Computer Graphics during the 8-bit Computer Game Era

Steven Collins*
Image Synthesis Group
Department of Computer Science
Trinity College Dublin, Ireland.

1 Introduction

The technologies being employed in current games have advanced to the point where computer game companies are now leaders in graphics research and indeed the requirement for realistic real-time graphics has arguably driven graphics research in areas such as image based rendering and visibility processing. This article will explore the early 8-bit computer industry (from about 1982 to 1990) and in particular the graphics architectures, algorithms and techniques being employed at that time in computer games. Rather than attempt a complete review of all the machines available at the time (including the coin operated cabinets, the dedicated home games consoles and the more general 8-bit home computers), I'll concentrate on what I know best, the Commodore 64, which was inarguably the most successful of the 8-bit machines but will also have a brief look at the Atari 400/800 and Sinclair Spectrum for comparison.

In later sections, I'll outline the architecture of the 64's graphics sub-system (and compare it with some of its main rivals), list some of the graphical techniques used in different genres of games and will also explore some of the more esoteric effects that can be squeezed from the 64 by exploiting quirks of its video chip. First, though, we'll look at the birth of the industry which has surpassed even the movie industry in annual turnover.

2 A Brief History of Time

Most people would probably associate *Pong* with the first computer game. In fact, as early as 1962 a game called *Spacewar* was created at MIT on a PDP-1 and there is some evidence of a *Pong*-like game which ran on an oscillascope at Brookhaven National Laboratory in Upton, New York. The first commercial computer game[1] was *Computer Space* which appeared in November 1971. It never really sold, however, and by the time production ceased a total of only 3,000 machines had been made. An excellent chronological archive, *I.C.When* [1], is a great source for historical information regarding the computer industry with particular reference to computer games.

2.1 The Coin-Operated Arcade Game

The first real commercial success was *Pong*, created by Nolan Bushnell in November of 1972 who, with fellow electrical engineer Ted Dabney, formed a company called Syzygy which later became Atari. Atari, now synonymous with the early game industry, had huge successes with *Pong*, *Asteroids*, *Missile Command*, (*Breakout* designed incidentally by Steves Jobs and Wozniak before they began Apple) and created the home game console market with the Atari 2600. Other manufacturers quickly followed suit including Bally/Midway and Taito (responsible for *Space Invaders*), and Williams (who released classics that included Eugene Jarvis' *Defender*, *Robotron: 2084* and *Joust*). For more information see the excellent *Videotopia* website [2].

2.2 The Home Computer

The golden era of the 8-bit computer game began around 1982 and continued until about 1990. Following on from the success of hobbyist computer kits (like the Altair and the Sinclair ZX-80) a number of computer

*Email address: Steven.Collins@cs.tcd.ie, web address: http://isg.cs.tcd.ie/scollins/

[1] This was an *arcade game* taking the now familiar form of an upright cabinet with dedicated computer hardware and black & white display.

Machine	CPU	RAM	ROM	Resolution
Atari 800	6502	48K	8K	320×192
BBC Model B	6502	32K	32K	640×256
Commodore 64	6510	64K	20K	320×200
Dragon 32	6809	32K	16K	256×192
Jupiter Ace	Z80A	3K	8K	512×368
Lynx	Z80A	48K	16K	248×256
Oric 1	6502A	48K	16K	200×240
TI 99/4A	9900	48K	16K	256×192
VIC-20	6502	5K	16K	no hires mode
ZX-81	Z80	1K	8K	64×48
ZX Spectrum	Z80	48K	16K	256×192

Table 1: A summary of a selection of the large range of 8-bit home computers that appeared on the market from 1982.

companies simultaneously released a range of powerful pre-assembled home computers, epitomised by the Commodore 64, the Sinclair Spectrum, and the Atari 400/800 (which I will simply refer to as the Atari 800 or just Atari; the 400 has a smaller RAM specification and hat a flat touch sensitive keyboard). In fact, there were many more contenders and a summary of these is given in Table 1.

The market leaders were the 64, the Spectrum and the BBC (in Europe) and the Atari (in the U.S.). Only Amstrad were able to make a major impression capturing some of the market share in the mid-Eighties with the CPC-464, but until the advent of the 16-bit machine (heralded by the Commodore Amiga and the Atari ST) the Commodore 64 was the most popular home computer. Its popularity was almost certainly due to the graphics and sound capabilities rather than O.S. (the 64's implementation of BASIC was notoriously bad) or speed (the processor is clocked slower than most of its contemporaries). It was a simple matter to achieve basic animation effects (the 64 has hardware support for *sprites*[2] and scrolling), and so it encouraged experimentation and an entire generation of programmers became familiar with the architecture and began to push the boundary of what was possible.

3 The Machines

The graphics sub-systems of the 3 main 8-bit computers were quite different:

[2] Sprites or MOBs (moveable object blocks) are small graphic elements of fixed width and height that may be positioned independently of the main screen and were provided for the implementation of moving characters in games. See Section 3.3.2 for more detail.

