is illumination by a light source. Facets which face the light source are more brightly illuminated than those which face away. As the object changes its orientation, so changes in illumination give additional visual clues to its shape and structure.

The new information we have to include is a pointer to the direction of the light beam. Figure IV shows how this is done by means of an illumination vector – yet another unit vector pointing along the direction of the light beam.

You might be able to guess what bit of maths will tell us the illumination intensity of a surface. It is the vector dot product of the surface normal unit vector **n** and the illumination vector **I**. This is where we can use **n** for a second time.

Using the dot product this time isn't just a convenient bit of maths which gives the right sort of answer. There's another way of writing the dot product which shows how it is related to the angle Θ between the two vectors. Since both n and I have size 1 it is:

**I.n = cosΘ**

What the dot product really gives, therefore, is the cosine of the angle between the light source and the surface normal. This is very useful because it turns out that the brightness of a surface really does vary this way. So in this case the dot product gives a good approximation to real life.

What actually happens in the program is that the components of the unit vector are multiplied by $2^{14}$ to make them a convenient size to use (like the sines and cosines we met last time) and so the result of the dot product is somewhere between $+2^{28}$ (directly away from light) to $-2^{28}$ (facing the light).

All that remains is to add $2^{28}$ and divide by $2^{25}$ (by right shifting) to produce a result between 0 and $f to give 16 illumination levels which can then be mapped to the colour palette.

In low resolution, which is the most colourful, 16 different colours can be displayed simultaneously on the ST out of a possible 512. This selection of 16 is called the colour palette.

There are tricks to exceed 16 for the screen as a whole by changing the colour palette while a picture is being drawn but we will stick to the basic 16. What we will do is set the palette with different shades of different colours so that the effects of illumination can be seen.

An excellent aid to understanding how the colour palette works is found in the Control Panel Accessory which comes with the ST. This shows three sliders of red, green and blue, each with eight possible settings in low resolution.

## COLOURING IN

This means there are 8x8x8 = 512 possible combinations. Have a play with the sliders to see what can be obtained. Why can only 16 of these be displayed at any one time? Because there are four colour planes for each pixel on the screen in low resolution which, put together as bits, make a colour nibble.

The value of the pixel's colour nibble is then used to index one of the 16 colours in the palette. It only remains to see how the colour palette can be set up from the settings on the control panel.

Figure V shows the connections between the control panel, the palette and the pixel colour. The words which specify the colours can be generated directly in hexadecimal from the control panel settings. For example, a setting of $777 means red=7, green=7 and blue=7. Once the colours have been chosen, they must be listed together and then loaded into the colour palette with a call to the operating system.

For our purposes, in order to simulate lighting, the colours will be different shades of the same colour. There is obviously a trade-off here. The example program uses eight shades of two colours in blue and red (called the intrinsic colours) although other combinations are possible. The colours are listed following the label palette in the file DATA_04.S.

## EXAMPLE PROGRAM

This month's example program on the CoverDisk shows the ST monolith, which has previously been used to show perspective and simple rotation, now in rotation about a vertical axis and illuminated from the right-hand side by a light source.

Also, hidden surface removal occurs so that when the ST points away from the viewer it is not drawn – this is vital for next month's program where a solid object with several sides will be displayed, in motion under keyboard control.

The main control program is 3D_05.S and is the one to assemble and run. It INCLUDEs all the others. All the new subroutines are in the file CORE_04.S, consisting of, in order, the calculation of the surface normal unit vector, a determination of whether a surface is visible and if so a calculation of its illumination.

Finally in the routine set_colr the illumination is combined with the surface intrinsic colour (red or blue) to produce the final displayed colour.

