CE301 Final Report

Nick Turner
nturne@essex.ac.uk
0605432

Dr Richard Bartle
rabartle@essex.ac.uk
Supervisor

Dr John Foster
jfoster@essex.ac.uk
Supervisor

N.Turner
April 2011

Abstract

This is a report on my efforts to create an arena-shooter style game for the Atari Jaguar. I begin with a discussion of the Jaguar, my motivation in selecting it, a brief history of the platform, and an overview of its technical merits and features.

I then discuss the game I have created, starting by explaining the goals I set for myself at the beginning of this project. I then go on to explain how much I achieved, followed by the things I failed to achieve, which is in turn followed by a discussion of some of the problems and errors I encountered along the way.

I then move on to discuss in more detail various aspects of the project, as well as detailed technical discussions about the capabilities and limitations of the platform. Particular attention is given to the graphical performance of the Jaguar and to the AI elements of my game.

In the conclusions I sum up my feelings about the project and my performance as a whole, as well as expanding on my future plans and the lesions learned from this project.

Finally in the appendices you can find additional information about various aspects of the project, a complete set of references, and a very comprehensive glossary of terms used in this report, which also serves as an index.

The full source code to each program created for this project can be found on the CD which accompanies this report.

Contents

Platform

The Atari Jaguar 64-Bit Interactive Multimedia System (Jaguar) is a fascinating and often overlooked piece of technology, and through the study of it, one can gain an insight into a fascinating part of video-game history. It's brief commercial lifetime spans a period of great change in the industry, and its technical design and commercial failure illustrate the rapidly changing paradigms and technologies of its time.

Why Jaguar?

One of the things I was asked at the recent "Project Open Day" was why I decided to write this game in assembly on the Jaguar, as opposed to say on a PC running Java. I suppose my main reason was a simple love of the platform. The Jaguar was the console I really wanted as a child and couldn't have until many years later, and it has always been a favourite of mine since then.

I taught myself assembly using it, and while many people are critical of its hugely flawed hardware and poor set of development tools, I take pride in overcoming these challenges. I have for a long time dreamed about writing a game for it[1], but lacked the time required to do so. So naturally when I realised that I could spend a year doing something I loved and getting credit for it towards my degree, I jumped at the chance.

History

The Jaguar's history is short and largely overlooked. Launched at Consumer Electronics Show (CES) in 1993[26], the systems commercial life ended early in 1996[21] when Atari merged with JT Storage Corporation (JTS) and effectively ceased trading.

The story of the Jaguar was not to end there however, and it continues to have a small but loyal following, with many new releases and a strong presence at retro gaming shows.

Pre-Release (1989-93)

The Jaguar was developed for Atari by Flare II (a trading name of Flare Technology) on behalf of Atari and was developed as a successor to the Panther[8] project Atari had been developing in-house. Jaguar took some of Panther's features — such as the Object Processor (OP) — and extended them, while also adding the Graphics Processing Unit (GPU) and blitter as a tightly coupled pair designed to do 3D rendering[23].

At the time it was believed that gouraud shading would be sufficient[23] — no other home console had produced polygonal 3D games, and ray-traced rendering on computers was in its infancy still, having first been developed in the early 80s[36]. This sadly proved not to be the case, and the consoles which followed just a few years behind the Jaguar excelled at texture mapping.

Atari had a handful developers working with them during the hardware design stages, most notably Jeff Minter[13] and Attention To Detail (ATD)[31]. Jeff Minter has always been a fan of Atari and its machines, and "Ever since the days of the Atari 800 he has been churning out psychedelic games full of llamas, sheep, and toilets."[10] His work on the Jaguar was to prove similarly psychedelic and produce some of the best titles the system has to offer.

Sections of the gaming press, fueled by a number of forward looking press releases and interviews from Atari, as well as the various "launch" events[26, 25, 27], managed to successfully create a significant amount of demand for the system in

[1] Although the design for this specific game was something new, thought of shortly before submitting the initial proposal.

advance of its launch[31]. Atari would later fail to capitalise on this by having an insufficient number of consoles available to meet the demand.

