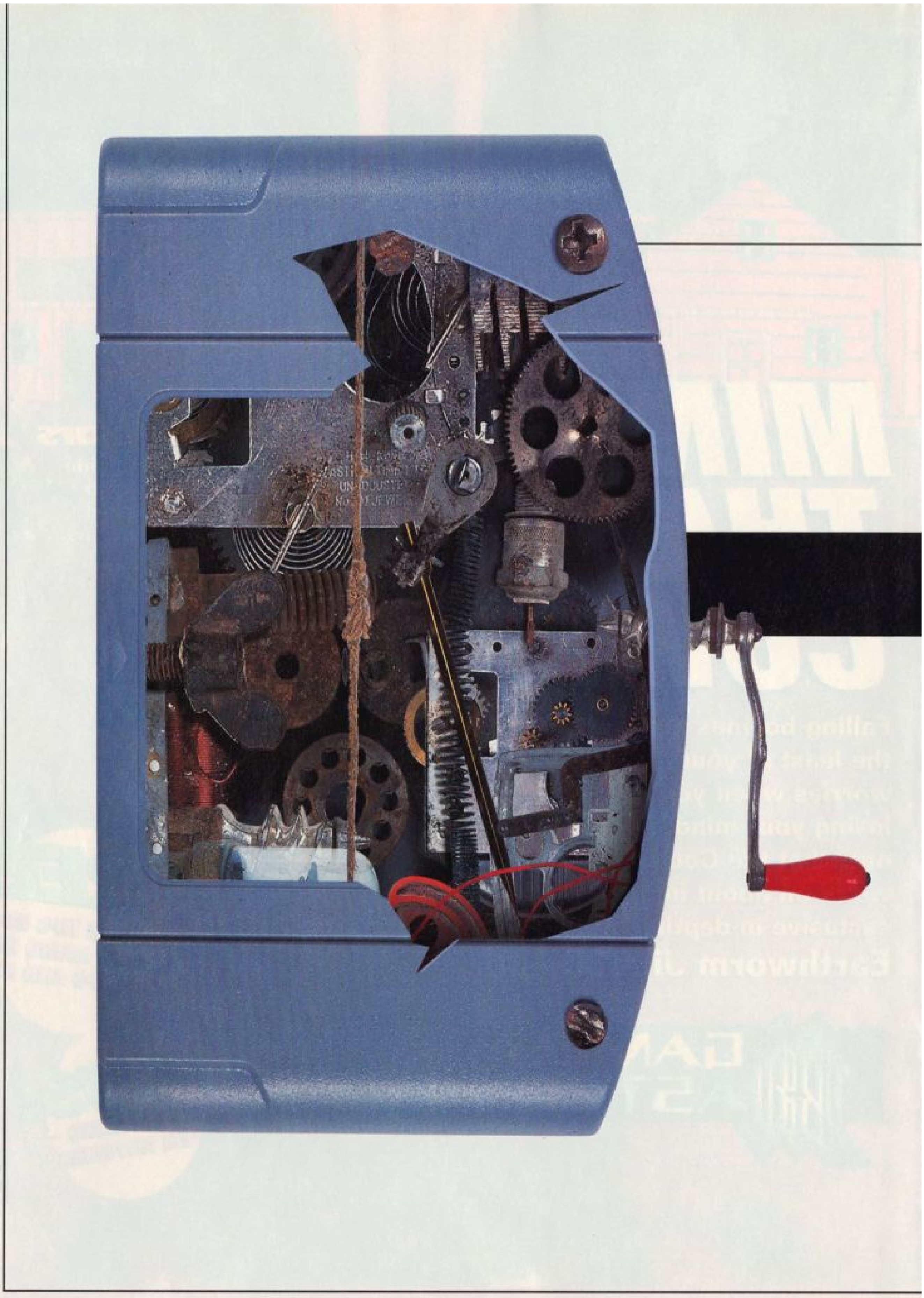
So, how do games actually work?

Article scanné dans le magazine N64 n°07 (Octobre 1997)

Sujet de l'article : Site

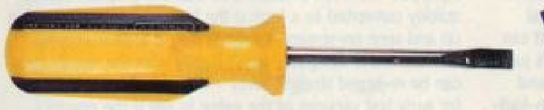
Now there's a question. Luckily, N64 Magazine isn't afraid of hard work, wand we set about discovering just what makes N64 games tick.