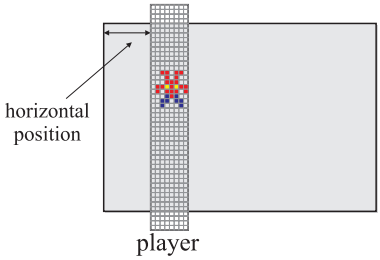

Figure 2: The Atari Player Missile (PM) graphics.

3.1 The Atari-800

The Atari (see the Planet Atari website [3] for more information) had the most powerful graphics system which is not surprising given the machine's lineage. The GTIA chip (George's Television Interface Adaptor) provided hardware support for sprites (called *Player Missile Graphics* or simply PM graphics), a large number of video modes and a display list processor, the ANTIC, allowing mode changes per raster line for advanced display effects. Both chips are memory mapped and have a large number of registers controlling their operation.

3.1.1 PM Graphics

Five 8-pixel wide columns can be displayed at varying horizontal positions (see Figure 2). These columns spanned the entire height of the display (i.e. 192 pixels). To move a player's graphic horizontally, the horizontal position register of the player was updated. For vertical movement, the bitmap data associated with the player was shifted in memory. The fifth player sprite may be optionally split into 4 2-pixel wide sprites each with independent horizontal control. These were designed for displaying missiles. This arrangement was ideally suited to certain types of games (particularly the *Space Invader* genre a good example of which is *Galaxians* shown in Figure 1(c)).

Inter-sprite and sprite-background priority could be specified and the GTIA chip would detect all collisions between sprites and background and latch these in registers, indicating the sprites which had been involved in the collision. The implementation was more flexible than that of the Commodore 64 in which a single bit registered sprite-sprite collisions and another flagged sprite-background collisions, requiring further testing of extent overlaps to determine which sprite had

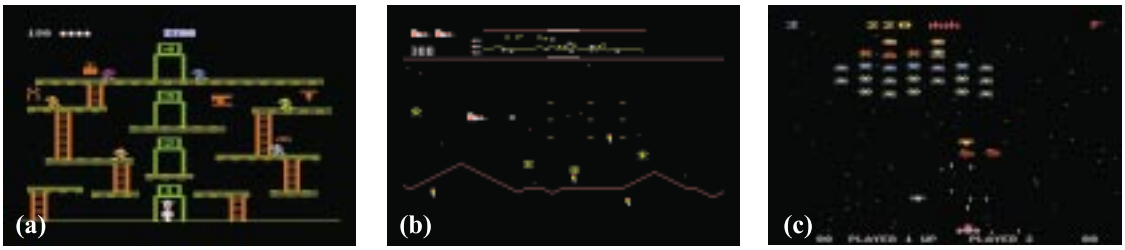

Figure 1: Classic Atari titles: (a) *Miner 2049er*, (b) *Defender* and (c) *Galaxians*.

been involved. This is analogous to the *broad* and *narrow* collision detection phases in use in most physically based animation systems [4].

3.1.2 Display Lists

The ANTIC chip was responsible for interpreting the display buffer for the GTIA chip. It was the ANTIC chip that determined the resolution and number of colours available on the display and it did so by selecting one of a large number of both text based and bitmap graphics modes. Unique to the Atari, however, was the *display list*, which later became an integral part of the Commodore Amiga's graphics architecture (the Copper chip of the Amiga provided functionality similar to Atari's ANTIC chip). The display list was a list of commands interpreted by the ANTIC chip and accessed via DMA (during which time it asserts control over the address bus by issuing a HALT signal to the 6502 CPU). Each command of the display list was capable of selecting one of the 16 display modes (which determines the resolution, number of colours and the interpretation of the display buffer). The display list had its own flow control implemented using *jump* commands so effectively the ANTIC chip was a processor operating in parallel with the 6502. Potentially, each line of the display could have its own entry in the display list, thus allowing selective control over each raster line. Thus many modes could exist on the screen at the same time (called *screen splitting*).

Hardware support for scrolling was provided through X and Y *scroll registers*, allowing the entire display to be shifted in the horizontal or vertical direction by up to 15 pixel positions. For larger scrolls, the display data was shifted in memory. Each display list command could enable/disable scrolling for its associated line, thus allowing split screen scrolling. Finally, each display list entry was capable of flagging an interrupt request, thus control may be passed to the CPU when the raster scan reaches a certain point in the display, facilitating synchronisation of the display and the software. These techniques were also in common use on the Commodore 64, but significantly less support was provided and display list functionality could only be emulated in software. See Section 5 for more details.

Depending on the mode, a number of colours could be displayed on the screen selected from a palatte of 16 hues. Uniquely, the brightness of these colours could also be specified (there were 8 luminance settings) giving a total palatte of 128 colours.