Atari were under considerable financial pressure, and although a number of fairly serious bugs were discovered[3], they were not rectified before the systems launch.

Commercial Life (1993-96)

Released with a spectacular launch party[27] in November of 1993, the Jaguar attracted significant interest from developers wanting to produce games for it. Problems with manufacture by IBM, whom Atari had contracted to assemble the Jaguar, and Toshiba, who were making the custom chips, led to poor initial sales[28], with demand greatly outstripping supply for many months[29]. Atari, who were traditionally very bad at promotion of their systems, were caught out by how much hype and demand the press managed to create for the Jaguar and were ill-prepared to capitalise upon it[28].

Atari's supply problems were not just limited to the consumer market, and they were very bad at delivering development kits promptly to studios that signed up[22, 31, 30], leading to many games being announced to the press before any work had even begun. Given how difficult developing for the Jaguar can be[3], this led to the majority of announced games never being completed. It also took a long time for Atari to release a PC based development kit, initially supplying a kit based around their own TT line of computers, which were unfamiliar to most developers and incompatible with PC tools. In fact the first PC based development kit was released by a third party, Cross Products[30].

By the time Atari started to get the Jaguar produced in large quantities, questions were already being raised about the quality of games for the system, with many of the early releases failing to live up to their promise, and in some cases being described as "lacking any originality or depth"[31]. The lack of quality titles would continue to be a problem for the Jaguar throughout its lifetime[9].

In late 1994 Atari settled a long running court case with Sega with a deal that saw Sega invest $90 million into Atari[33]. The deal also opened the way to the companies cross-licensing games for each others systems, although this would not actually happen during the Jaguar's lifetime.

One of the major contributing factors to the failure of the Jaguar and Atari was the timing of the release of the system, combined with the ever present rumours of more powerful systems yet to come. The Jaguar was technically far more powerful than the 16-bit systems that dominated the market when it was launched, but lacked captivating games initially, and was not as powerful as the systems that were to follow shortly behind it.

As Sam Tramiel once said when promoting the Jaguar "What is the most fun the consumer can have with with the system? That's what the consumer cares about."[15] The Jaguar was the first of what would come to be known as the 5th generation of home consoles, but launched two or three years before the consoles that would come to dominate that era. As he accurately put it "I'm convinced the consumer is out there saying, "I'm confused."

The public were confused and torn between holding onto their old systems, which were still receiving a steady stream of good games, buying the Jaguar which was in stores, but had very few good games, and waiting for the Saturn, Playstation, or N64, all of which received a huge build-up in the gaming press long before their release[9, 11]. The N64 started being talked about as much as three years before it's eventual release.

Atari, having failed to capture an early lead in the market, continued to push ahead with new hardware such as the

Jaguar CD addon, and Pro Controller[19], as well as developing a virtual reality headset that sadly was never to materialise[2]. The quality of games releases did improve, with titles such as Doom and Alien vs. Predator coming out in 1995 and adding to the already released Tempest 2000 (T2k), but it proved to be too little, too late.

Post-Atari (1996-present)

In February 1996 Atari performed a merger with JTS[21] and effectively ceased trading. In 1998 JTS sold the remaining intellectual property it had acquired from Atari to Hasbro Interactive (Hasbro). In 1999 Hasbro released the rights to develop for the Jaguar[16], effectively making it an open platform from a legal standpoint[3]. In 2001 Infogrames bought Hasbro Interactive and acquired the Atari name and intellectual property[12], but has not shown any interest in the Jaguar.

There has existed a small, but lively and active community on the internet surrounding the Jaguar ever since its release. As happens with many small online communities, the Jaguar's scene is quite factionalised. However there are some talented people releasing new games and hardware for the system, which are generally well received by the majority of Jaguar users.

New hardware releases have included the "Underground" Boot-ROM replacement Behind Jaggy Lines (BJL), which allows code to be uploaded into memory via a cable connected to a PC, several modified controllers — most notably Rotary Controllers for T2k and Rapid Fire Controllers containing a small circuit to automate the action of the fire buttons — networking devices such as the JagLink2, and various types of flash-programable cartridges which allow for cartridge emulation, and in some cases remote debugging.