SO, HOW DO GAMES ACTUALLY WORK?

Now there's a question. Luckily, N64 *Magazine* isn't afraid of hard work, and we set about discovering just what makes N64 games tick.

by Tim Norris



When you impatiently stuff a cartridge into the slot on the top of your Nintendo 64 and gape at the marvels of the latest generation of video games as they explode onto your telly screen, have you ever wondered, even briefly, how it all got in there? All that sound, all those colours, all those pretty pictures moving around really fast, all those levels, all that, er, 'game'. Obviously it's got something to do with computers, hasn't it?

Much has been made over the last few months, not least by this fine magazine, about the wonders of the N64. It's a Silicon Graphics machine in a little box, they say. The CPU is a 64-bit RISC processor running at 93.75 MHz with another 64-bit RISC processor (the Reality Co-Processor) running at 62.5 MHz just to handle the graphics. It can 'do' anti-aliasing. It can 'do' mip-mapping. It can 'do' astonishingly clever things with digital sound.

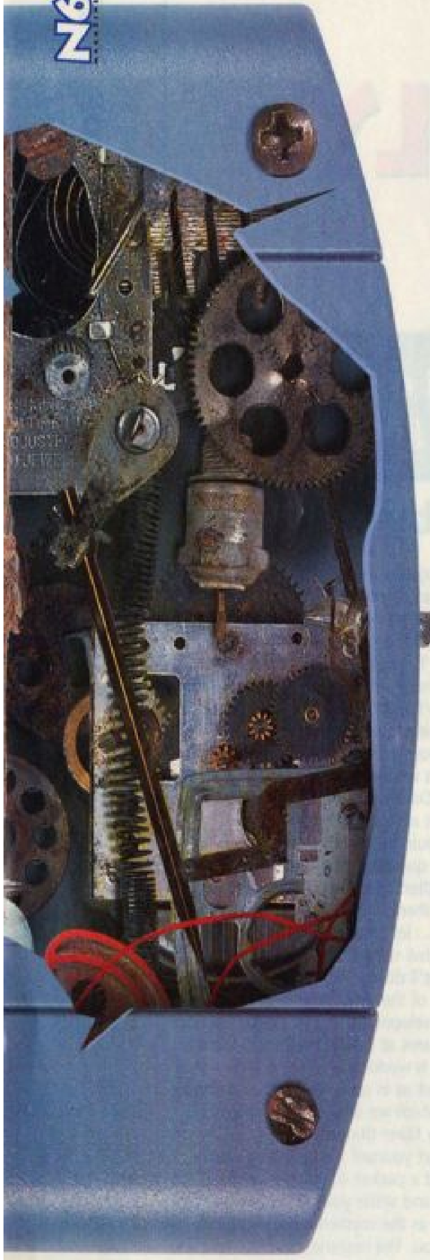
And all that complicated electronics inside your saucy-looking console must do something with some sort of program stored in the cartridge, right? Of course it does, you say. Any fool knows that. Tch.

Well, yes, all right, but what sort of program is it? How did it get to be there? How was it 'developed'? What sort of arcane electronic jiggery-pokery was performed upon the ideas of the game's designers to turn their frantic gibberings into a playable game? ("Yeah, it'll be great, man, there'll be these, you know, like alien killer death monsters on these huge hybrid motorbike-hovercraft things and they'll, you know, burst out from behind these weird concrete trees, except that they'll not be trees but...") What makes the program contained on the chips inside your cartridge any different from the programs we used to tap into the Amstrad CPCs in Dixons in the 1980s to print rude messages across the screen? Eh? EH?

It would be foolish to begin a piece like this with questions like that if it were not our intention to try to answer them. Or some of them, anyway. Well, when we say 'answer'... let's not get carried away. It's a complicated subject and all we can promise is that we'll do our best to shed some light on some of the more interesting aspects of game development. We've spoken to one of the teams at DMA Design (the team in question is working on *Silicon Valley*) and they've let us in on a few of their technical secrets which we shall share with you. Can't say fairer than that.

So get yourself a comforting warm drink and a packet of your favourite biscuits, and settle yourself in the comfiest of chairs as the mysteries are laid bare before you. The material covered here will NOT appear in the exam, but there may be a test later in the week.