3.2 The Sinclair Spectrum

The Spectrum (originally known in the U.S. and Canada as the Timex/Sinclair) distinguished itself by having no hardware support for sprites which became a major stumbling block for graphics programmers developing for the machine. In fact, the Spectrum was a marvel of minimalist engineering, lacking even a dedicated video chip. All video I/O is performed via an ULA which controls the lower 16K of RAM, of which 6912 bytes are used for the display buffer. The is made up of a 256×192 bitplane with a 0 representing a pixel to be coloured with the background colour and a 1 indicating the use of the foreground colour (as is the case with many of the other 8-bit machines, including the Atari and the Commodore in certain modes). However, an ATTR buffer of 768 bytes encodes unique foreground and background colours for each 8×8 pixel square (and also selects between 2 brightess values and toggles flashing). Thus from the 16 available colours, 2 were selected into each block.

As a result it is very difficult to avoid *colour bleeding* artifacts (and unlike in radiosity solutions, these are to be avoided). When an animating character (implemented usually as arrays of 8×8 pixel blocks) moved

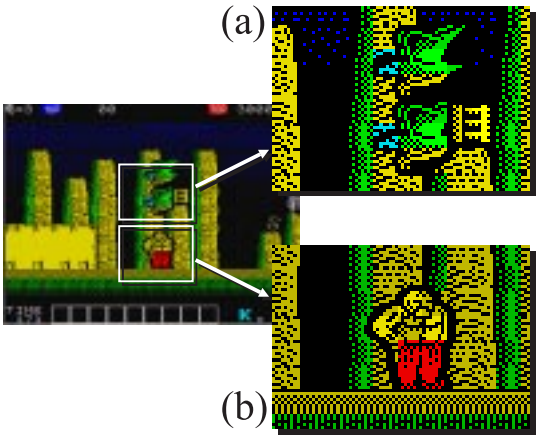

Figure 3: Problems arising from the 2 colour per block limit on the Spectrum (using *Karnov* as an example): (a) colour bleeding from the bird characters, (b) large surround around the main character.

smoothly across a background, if the character was a different colour to the background, it was often impossible to serve the colour requirements of both character and background graphics within single blocks, thus usually the character colour was used both for foreground and background graphics and so the character colour appeared to have bled into the background (see Figure 3). To minimise this, many games either a) avoided colour altogether, b) confined animation steps to multiples of 8 pixels in any direction or c) created a thick border surrounding the character to minimise the effect of bleeding.

As with the 64 and Atari, it was possible to synchronise the software with the display using interrupt handlers invoked in response to ULA interrupts to prevent flicker (see Section 5.1). For more details see the *Planet spectrum* website [5].

3.3 The Commodore 64

The 64's graphic capabilities were provided by MOS Technologies 6567/6569 VIC II chip (Video Interface Controller). These devices were originally designed for cabinet based games and graphics workstations and had excellent graphics capabilities, surpassed only by the Atari's GTIA/ANTIC devices. The VIC device supported 3 character based display modes, 2 bitmap modes, 8 hardware sprites, hardware assisted scrolling, a palette of 16 colours (no luminance control) and light-

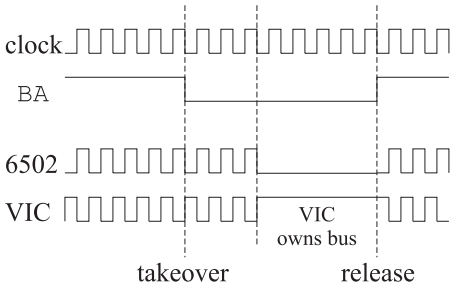

Figure 6: Bus arbitration on the Commodore 64. Normally the 6510 and the VIC accessed the bus on alternate clock phases, but the VIC could take over the bus for certain periods (by asserting pin BA), locking out the CPU. There was a short delay between the change in the BA line and the VIC actually asserting control to allow the CPU to finish any pending writes to RAM.

pen support. As with the Atari, the device was memory mapped and addressed 16K of DRAM. It had a 12-bit data bus to allow simultaneous connection to main memory (8-bits) and 4-bit static RAM which contained the colour information for the screen.

Since the launch of the Commodore 64, the VIC chip has been reverse engineered to the point where probably every nuance of its operation is now understood. This has allowed programmers to take advantage of some quirks of the design which facilitate certain graphical effects that would be impossible to achieve through software alone. We'll examine some of these in Section 5.4. For general information regarding Commodore products and software visit Jim Brain's *CaBoom* website. For more details about the inner workings of the Commodore 64 visit the CBM Document Page) [6] and for an excellent review of the VIC chip functionality read Christian Bauer's technical article [7]. Christian is the designer of *Frodo* an excellent Commodore 64 emulator.