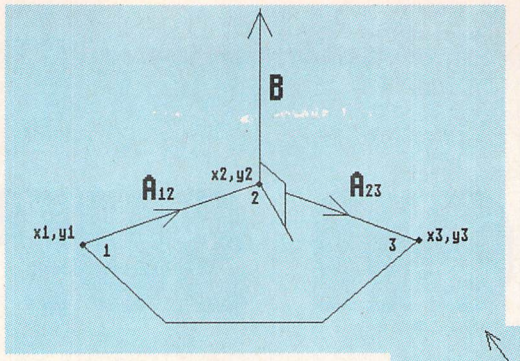



***Figure III***
***Calculating the normal vector which points out at right angles from a surface***

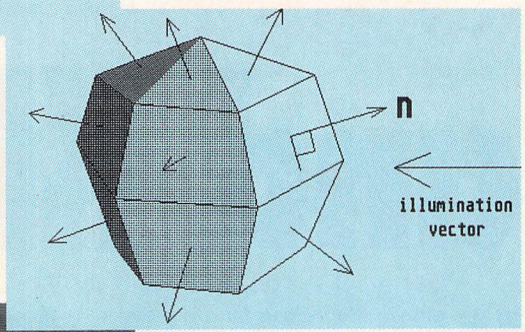



***Figure IV***
***An illumination vector gives a pointer to the direction of the ight beam***

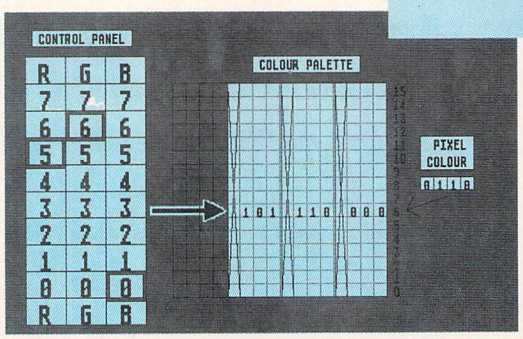



***Figure V***
***The connections between the control panel, the palette and the pixel colour***

### IN THE BOOK

This article and programs are adapted from *Real-Time 3D Graphics for the Atari ST* by Andrew Tyler, published recently by Sigma Press. The book includes everything here plus much more besides.

# Getting it together

***In this, the glorious finale, Andrew Tyler reveals all...***

Well here we are at the end of this series on real-time 3D computer graphics. In this instalment we will use all the features developed in the earlier parts of the series to construct a real 3D object with rotation and movement, controlled from the keyboard, and a fancy shear transform just to show how dramatic effects can be implemented at the press of a key.

The object is a 3D cube – simply six ST monoliths glued together. Hidden surface removal makes sure the back faces aren't seen and a lighting source from the right highlights surfaces so that right-facing surfaces are the brightest. This is very effective when rotation takes place and gives added realism to the whole image.

## INSTANCE SUCCESS

The object we are going to construct here is an ST cube. Figure I shows what it looks like. It is, in fact, six ST faces joined together at the edges. All we have to store in memory is the detailed information necessary to draw one ST face (essentially the same as the familiar ST monolith) together with the orientations of all the faces.

There is a special name given to this process of placing an object in its right position and orientation in the world frame – it is called an instance transform. Constructing the ST cube from the six faces requires an instance transform on each of the faces.

To construct the cube in this way an ST cube face is successively rotated and displaced six times to make each of the six sides. The rotations and displacement of each of the faces are in the lists inst_angles and inst_disp in the data file DATA_05.S.s.

The rotation angles are in the order $\theta$, $\phi$ and $\varkappa$ which are rotations about the x, y and z axes respectively. So for the first face the angles are (0,0,0) and the displacement is (0,0,0), which means lying in the x-y plane. The next face has angles (90,0,0) and a displacement (0,100,0) which means it is rotated from the first face by 90 and then moved 100 along the y axis, and so on. The ST cube isn't really a solid object at all – it's a hollow box.

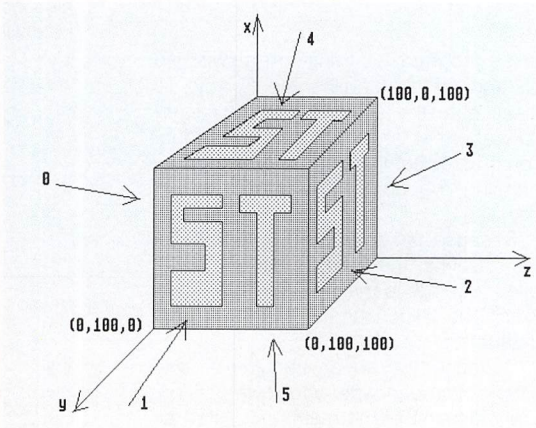



***Figure I. Assembling the ST cube***

## WHERE AM I?

We have used simple rotations in earlier parts to rotate the ST monolith. Now we are going to use them again to rotate the whole ST cube once it is assembled. This time the rotation is done in a slightly different way to let us introduce the idea of a view transform.