GO! GO!



Painting pictures

Any games console, the N64 included, spends the majority of its time constructing the graphics you see on your TV screen. So what goes on in between the programmers deciding there'll be a green elephant in their game, and a green elephant actually appearing in your living room?

The N64 is capable of many visual marvels. It can handle a sufficiently large number of graphical objects and perform upon them such an abundance of effects as to be capable of creating extremely realistic virtual worlds in which we can play. But although it can process the images, it can't create them. It's just a carefully constructed lump of plastic metal and silicon, after all. What does it know about go-kart-driving dinosaurs or Italian plumbers? Nothing, that's what. Someone or, rather, some many, have to draw the pictures and work out what should happen to them as the game progresses. But how, etc...?

If the N64 is just a scaled down Silicon Graphics machine in a living room-friendly box (the new SG workstations wouldn't look out of place in a modern home, it's true, but they're still a bit pricey) it would make sense to design the graphics on a Silicon Graphics machine. So they do.

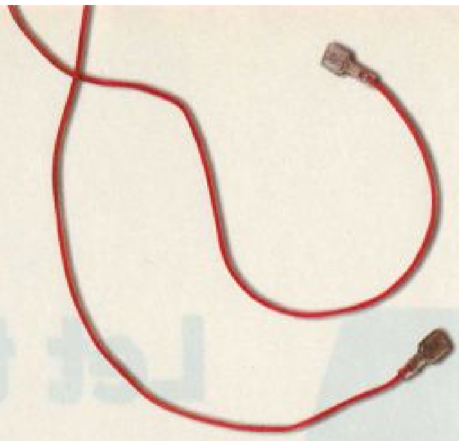
For the 3D polygon modelling, the DMA designers use SG machines running a program called Alias. It is, they say, the sort of high-end 3D software that the film industry has been using for years to create the special effects seen in movies like Jurassic Park and The Mask. Using this kit they can design and animate all the characters and objects in the game and play with them to their hearts' content.

Sooner or later they'll want to see what they look like on an N64, though, and again that's where the link with SG comes in handy. The Silicon Graphics machines are all equipped with N64 emulators so that the graphics can be quickly converted to a format the N64 understands, loaded up and seen on-screen exactly as they'd appear on your console. If the designers aren't happy with the results they can be re-jigged straight away – they don't have to wait for early test versions of the game to see if the graphics work, they just check everything as they go.

Even in these days of hyper-realistic 3D, there's still room for good old fashioned computerised painting. Manipulating images pixel by pixel to create bit-mapped images has a place in the world of the N64 just as it did on the Spectrum, and more conventional software like Adobe Photoshop (which Wil uses here on **N64** Magazine to draw his strange pictures) and Dpaint is pressed into service at DMA to create bit-mapped graphics for textures and the like.

Finally, the developers have their own game editing software which has a built-in polygon editor "for messing with landscapes" (see the section headed 'Building worlds').

When everything has been modelled, drawn, shaped and heartily mucked about with, the images are converted into the correct graphical formats for the game, given names that make sense to the programmers (so the game can call them at the right moment) and, eventually, linked to the program code. And that's it.

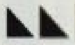


Some of those special graphics features

In the mistaken belief that it'll make us look clever, we've already bandied about phrases like 'mip-mapping' and 'anti-aliasing'. They sound mysterious and impressive, and help to make it appear that the N64 is capable of the 20th century equivalent of magic. Actually, it's all a bit more prosaic than you might imagine.

Anti-aliasing

Let's take, as an example, anti-aliasing. When blocks of contrasting colours overlap on a digital image, the jagged edges of the shapes can be disappointingly obvious. It's partly to do with the



Nasty pixelation occurs on non-anti-aliased shapes. Look!

simple fact that the image is just made of a bunch of square-edged pixels, but there's a certain amount of strangeness where the frequencies of the analogue signals that the television or monitor uses interfere with each other as well. 'Aliasing', it's called.

The results aren't the sort of thing we modern users expect, so something has to be done, and to get rid of the aliasing they use, er, anti-aliasing. The edges of the shapes are blurred using a mathematical filter as the image is drawn on the screen and the results are much more pleasingly realistic than untreated images. Edges seem to blend and blur in a much more natural way.