3.3.1 Video Memory

The VIC addresses a total of 16K which was composed of a number of registers, character RAM and display RAM. The VIC always had access to colour RAM via the hardwired connection to the upper 4-bits of it's data bus. The 6510 processor and the VIC both required access to memory and so the 64 implemented a simple bus arbitration scheme: they shared successive bus phases. The CPU had control of the bus during positive

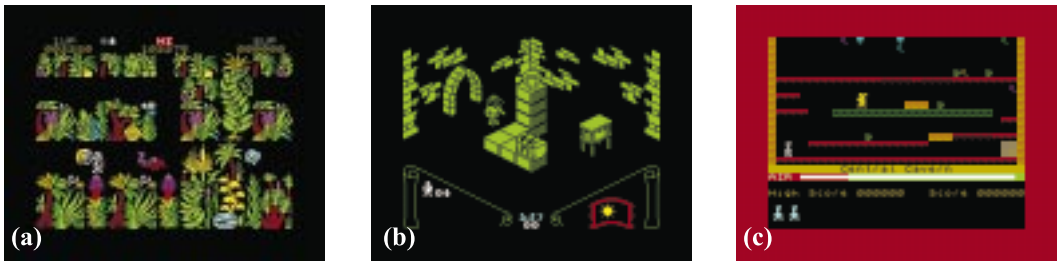

Figure 4: Classic Spectrum titles: (a) *Sabrewulf*, (b) *KnightLore* and (c) *Manic Miner*.

Figure 5: Classic Commodore 64 titles: (a) *Ghosts 'n Goblins*, (b) *Impossible Mission* and (c) *Paradroid*.

clock phases and the VIC had control during negative clock phases (see Figure 6). At certain times, the VIC needed to access RAM for longer than one half clock period and so it asserted full control over the bus, effectively locking out the CPU. It is this locking out that can cause synchronisation problems during timing critical operations (e.g. disk accessing) and so for certain operations the VIC device was disabled (thus disabling the display). Unusually, the 6510 *always* accessed memory during positive clock phases even if executing instructions which do not require access to the bus, and so it was not possible to use the CPU when the VIC asserted control over the bus.

When in a *character-based mode*, 1000 bytes of screen memory were used to specify the character symbol to use in each of the 40×25 character positions. Each character was itself a block of 8×8 pixels. The default character set was available in ROM, but the VIC could be pointed at RAM to allow the creation of user defined characters. The VIC could generate and display 256 such characters. The foreground colour for each character position was supplied by the colour RAM mentioned earlier.

In *bitmap mode* a full 8000 bytes was used to address the 320×200 pixels of the display. To facilitate cheaper implementation via the VICs memory scanning architecture, the bitmap data was arranged rather unusually

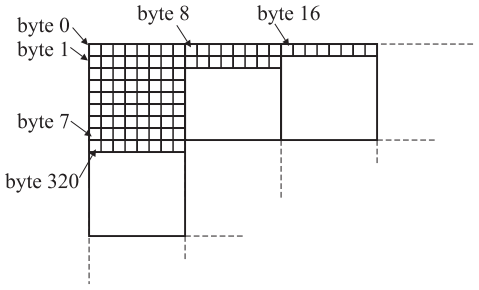

Figure 7: Display memory layout in bitmap mode. Display bytes were ordered as the would be in character mode to simplify the implementation of the scanning hardware in the VIC chip.

as shown in Figure 7. As can be seen, this arrangement was similar to the memory scanning sequence the VIC would adopt for character based modes.

3.3.2 Sprites

The 64 had 8 independent sprites, each being a block of 24×21 pixels (i.e. 63 bytes of graphics data per sprite). Unlike the Atari, the 64's sprites were free to move both horizontally and vertically. The VIC resolved collisions between sprites and between sprites and screen data and latched this information in registers

to be read by the software (or would raise an interrupt if enabled). Sprites, like the Atari's PM graphics, could be stretched vertically and horizontally by a factor of 2. Display priority was fixed between sprites, with sprite 0 always in front and sprite 7 to the back, but priority with the screen data could be specified by the user, allowing for basic depth effects exploited in many 3D games (see Section 4.1 for more details). Unlike the Spectrum, sprite colours were managed independently of the background graphics and so there were no colour bleeding artifacts.

3.3.3 Scrolling

Hardware scrolling allowed the entire screen image to be offset by up to 7 pixels in either the horizontal or vertical direction. For scrolls larger than this, the software was responsible for shifting the display memory appropriately when the hardware scroll limit was reached. To achieve independently scrolling regions within the same screen, the programmer had to implement more complicated raster methods (see Section 5 for more details).

3.3.4 Colour

The 64 had a fixed palette of 16 colours. Border and background colours were specified using the appropriate VIC register. Foreground colours could be specified for individual character positions using the colour RAM (in both character based and bitmap modes). The VIC chip also supported a multi-colour version of each mode (and multi-colour sprites). In all cases, when multi-colour mode was selected, pairs of bits in display memory were used to specify the colour (background, multicolour1, multicolour2 and foreground). Whereas the foreground colour could vary from character position to character position, the remaining 3 colours were fixed for the entire display. A consequence of multi-colour mode was a halving of the display resolution and thus it was frequently dubbed *fat pixel mode*. Figure 8 illustrates both normal and multicolour sprites.

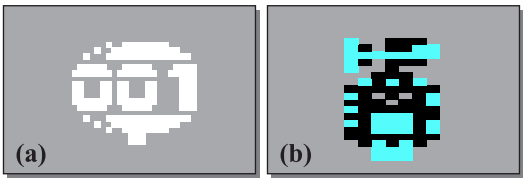

Figure 8: A sprite in a) normal mode and b) in multicolour mode.