The games released post-Atari can broadly be divided into two categories: commercial games released on cartridge or CD and sold for a profit, and homebrew games released for free, typically as an upload-friendly file or CD image, sometimes with a "collectors edition" sold in limited quantities on CD.

Technical

Technically, the Jaguar was a revolutionary improvement over the 8 and 16-bit systems which had come before it, but was always considered difficult to program for.

Bus and Memory

The Jaguar features a 64-bit wide bus, which connects the main RAM via the memory controller to all of the processors, cartridge port and boot ROM. The main system memory is 64 bits wide and consists of a single 2 megabyte bank starting at address 0. All peripherals and internal registers are memory-mapped and occupy address ranges at the high end of memory[5].

The Jaguar has five main processors, and was designed around the concept of saturating memory throughput[8]; therefore all five processors are all bus masters, but there is a fixed hierarchy of priority levels[4]. Processors that use the bus only sparingly, or have a need for low latency access have higher priorities, and are able to steal the bus away from lower priority processors that use the bus more intensively — with the result being that the bus is in active use for a high percentage of the available time.

[2] I have been lucky enough to use one of the two prototype units known to exist, and it is very impressive. True VR has never been achieved on a home console, and if Atari had managed it with Jaguar VR, I believe history may have been rather different.

[3] Although the rights were released in 1999, the encryption keys were not discovered until 2003[35].

Processors

The five processors that make up the Jaguar are housed on two custom fabricated ICs — named Tom (containing the GPU, OP and blitter) and Jerry (housing the Digital Signal Processor (DSP)) — while the Central Processing Unit (CPU), a Motorola 68000, is housed in a separate IC.

The CPU is the lowest priority processor on the bus, and also the slowest part of the system, due to its Complex Instruction Set Computer (CISC) architecture, its 16-bit data bus, its lack of local cache and being clocked at half the rate of the other processors. Despite these limitations, many games make heavy use of the CPU, sometimes to the detriment of their overall performance[23]. Because the 68000 was also used in the Sega Megadrive, Atari ST, and Amiga ranges, a number of Jaguar games were ports from these platforms with only minor changes to account for the differing hardware.

The OP handles the task of preparing each line of the display. It takes its instructions in the form of an Object Processor List (OPL), a linked-list of 'object' definitions. Each object is a tightly packed data structure of between one and three 64-bit (Phrase)-lengths. Objects can instruct the OP to draw a bitmap image, or a scaled bitmap image, with a number of options and bit-depths, a stop object to halt processing for that line, an interrupt object to pause processing and request the attention of the GPU to perform some task, or a branch object which will fork the list depending on some variable (typically the current line number). The OP is a 64-bit chip and has the highest priority on the bus[4], it reads both instructions and data as whole phrases, and writes into the line buffer at a maximum rate of 2 pixels per clock cycle[4].

The core processors of the Jaguar are a pair of Reduced Instruction Set Computer (RISC) processors, the GPU and DSP. They share many of the same instructions, but each have a few specialised instructions to assist with their respective tasks of graphics and sound generation[4]. Each chip is 32-bit internally, and contains 64 32-bit registers, a powerful Arithmetic Logic Unit (ALU) with fast multiplication and barrel shifters, and a systolic matrix multiply unit, making them very capable general purpose processors. Through the use of a highly pipelined design, they can achieve an instruction throughput of one instruction per tick, with each instruction taking between three and four ticks to complete on average.

The GPU has 4 kilobytes of local RAM, and the DSP has 8 kilobytes; however, due to a bug in their design, they are unable to execute programs out of main system memory, and are thus constrained to running from their local RAM[3]. This means that large programs need to efficiently handle the paging in and out of code, usually with the help of the blitter. The GPU is able to make phrase-width transfers to and from main memory, but conversely the DSP only has a 16-bit data bus and must make 32-bit transfers in pairs. The DSP has direct access to the hardware timers and audio circuitry located on Jerry, as well as 4 kilobytes of wavetable ROM, synchronous and asynchronous serial communication, and the control pad interface. These features, combined with it's extra local RAM mean that programs running on the DSP should require less access to main memory and thus require the bus less often.