Computer paint packages have been able to do this sort of thing for years, but the N64 is able to anti-alias images as it displays them, so moving bit-maps and polygons never suffer from unsightly jagged edges.

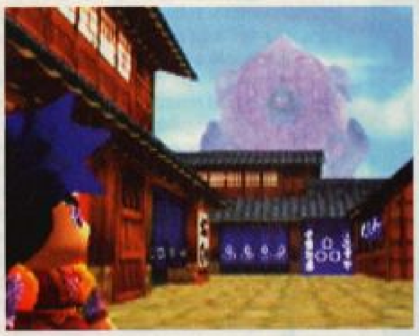
Mip-mapping

And mip-mapping, what's that? If you've been reading since issue 1 you might remember that mip stands for 'multi in partem', which is your actual Latin and probably means something like "in many parts" but don't quote us. (Latin O level was a long time ago – so long, in fact, that it was an and O level not a GCSE.) The full name of the

The more you 'zoom in' on a texture, the more blocky it becomes.

technique is 'tri-linear mip map interpolation', and it's used to change the level of detail on distant textures, again to keep things looking more real (or less unreal, anyway).

Ask any Sesame Street viewer and they'll tell you that as objects get further away they appear to get smaller. Scaling images is no problem for a modern computer, and it can be done very quickly, but you start to get problems when the textures are using contrasting colours and are scaled very small (in other words, when the object is supposed to be very far away or seen at a very shallow angle). When that happens you get 'moiré interference' – that ripply effect you sometimes get on telly when someone's wearing something with very fine stripes. (The name comes from a watered design effect they use on silk and other fabrics, by the way.)



Mip-mapping allows Goemon here to look clearly at things both close up and a long way away.

To avoid this problem, the N64 allows designers to store a number of different versions of each texture, and it decides for itself which version to be used based on the distance from the camera or the angle of view. The designers at DMA working on *Silicon Valley*, for instance, use a set of six versions of each texture at definitions of 32x32 pixels, 16x16, 8x8... and so on to 1x1. They're designed, stored, and labelled, and the N64 just has to choose which one to use to give YOU the best possible effect.

And there's more

And that's not all. The N64 can decide how many of an object's polygons to show ('load management'), allowing far-distant objects to be drawn very small and preventing them from suddenly popping up into view as soon as the machine is capable of drawing them at a reasonable size. It can decide not to bother to draw polygons that are out of the camera's line of sight (the backs of objects, for instance) to reduce the amount of work it has to do and so keep things moving quickly and smoothly ('depth buffering'). It can handle reflections, it can map textures onto 3D objects, it can do that Gouraud shading thing which can make flat polygons appear to be curved (doing away with all the tedious maths that would be needed to move curved surfaces around) and it can create fog.

None of that is exceptionally wonderful – people have been doing all that in PC games for ages. What's wonderful is that the N64 can handle all that for itself – it's built into the hardware. PC games can do all those things, but only in the software, and every little tiny effect has to be handled by the increasingly-knackered CPU. Frame rates slow down alarmingly as you pile on the detail until eventually it becomes impossible to fly your plane/drive your car/whatever because at two frames per second you can't see what on earth is supposed to be going on. The N64 doesn't suffer from that sort of problem because the Reality Co-Processor is handling all the graphics stuff. It makes the designers' work easier, too: they say to the programmers, "We want this object to move that way across the screen and then disappear into the background." And the programmers don't have to work out clever algorithms to do the job, they just tell the machine, "You heard them, do it like they said."

GO! GO!

Let the music play

Games need to assault your ears as well as your eyes, as anyone who's even been asked to turn down *Turok* will attest. But without a CD or a tape recorder or anything, where do N64 noises come from?