Figure II shows you, the viewer, looking at the ST cube. The way the rotation can be implemented is to keep the cube fixed in space and move the observer around.

So, if the observer moves round the cube in an orbit, but always looking towards it, the cube will appear to rotate. This sounds like an awfully complicated way of rotating the cube but it emphasises the equivalence of rotating the observer one way to rotating the scene the other way. Just in case this sounds a bit confusing, consider the following experiment.

Imagine yourself sitting in a swivel chair positioned at the centre of a circular carpet in a room with black featureless walls. Since there is no external reference point – apart from remembering what went on – it is not possible to distinguish between rotating the chair to the right on a stationary carpet, or keeping the chair fixed and rotating the carpet to the left.

You see the same relative movement of chair and carpet, and the view of the carpet pattern from the chair is the same in both cases. There are special names given to these two different transforms: rotating the observer is called a co-ordinate transform and rotating the object is called a geometric transform.

## A SIMPLE WORLD

With a simple world of one object they are equivalent – but opposite. With a complicated world we do not want to move each object separately so we just move the observer. In any case that is just what happens when the observer is free to roam around the world.

In our case, when the observer is moved the world is seen from a different viewpoint and this is what we call the view transform. Figure II shows the details of how to keep a record of where we are relative to the ST cube.

So, if you like to think of it that way, what we are about to do is a view transform from the world frame to the view frame of the observer. In the first part of the program the cube is assembled and placed in its final stationary position in the imaginary world inside the computer, and then we want to see what it ➤

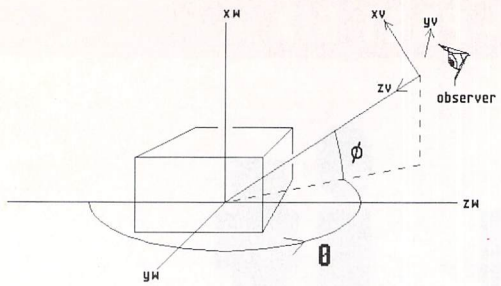



***Figure II. Looking at the cube***

looks like from where we are. We are always a constant distance (100) from the cube but the angles θ and ϕ are variables, changed from the keyboard. In fact the combination of rotations which make up this world-to-view transform is hardly any different from that required to place the object at its final orientation in the world frame (object-to-world transform), but we like to think of them as different so as to distinguish their different functions clearly.

Figure III shows what the observer sees. That's what the cube looks like to her/him, and therefore forms the basis of the perspective projection on to the screen.

We have to invent a strategy for following the motion of the observer in his orbit. There is more than one option. The simplest is to keep a record of the angles θ and ϕ and do just two rotations to get to the final position. It would also be possible to include a "roll" about the line of sight to the cube with an additional rotation about the z axis.

This simple method works OK and is widely used, but it does give a curious orbital type of motion like following lines of longitude and latitude. The angle θ is the angle of longitude and the angle ϕ is the angle of latitude.

Figure II tells us in what order to do the rotations by seeing what to do to get the observer back looking along the zw axis: rotate first about xw by - θ to get into the xw-zw plane and then about yw by - ϕ to get back along the zw axis. If this sequence of rotations is actually applied to the object, then the overall effect is the same.

## THE EXAMPLE PROGRAM

To see how the orbital type motion results from doing things this way, remember the order of rotation is always the same: first θ and second ϕ. So if θ is zero, then increasing θ continuously makes the object appear to rotate from right to left.

But if ϕ is fixed at 90, increasing θ continuously now results in looking at the top face of the cube which spins round clockwise. Confusing isn't it! That's what 3D rotations are like! If the changes to θ and ϕ are controlled independently from a keyboard, joystick or mouse, it's quite easy to get lost if you don't realise what's going on.

In this month's program the two angles are controlled separately from the f3 and f4 keys. You can make the cube move away from or towards you (up to a limit) with the keys f2 and f1. f6 stops the motion and f7 terminateS.

Since rotations can be implemented at the touch of a key, things change on the screen almost instantaneously. There's nothing wrong with that, if it's what you want, but in the real physical world objects don't move instantaneously. What holds them back is their inertia, or mass – some physics creeping in here. To try to inject some illusion of reality into the ST cube, a simulation of inertia has been included in the program. A simple trick has been employed to do this. Instead of the object actually rotating by a fixed amount when a key is pressed, instead an angle increment is increased.