Finally, the blitter is a specialised processor designed to quickly copy blocks of memory. It features two very flexible address generators, for source and destination addresses, and can perform a range of complex tasks at a rate limited only by memory access times. It can perform simple block transfers, fills and copies of rectangles within larger images, line drawing, image rotation and scaling, character painting, patterned fills, single scans of polygon fills, gouraud shading and z-buffering[4]. It is designed to be used in close

[4]Except for the DRAM Refresh mechanism.

co-operation with the GPU, with the GPU calculating what needs to be drawn, and the blitter doing the low level drawing and copying. As discovered by Jeff Minter, interesting effects such as the pixel-shatter used to great effect in T2k can be produced by swapping the roles of the address generators, something he was once chastised for by one of the hardware designers[13].

Colour Models

The OP can handle scaled or unscaled images at 1, 2, 4, 8, 16, and 24 bits per pixel. Images of 1–8 bits per pixel are converted to 16-bit through the use of a flexible palette.

The Jaguar has two different colour models it can make use of. Standard Red, Green, Blue (RGB) at 16 or 24 bits per pixel, and the Jaguar specific 16-bit Cyan, Red & Intensity (CRY) colour scheme.

CRY makes use of two 4-bit colour vectors that index into a 16×16 grid of colours, modified by an 8-bit intensity vector that determines the brightness of the colours. At an intensity value of zero, all colours are black, whereas at intensity 255 colours are vibrant and intense. Unlike the more common Hue, Saturation, Luminance (HSL) model — which places white at one end of the luminance range, black at the other, and intense colours at the mid-point — white can be found in the CRY model at the middle of the two colour vectors when the intensity vector is at maximum.

It is also interesting to note that there are four separate values for white, all of them ever so slightly off from "pure" white as we typically think of it.

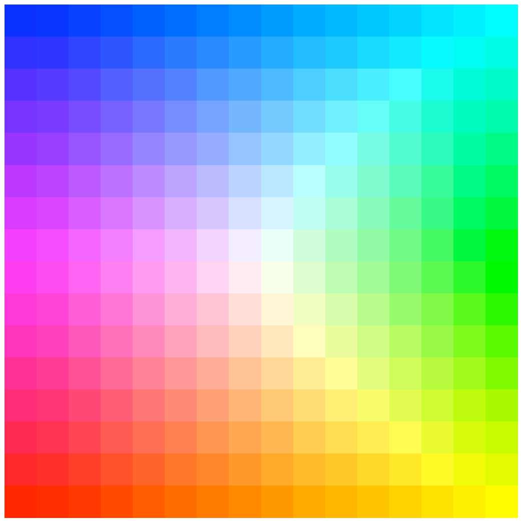


Figure 1: The CRY colour-space at maximum intensity.

CRY mode graphics also allow for hardware assisted partial transparency through the OP and Read, Modify, Write (RMW) objects, which treat the three channels as signed offsets to be added or subtracted from the value already in the line-buffer for that pixel. The values then saturate at their minimum and maximum values, and can be used for coloured lighting or other transparency effects, but require much more care to use than the more common (in modern times) alpha-channel based transparency.

While the CRY model is quite capable, it is quirky, and not many games used it to its full potential during the lifetime of the Jaguar.

More details about how the colour model is constructed, and conversion tables, can be found in the Jaguar Manual[4].

Controller

The Jaguar shipped with a distinctive, but generally unpopular control pad. It consists of a digital Directional Pad (D-Pad), 3 main buttons (A, B, and C), 'pause' and 'option' buttons, and a 12-key numeric keypad (1–9, *, 0, #). The numeric keypad had two small slots on each side, into which a plastic overlay could be placed. These overlays, supplied with games that made extensive use of the D-Pad, allowed for the buttons to be given graphical or textual labels, and was, in effect, a simple kind of software-specific re-configurable interface.

The use of overlays was very similar to the Atari 5200 console, an 8-bit system that sold very poorly. While the 5200 controller was arguably more advanced than the Jaguar's, featuring an analog joystick, it was notoriously unreliable[11].