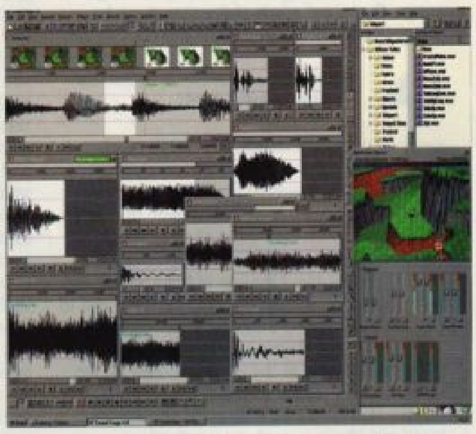
It might be worth taking a few moments to remind ourselves, in the broadest possible terms, what digital sound is all about. Sound travels through the air as waves – small, rapid fluctuations in air pressure. But you knew that. Analogue recording captures these rapid fluctuations and turns them into an electrical signal whose voltage varies with time. It's as if the electrical signal were an image of the sound wave, with the infinitely variable voltage rising and falling over time in the same way as the air pressure. Once you've turned the sound into a signal like that it can be recorded, manipulated, or just re-directed to loudspeakers. Digital recording, on the other hand, involves taking regular, frequent snapshots of the signal and saving each snapshot as a number. What you get is a series of electrical pulses (hence Pulse Code Modulation, or PCM) that give a picture of the way the signal changes over time, which can be used to replay the sound (after some suitable manipulation and filtering, natch).

The level of the signal at the moment of the sample is recorded as a number (this is called 'quantisation', jargon fans). Obviously, the more numbers you've got to choose from, the more precisely you can record the level, which is where the notion of more bits = higher quality comes in. If you use an 8-bit binary number to record the signal you've only got a choice of 256 possible divisions, and the results will sound a bit crap. Any sample whose level falls between two numbers will be rounded to the nearest one and you get 'quantisation error'. To minimise the unpleasantness you can increase the width of the bit field, and the N64, like CDs, uses a 16-bit system giving 65,536 divisions. It sounds sufficiently like Real Life as to be indistinguishable by mere human ears.

Obviously, if you can record sound digitally you can synthesize it as well, and the sound and music designers use a combination of the two to produce both the spot effects and the music for N64 games. The music world has been using MIDI (Musical Instrument Digital Interface) for years. It's a way of recording and communicating everything about a musical performance except the actual sound, which is reproduced by a synthesizer. This means you can write a piece of music and define everything about its performance – the length and pitch of every note, as well as some clever effects – and give it to someone else to perform on their synthesizer. There's much less data to be stored than with a sound recording, and it can be edited and performed any way you choose, using any instrument sound you synth is capable of making.

So, the kit inside the N64 processes digitally recorded sounds as well as synthesising wave forms to be played by MIDI-type data, and that's where they get all those great beeps, thumps, tunes and whistles.

At DMA they use the same sort of 'real' musical instruments you'd find in a recording studio. A computer running a program called Emagic Notator Logic is connected through a MIDI interface to a Peavey SP synth which has been programmed to emulate the N64. In this way the score can be composed and played by real musicians (and not the tone-deaf programmers responsible for many of the 'classic' game tunes of old). Sound effects are played from a 200-disc CD changer (try getting one of them into the boot of a BMW) containing FX CDs from the likes of Twentieth Century Fox and Hanna-Barbera.



△ *Here, Forge is being used to manipulate a number of different game noises (represented by the squiggly up and down lines). It looks a bit like a multi-hearted monster connected to an electro cardiogram.*

When they've got the sounds they want, everything is digitally transferred through a Turtle Beach Pinnacle sound card onto a Pentium PC running more sound manipulation software, this time Sound Forge. The sound track is modified, mixed, re-sampled, converted and generally mucked about with until they have exactly what they want. And then they put it in the game. Simple.

One of the N64's more exciting features is that it's actually capable of 'creating' music as it goes along. No-one has made use of this little trick yet, but it should be possible to get some interesting and unique soundtracks for games that are, for example, entirely dependent upon the way you're playing the game. Or the time of day. Or anything. Fascinating stuff.

Making it all work

So the graphics are all designed and the music sounds fantastic. What's they need now is to be turned into a game, with controls, explosions, scoring and a plot. It's time to call in the Programmers.

A little history for you. The first complete computer program was written by Ada Byron, Countess of Lovelace (and daughter of Lord Byron, the famous poet and wearer of shirts with big sleeves), in 1835. She wrote it on punch cards, and it would have run on Charles Babbage's Analytical Machine if only he'd been able to build it.