The angle increment is the amount by which the angle is increased each time. If you think about it this means that the object rotates faster and faster the longer a key is pressed. Likewise it takes some time to slow down even after the key has been released. The effect of this looks like inertia.

## SHEAR MADNESS

Here's something flashy you can do with geometry. It's especially easy for us because of the way matrices have been used to do transforms. In fact, if you follow how the next transform is done you can invent your own, sometimes with quite fascinating results.

Rotational transforms are necessary for all motion – other than simple linear displacement. But they aren't the only kind of transform that can be done by matrices. Rotations have the important property that they don't change the volume of an object.

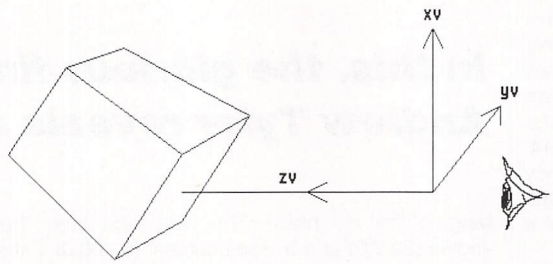



***Figure III. What the viewer sees***

The transform we are about to use does. It squashes the object but otherwise leaves it intact. It's called a shear transform. Figure IV shows a cube after a particular type of shear. It's as if two opposite corners have been pulled apart.

The resulting object is a lozenge-shaped diamond. What has happened is that all x values have been incremented in proportion to their y and z values. The transform that does this is:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

In the program this transform is switched on and off by toggling the f5 key. To make sure the arithmetic doesn't overflow, there is an accompanying reduction in size. But this can be corrected by bringing the object closer with the f1 key.

Because the shear matrix is 3x3 like the rotations, the two of these are multiplied together to make one overall transform. The jargon name for this is concatenation.

Because the shear transform is so simple, you can write your own version. The elements of the matrix are listed as shearmtx, in row order, in the file DATA_05.S. Just rearrange the 1's and 0's and see what you get. Don't use numbers greater than 1 or arithmetic overflow will occur with unpredictable results.

Input to the program from the keyboard is straightforward – the operating system of the ST provides several routines. We use two BIOS routines called BCONSTAT and BCONIN in succession. The first, which we have already used repeatedly to terminate programs, reports whether a key has been pressed, and the second finds out which one by returning the key code in the upper word of register DO.

You'll find the second in the file SYSTM_03.S. The only keys we are interested in are the function keys f1 to f7 which have the codes $3B through $41. In the subroutine key_in in the CORE_05.S file the key code is used as an index to a jump vector to call an appropriate subroutine.

Here is an ST cube which you can rotate, shear, move away from or towards under keyboard control. Don't hold your finger down on a key for long since the buffer isn't emptied and a long press will fill it up with a repeating code.

Copy this month's programs on to your disk with all the accumulated programs from the series and assemble the control file 3D_06.S. It will INCLUDE all the others. An already- assembled file 3D_06.PRG is included in case you have problems.

One last thing. Although the programs have been tested extensively, there is no guarantee that they won't ever crash, particularly if you alter them slightly. Sometimes when a program crashes it corrupts the data on the disk and in memory.

For this reason you should always work with a copy of the files and remember to keep the write-protect window open on the master disk. Following this simple procedure will prevent a lot of unnecessary grief.

## THE BOOK

This series has been adapted from the first part of a new book *Real Time 3D Graphics For The Atari ST* now available from Sigma Press. The book contains all that has been discussed in the series in greater detail, together with many additional topics including input from the joystick and mouse, creating a model world and flying around it under joystick control.

In short, everything you need to start a flight simulator of your own, all in colour, with illumination highlighting and running in real time. *Real Time 3D Graphics For The Atari ST* also contains eight technical appendices to help with the programming.

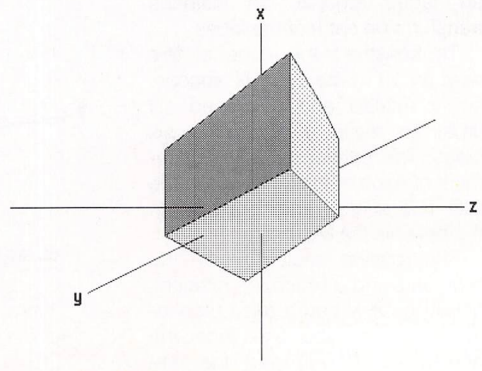



***Figure IV. The sheared cube***