4 3D Game Graphics

Its fair to say that around 1985 (when the games industry was in full swing) computer graphics used in games were quite primitive when compared to the state of the art in graphics research. At the time when the Hemicube method for Radiosity and distribution ray tracing were being developed the pinnacle of graphical achievement in the games scene was some clever visibility determination in the seminal *Knight Lore* (see Figure 9(h)) from Ultimate Play The Game (now called Rare). The technique, named *filmation* (which I've always associated with SuperMarionation, StingRay and Gerry Anderson!), was remarkable at the time though and represented the first real attempt at detailed 3D isometric graphics. The use of 3D in games dates right back to the earliest days: Atari's *Battlezone* (the original first-person perspective game) was a classic 3D tank simulation (rumour has it that the game was adopted by the U.S. D.O.D. for training prospective tank drivers) and the *Star Wars* coin-operated game featured the famous Deathstar tunnel strafing sequence. Figure 9 shows some "important" games (in terms of advancing the graphical standards employed in computer games).

4.1 Depth Priority

A large number of techniques were used to convey the impression of depth. The simplest involved the use of sprite-background or sprite-sprite priority to achieve a degree of hidden surface removal. *Nebulus* used this effect to achieve the appearance of rotation around a central tower, as can be seen in Figure 10. In such circumstances, usually the graphics were tailored to avoid any ambiguity (i.e. a sprite should never need to be both in front of one piece of the foreground and behind another).

4.2 Character Graphics Animation

This method was used to create some of the least CPU intensive 3D effects. It is analagous to colour lookup table animation; if a single 8×8 pixel character is repeated over an area, then a cheap scroll effect can be achieved by simply rotating (or shifting) the bits representing the character. All the character blocks displayed

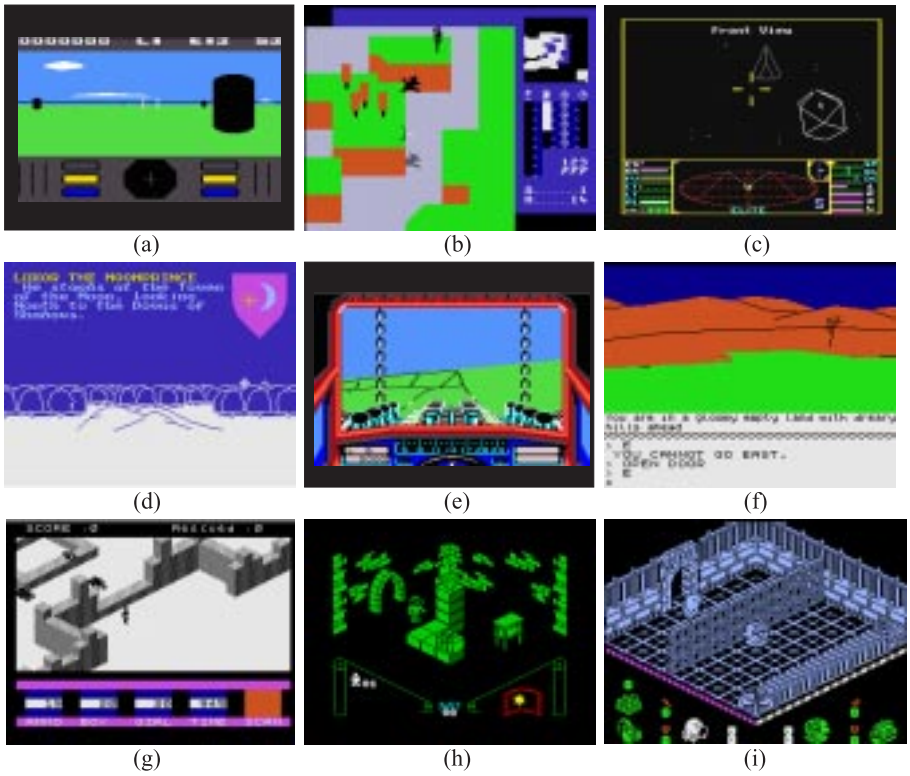

Figure 9: Definitive 8-bit computer games: a) *Encounter*, b) *Tornado Low Level*, c) *Elite*, d) *Lords of Midnight*, e) *Stunt Car Racer*, f) *The Hobbit*, g) *Ant Attack*, h) *Knight Lore* and i) *Head over Heels*.