Since then (1835 – we can't get over that) there have been many computers, and many means of programming them. Things took something of a backward step during the 1940s when the first electronic computers were built, and the idea of putting programs on punch cards was replaced briefly by the need to switch switches and re-plug plugboards, but it soon settled down and programmers returned to the lovely and entertaining task of writing down all the 1s and 0s that computers feed on. Machine code programming was born.

Assembly language (using difficult-to-remember mnemonics and abbreviations for machine code instructions which are then 'assembled' into machine code) isn't much better, and it wasn't until the late 1950s that the first of the high level languages came into use. Languages like FORTRAN, COBOL and ALGOL used (use, indeed) English words and phrases to build their instructions. When the program is complete it's 'compiled' into machine code, and the computer attempts to run it. They're easier to use because they can be more easily understood by real people and don't require an intimate knowledge of the workings of computers. Over the years a number of high level languages have been developed and each has found a use in some branch of computing. (Except Pascal, which was taught to engineering students as a cruel joke – ahahaha, how we laughed.)

Of them all, a surprising survivor is C. Surprising, not only because it has survived as the dominant programming language of the microcomputer since its invention in the early 1970s but also because its designer, Dennis Ritchie, chose to give it such an appallingly uncharismatic name. It's called C because its predecessor was called B. Splendidly imaginative. (Some genuine imagination was called into play when its own successor was named, though: that's called C++. It's not much more exciting, but it was at least unexpected.)

The Internet's on-line dictionary of hackers' jargon (there's a mirror at http://beast.cc.emory.edu/Jargon30/JARGON.HTML, but there are others) has this to say: "C is often described, with a mixture of fondness and disdain varying according to the speaker, as 'a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language'." Another reason to wonder why it has survived as long as it has.

Still, survive it has, and the programmers at DMA use it exclusively. They say, "We use C for all our programming basically because of speed. C compilers these days can optimise code amazingly well, making the code almost as fast as if it were written in pure assembler. We could, of course, write our game in assembler, but then it wouldn't be released until 2010." C might be hell, but when it's compiled into machine code it's neat and clean without too many redundant or inefficient lumps of code to slow things down.

Code can be compiled and tested on the N64 emulators in the same way as the graphics, and if it works there are parties and rejoicing all over the land. Probably.

A bit of Silicon Valley

And what does C code actually look like? Here's some code they prepared earlier (again from *Silicon Valley*, of course – sorry it's a bit scrunched up) to give you a brief flavour:

```
//////////////////////////
//////////////////////////
//////////////
// NAME : pl_UpdateDeath()
// PURPOSE : Checks if
player is dead. If so
sets up appropriate vars.
// RETURNS : Nothing.
// PARAMETERS : Nothing.
//////////////////////////
//////////////////////////
//////////////
void pl_UpdateDeath( void
)
{
   if (gsPlayerInfo.Dead
== PL_DEATH_COUNTER &&
(!gPlayerIsDead))
   {

wp_StartWipe(WP_TYPE_FADE_
OFF);
   }

   // Has Player been
dead for wait amount ?
   if (gsPlayerInfo.Dead
== PL_DEATH_FADE_COUNTER
&& (!gPlayerIsDead))
   {
   // Ahh, loose a life.
   gsPlayerInfo.Lives-;
   }

   // Update Lives info.
   if
((gsPlayerInfo.Dead>
PL_DEATH_FADE_COUNTER) &&
(wa_WipeFinished()))
   {

   // Tell game to reset
energy.
   gPlayerIsDead=TRUE;

   // Set last level to
stupid val so we load a
new level.
   gLastLevel=99;

   // Player is no longer
dead.
   gsPlayerInfo.Dead=0;

   // Fade our music out
   SetSeqpFading(2, 6,
20, 0);
   SetSeqpFading(3, 6,
20, 0);

   }

   // Game Over ?
   if
(gsPlayerInfo.Lives==0)
   {

gGameAction=GA_GAME_OVER;
   gNextWave = TITLEWAVE;

gStartLevel=gsPlayerInfo.L
evel;
   }
}
```

It's important, should there ever be a need to debug or upgrade the code (or even rip it off for another game), that people shouldn't have to read through the whole thing working out what every line does so, as you can see, at least half of the work is in writing the comments (the lines starting with //) to describe what each command does. But as they say, "If you can't understand it then we're not doing our job properly."

GO! GO!