Atari released a second control pad, the "Pro Controller" in October 1995[19], featuring 3 additional main buttons (X, Y and Z) and two shoulder triggers (L and R), as well as a more comfortable D-Pad and buttons. The extra buttons map directly to the keys on the numeric keypad, and although there were several games that were developed with the new layout in mind, their function in most games is somewhat arbitrary.

Jaguar CD

Released in September of 1995, the Jaguar CD addon sits atop the base console and provides a dual speed CD-ROM drive as well as a pass-through cartridge slot. From a technical point of view the data is encoded strangely, being written as audio tracks on the second (and subsequent) sessions of a multi-session disk. This was intended to prevent people accidentally playing the data tracks when inserting the disk into audio equipment, since red book audio CD's are only permitted one session.

The Jaguar CD unit has a custom interface chip known as Butch, and can stream audio data to the DSP without using the main system bus.

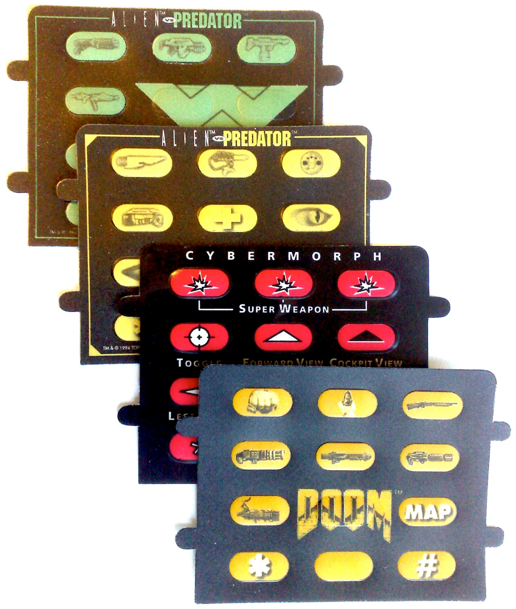


Figure 2: Jaguar controller overlays.

Objectives

I set out with a fairly clear vision of the game I wanted to produce, and of the things I wanted to achieve in terms of creating something new that had not previously been seen on the Jaguar, and that pushed it technically in new directions.

Technical Advances

I wanted to create a game which stood out in three specific areas.

Firstly, I wanted to create a game that used the full vertical resolution the Jaguar is capable of. Apart from a simple proof of concept demonstration[38], all Jaguar games run at half the Standard Definition Television (SDTV) resolution or less, with some drawing as few as 200 lines of video.

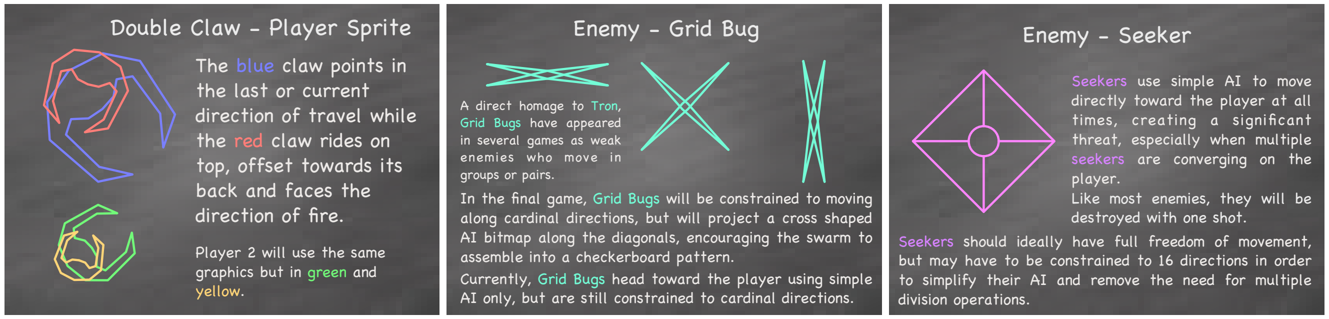
Secondly, I wanted to create a game that was designed to use a non-standard controller. There have been only two games — T2k and Total Carnage (TC) — that can take advantage of special controllers, and neither has had their controllers officially released, although a number of fans, myself included, did release several ranges of Rotary Controllers for T2k. I wanted to write a game that supported the same Dual-Stick Controller that TC is designed for.