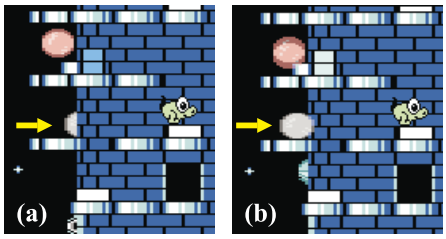

Figure 10: Using sprite background priority for 3D depth effects. In (a) the sprite has lower priority and appears behind the wall, whereas in (b) its priority has been raised and so it appears in front of the wall (thus it appears as if the ball has travelled around the corner of the tower). This priority switch was done through software.

with this character will scroll accordingly giving the impression of a moving area. The technique comes into its own when combined with hardware scrolling. For example, if the hardware scroll is shifting the display 4 pixels to the left each frame, and assuming we've draw "background" characters using the character we will animate, then animate the characters by rotating their bit patterns 3 bits to the right. The net effect is that the background characters appear to scroll left at a rate of 1 pixel per frame, whereas the rest of the image scrolls at 4 pixels per frame. We have created a parallax scroll (see Figure 11). This technique has been used in countless games, most notably *Parallax* (I wonder how they came up with that name), *Bounder* and *Uridium* (a final explosion sequence used the technique to simulate fire spreading in the opposite direction to a fast scroll).

4.3 Isometric Graphics

Isometric graphics (originally appearing in Sega's *Zaxxon* arcade game, are probably best represented by

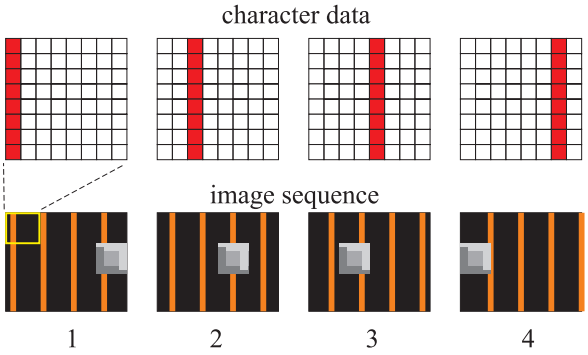

Figure 11: The parallax scroll effect involves shifting the data making up background characters by an amount not equal to the hardware scroll amount. In this example the display is shifted left by 8 pixels each frame and the character data is shifted right by 2 pixels, and thus appears to move 4 times more slowly than the foreground (unshifted) character data.

Figure 12: By drawing elements of the image in depth order, a 3D image was created with consistent visibility.

the Ultimate Play The Game's filmation games series which began with *Knightlore* in 1984. Since then there have been a large number of games employing the technique (notably *Spindizzy*, *Head over Heels*, *Batman*) which was achieved using depth ordered drawing. In almost all cases, the data to be drawn was aligned to a grid and viewed from fixed orientations (usually permitting rotation of the view through 90 degrees) thus simplifying the depth ordering. Figure 12 demonstrates the result of this technique. Arguably the technique was an old one. *Ant Attack*, released in 1983, employed the Soft Solid 3D technique to achieve believably 3D worlds (see Figure 9(g)). Using a similar method, the famous *Lords of Mignight* game composited flat bitplanes in a depth ordered manner to put together landscape vistas as you travelled through a vast world (see Figure 9(d)).

Figure 13: The visibility determination algorithm employed by *Elite* resolved local visibility only via back-face culling. Note the incorrect visibility indicated by the arrow.

4.4 Wireframe 3D

One of the earliest of the wireframe based games was David Braben and Ian Bell's *Elite*, originally released on the BBC Micro and which remained the best selling game for a long time. Its combination of space trading, vast playing area and atmosphere more than made up for the rather sluggish frame rates (which often dropped as low as 1 a second if a number of ships were being displayed simultaneously). *Elite* implemented back-face culling per object but no global visibility testing was performed (see Figure 13). One of the major innovations was the superlative 3D radar control which remains one of the most intuitive 3D navigation control I have come across and was patented by the authors.

Other noteworthy examples include *Mercenary* which defined the standard for Commodore 64 wireframe graphics with update speeds significantly faster than those of *Elite* and which allowed you to discover the joys of flying a piece of cheese! *Stunt Car Racer*, shown in Figure 9(e), showed what could be done with filled polygons, and though slow to update it managed to (ironically) convey a convincing sense of speed and momentum.

4.5 Bas-Relief

A special mention must go to Andrew Braybrook who possibly still is the most famous of Commodore 64 programmers. I myself was enthralled by the "Game Diaries" that he published in popular magazines of the time chronicalling the development of both *Paradroid* and *Morpheus*. Andrew was undoubtedly responsible for the hugh interest in the use of bas-relief for imparting a sense of 3D to a game (you get the same ef-

fect by passing an image with good contrast through an embossing filter), and it became a favourite method of mine when designing 64 graphics. See Figure 14 for some examples. Games making use of this technique included *Paradroid* (of course), *Uridium*, *Sanxion* and *Parallax*.

5 Raster Effects

Whereas the Atari has its ANTIC chip and the associated display list, to achieve similar results on the 64 or the Spectrum you were required to implement your own interrupt handlers called at key moments during a screen refresh. Many of the more esoteric effects possible with the VIC chip relied on precise manipulation of VIC registers during each refresh. However, raster interrupts were a necessity if you required smooth scrolling, flicker-free screen updates or split screen display modes.