Building Worlds

So that's the graphics done, and the sound, and they're working together as a game. But wait! We've completely forgotten to design a world in which the action can take place. Quick...

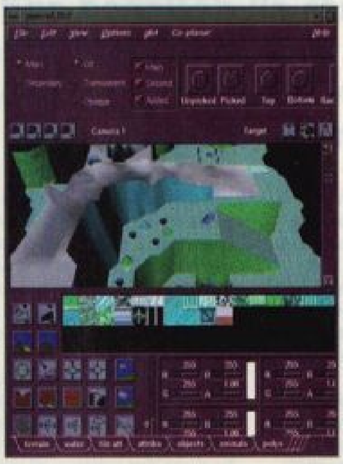
While all that excitement with computers and expensive software is going on elsewhere in the building, the game's levels are designed in a sleepy corner, tastefully appointed with comfortable antique furniture. And hammocks. High Technology has been sent out to buy some biscuits, and a pleasing calmness descends upon the creative process. HB pencils and large sheets of clean, white paper are placed upon desks and the game levels are designed WITHOUT THE AID OF COMPUTERS. It's astounding in this day and age that anything at all, never mind the design of something as complex as a computer game, could be achieved in so rudimentary a method as drawing it on a sheet of paper (with a pencil, for goodness' sake).

Still, as they say, "If a level can't be played through on paper then it won't play through in the game," so a great deal of time and effort can be saved from the outset merely by not trying to create levels that have no chance whatsoever of working.

So, you've got a game concept – a main idea for everything to hang on. You've got designers working on landscapes, objects and characters (or animals, or monsters, or whatever). The musicians are composing a sound track and designing sound effects. There's a programming team working to tie the graphics and the sound together inside the game concept, making sure baddies behave in the right way, that the right things happen when the player is shot/bitten/belayed about the head and neck with heavy blows from a wooden club. But even then, even when everything seems to have been done, the individual levels still have to be designed. And they do it on paper.

Well, they do it on paper at first, but when they're sure the paper model works High Technology is summoned back from its shopping trip and they call upon the mighty power of SVEN. SVEN is a two-metre tall Norwegian timber laminator with biceps the size of Oslo airport, who... No, wait, that's not right. SVEN is a game editing program that runs on Silicon Graphics computers. Yes, that's more like it.

Assuming that everything else is working more or less okay, SVEN can be used to build landscapes, place objects and set up baddies for entire levels. Once all that's done it can generate the level and put it onto the game for immediate testing in one of those ever-useful N64 emulators. Any changes needed? No need to strip out the code and re-program the level, just change the parameters in SVEN and have another go. It is, the boys at DMA say, a monster package, and it allows them to change almost every setting in the game from the comfort and familiarity of a nice graphical interface without having to get their hands dirty over and over again with all that tedious computer code nonsense.



△ A program built especially to manipulate computer games, SVEN doesn't come cheap. It requires a super-charged monster of a computer to run as well.

Is that it, then?

More or less, yes. Creating N64 games is a skilled and complex business, and one which involves a distressingly large amount of capital kit. In the early days of computer games, any kid with a £99 Spectrum could write a best-selling game. And many did. As time wore on, home machines became more expensive, but it was still within the average punter's financial power to buy a computer and a programming language and have a go for themselves. Things became more difficult as the 16-bit consoles took hold, but still all you really need to create SNES or Mega Drive games is a decent PC and an emulator.

And now? To even begin to work on an N64 game you need: at least one Silicon Graphics computer; N64 emulation software; 3D modelling software and at least one image manipulation package; sound manipulation software, MIDI authoring software, synthesizers and a huge library of effects CDs; level designing software; a copy of C (with a manual); and as many talented designers, musicians and programmers as you can find. And all that before you can even begin to think about the actual game.

The video games industry isn't just a hobby for nerds any more, it's an expensive business. People bemoan the dwindling numbers of independent games developers producing exciting and innovative games in their spare time, but it's the price we pay for technical wonderfulness. Luckily, the more successful of the established independents, companies (DMA Design and Rare, for example) have got the capital to invest in the tools and people they need to get into N64 development. From people like this, we can expect to see games designed by people who care about games, rather than games designed by people who want to ride the gravy train and imagine that pretty pictures will be enough. Let's hope it works out that way, eh?