5.1 Flicker Free Animation

Everyone knows that in order to eliminate flicker, you must synchronise the update of the display with the frame refresh (in particular avoid drawing into an area of the screen that is currently under the raster beam). Current video hardware usually implements this via a double buffer switch which is synchronised in this manner. On the 64 the normal method was to enable VIC *raster interrupts* and request an interrupt on a line just beyond the bottom of the visible display (during the *vertical blank*). The interrupt handler was then responsible for updating the display before the raster returns to refresh the next frame. Figure 15 shows the raster phases for a screen in normal mode.

5.2 Splitting the Screen

A trivial implementation of a split screen mode involved simply requesting a raster interrupt at the line we wish the split to occur at. The interrupt handler then simply switched modes as required and reinitialised the raster interrupt to occur sometime during the *vertical blank period* to allow the mode to be flipped back in time for the next raster refresh. This works quite well for static screens and horizontal scrolling, but when vertical scrolling is required within a split window and when sprites are allowed to cross the split boundary the timing of the split become more critical. We need to ex-

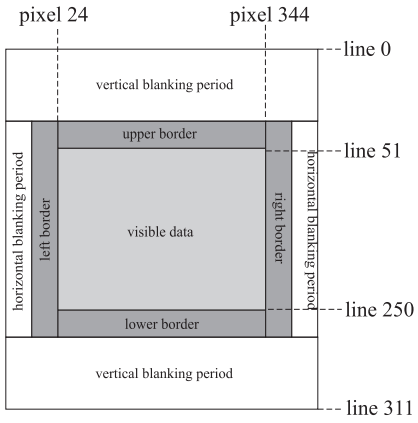

Figure 15: The geometry of the raster screen.

amine briefly the timing of a single raster line to understand the techniques required to achieve a rock-steady split. Wherein lies the problem? Recall that the VIC was capable of locking out the CPU when it requires the bus for graphics data accesses. If this happens at a split point the result could be a nasty flickering line around the split point representing the delay introduced as a result of the CPU halt.

Normally the CPU has access to the bus each positive phase of the clock cycle. There are 2 reason why this might be interrupted:

Sprite Data Access: if any of the 8 sprites lay across the current raster line, the VIC reads an extra 3 bytes from memory for the graphic data for each sprite. If all 8 sprites were active on the line, the CPU would be halted for a total of 24 clock phases.

Bad Lines: every 8 lines, the VIC loaded the character datarequired for the next 8 lines of display from RAM. This required an additional 40 cycles and thus the CPU would be locked out under these conditions also.

The position of the bad-lines is affected by the current vertical scroll position (which is independent of the sprite vertical positions), and thus it was a little tricky to keep track of these events. At worst the CPU will only have bus access rights for 6 phases of a line as shown in Figure 16. This is the source of the flicker. The processor didn't have time to flip modes before the raster beam hit the visible portion of the screen and so the split point would jump back and forth across the line as sprites crossed it and as the scroll position varied. To

Figure 14: The bas-relief effect was a great way to simulate raised surfaces: (a) *Herobotix*, (b) *Sanxion* and (c) *Parallax*.

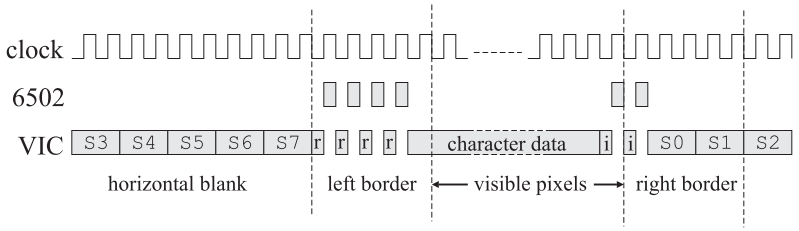

Figure 16: The timing of each line was fixed and the sequence of bus accesses was fully deterministic and depends on the line position, the scroll position, the number of sprites active on the line and the display mode currently active. The Sn blocks indicate a read of sprite n data and the character data is read only during *bad lines*.

account for this you had to take careful steps to ensure this is minimised by introducing variable length delays (usually achieved using `nop` (no operation) instructions and a liberal does of self modifying code).

5.3 Hordes of Sprites

To increase the number of sprites being displayed simultaneously you simply needed to change each sprite's vertical position once the raster had completely displayed it. The VIC did not keep track of the number of times a sprite was displayed; it simply examined the contents of the sprite y-position registers and at each raster line displayed those sprites that lay on the current line. Sprites could therefore be reused as many times as required with the proviso that a sprite could not occupy a single raster line more than once (this was known as *sprite multiplexing*). This created some difficulties in determining the optimum raster interrupt line after which sprites were be re-positioned. Consider the scenarios depicted in Figure 17 for example. If we used the simplest method and always positioned the interrupt request at the last line of the lowest sprite then we could potentially get sprite splitting artifacts where sprite data breaks up due to a sprite being repositioned before its display had completed. What was required was an optimisation step which minimised sprite splitting; there would not always be a solution (i.e. if the software required more than 8 sprites on a raster line, then something had to give), but through clever scheduling it was possible to minimise the problems. Some games suffered terribly from sprite break-up, the best example of which was *Commando*.

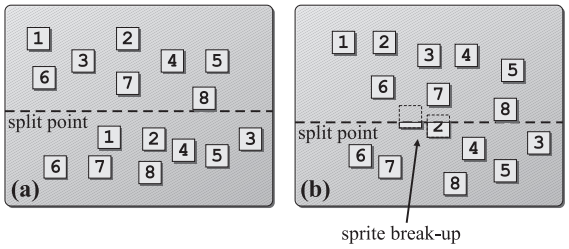

Figure 17: (a) demonstrates the successful application of sprite multiplexing. An interrupt is set to occur at the line corresponding to the last line of the 8th lowest sprite. At this point the sprites are re-positioned vertically to assume the positions of the next 8 sprites. In (b) however, this simple scheme fails, as sprites 1 and 2 have not been repositioned in time to display the lines above the split point.

5.4 Quirks

There were so many quirks to the VIC chip that programmers frequently exploited to introduce interesting effects to a game or simply to show that they could! I'll mention one: turning off the top and bottom borders. These borders are there for a reason; to hide data during vertical scrolling and to allow a buffer zone under which sprites could disappear gracefully. Getting rid of the border allowed you to display sprites in these normally hidden areas and thus you could achieve larger screen displays (sometimes known as *hyperscreen* though it wasn't exactly Omnimax). The enabling and disabling of the border was performed at pre-determinded scanlines and the current border state was recorded by the *border flip-flip*. The border was normally enabled at line 251 and disabled again at line 51. For smooth vertical scrolling it was necessary to shrink the vertical extents of the visible region (i.e. the border grew by 4 pixels at the top and bottom) to cover invalid lines. In such cases the border was enabled at line 247 and disabled again at 55.

To kill the border you initally selected a normal border (ON at 51 – OFF at 251), wait until the raster had reached line 248. Now you switched to an expanded border (ON at 55 – OFF at 247). In this mode the VIC only toggled the border state on line 247, but we were now on line 248, and so the VIC "forgot" to enable the border. After line 251 simply you reset the border to its normal state so as not to confuse the VIC when it came to disable the border on the next frame.

6 How Far Have We Come...

The Commodore 64's reign ended in the early nineties. This marked the end of the 8-bit computer (the 64 was probably the last of the popular 8-bit computers) and suddenly the 16-bit era was upon us with the Commodore Amiga and the Atari ST, and now in the late nineties these machines have been surpassed by the PC which currently holds the home computer crown. Games today are rarely ever the result of a single programmer and involve teams of programmers, graphics artists, musicians, directors, actors, script writers (and I'm sure there are probably grips and Foley artists and hairdressers). In the golden 8-bit era the programmers were the heros. Everyone waited for the next release from the afore mentioned Mr. Braybrook, or Jeff Minter, Tony Crowther, Paul Noakes, Goeff Crammond, David Braben, Steve Turner, John Phillips and so many others. Usually the programmer was also responsible for the graphics (though not always) but often somebody else would provide the music. The famous musicians of the time were Rob Hubbard, Martin Galway, Ben Dalglish and the Maniacs of Noise among others. The Commodore 64's SID chip (sound interface device) was an excellent 3-oscillator sound generator with resonant filtering that was pushed to the limits by these guys. With multiplexed chords, pattern based sequencing and sampled drum sounds some of the music created was quite amazing. The music for *Parallax* (about 20 minutes worth) by Galway and *Masters of Magic* by Hubbard were among the very best.

I suppose the attraction back then was the accessibility; you felt that you too could partake in the programmer's quest for the ultimate game, whereas today the industry has grown up and we go to College, get our degrees and then get a day job with a games company.

Current 3D technology (with Direct3D, OpenGL and the plethora of 3D acceleration cards) and 16 or 32 bit multi-channel sound and genetic algorithms for creature intelligence and CDROMs with gigabytes of level data have certainly changed the face of the computer game. Its now an interactive immersive environment with entities and goals and strategies. So have things really changed? What metric might we use to judge this; if I were to apply the metric of level of excitement generated, or the fear, or the sense of achievement at having completed a goal, then we haven't moved at all. I consider *Paradroid* and *Mission Impossible* to have been the best of the 8-bit crop. I would now consider *Quake II* to be the best of the current crop. I get equally as much enjoyment out of each. All I can really conclude is that the technology and the industry has grown up, but I haven't.

References

[1] *I.C. When* website,
`http://www.l4software.com/icwhen/`
[2] *Videotopia* website,
`http://www.videotopia.com/`
[3] *Planet Atari* website,
`http://www.geocities.com/SiliconValley/Vista/3015/atari.ht`
[4] Hubbard, Philip. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[5] *Planet Spectrum* website,
`http://www.nvg.unit.no/sinclair/planet/index.html`
[6] *CBM Document Page* website,
`http://www.hut.fi/Misc/cbm/docs/index.html`
[7] Bauer, Christian. The MOD 6567/6569 Video Controller (VIC-II) and its Applications in the Commodore 64, 1996. Available on the web at
`http://www.uni-mainz.de/ bauec002/FRMain.html`