Thirdly, I wanted to make a more extensive use of CRY RMW images than had previously been used. While it is hard to be certain of their use in other games, they seem to be relatively uncommon. I wanted to create a game that would make extensive use of multiple RMW layers to create complex backgrounds.

Gameplay Elements

I set out to create a fast paced, "Arcade Action" style of game, similar to Robotron 2084 and Geometry Wars, which would both pay homage to the video games of the 1980s and early 90s, while providing updated graphics that earlier systems would have been incapable of, and that would not look out of place on modern consoles.

Virtual Reality and Cyberpunk were strong themes in a number of Jaguar games (T2k, I-War, Black Ice White Noise, Syndicate, etc.) and I set out to continue that tradition thematically, with clean lines and bright coloured abstract shapes, while incorporating tried-and-tested fast paced action[37].



Double Claw - Player Sprite

The blue claw points in the last or current direction of travel while the red claw rides on top, offset towards its back and faces the direction of fire.

Player 2 will use the same graphics but in green and yellow.

Enemy - Grid Bug

A direct homage to Tron, Grid Bugs have appeared in several games as weak enemies who move in groups or pairs.

In the final game, Grid Bugs will be constrained to moving along cardinal directions, but will project a cross shaped AI bitmap along the diagonals, encouraging the swarm to assemble into a checkerboard pattern.

Currently, Grid Bugs head toward the player using simple AI only, but are still constrained to cardinal directions.

Enemy - Seeker

Seekers use simple AI to move directly toward the player at all times, creating a significant threat, especially when multiple seekers are converging on the player.

Like most enemies, they will be destroyed with one shot.

Seekers should ideally have full freedom of movement, but may have to be constrained to 16 directions in order to simplify their AI and remove the need for multiple division operations.

Figure 3: Player and enemy designs.

Achievements

My goals for this project were ambitious, and while my initial estimation of time was more than a little optimistic, I feel I achieved a significant amount, and most of the tasks I set out to.

High Resolution Video

This objective was achieved at the start of the project, and maintained throughout the entire project. By writing the entire game to support high-resolution output, all of my OPL generating code was made more complicated, which as a result took longer to debug and slowed the entire project slightly.

However it is a sacrifice worth making and I would not have wished to do otherwise. When this game is released it will be unique in the quality and resolution of its graphics[5]. I also discovered a number of interlaced video related bugs and their consequences over the course of this project, which I was able to share with the developer community.

In terms of temporal resolution, I have also managed to maintain the maximum possible frame-rates of 60 Frames Per Second (FPS) on NTSC and 50 FPS for PAL[6]. One interesting aspect of a game like this, is that different elements of the display can have their own frame-rates, with, for example, the enemy sprites and bullets updating at 60 FPS while the background layers are only redrawn every other frame for 30 FPS.

Controller Input

The controller reading and abstraction code proved very easy to write and test. Since the game was designed from the beginning to be able to make use of several different control-schemes, the input routine abstracts the data into a pair of values (one for each player) that can be referenced from elsewhere in the game without worrying about the specifics of the controller type in use. Currently the game supports three different control methods but I plan to add one or two more before it is released. It supports both the ordinary Jaguar controller, the Atari Pro controller, and a custom Dual-Stick Controller, which is in turn fully compatible with TC.

Regarding CPU Choice

So far all of my code has been for the CPU, with the OP doing all of the graphics work and the GPU, DSP and blitter sitting idle.

This arrangement was chosen for two reasons. Firstly writing code for the CPU is much easier due to the flexibility of its CISC architecture, and secondly because in order to properly pipeline RISC code, it should be planned out and tested first, then re-arranged to minimise pipeline stalls. In effect, the code I have written for the CPU can be considered pseudo-code for the GPU and will make the translation to it far simpler than trying to write directly in RISC assembly.

Player Sprite Display

I spent considerable time on the design of the player graphics, eventually settling on a "Double Claw" design that is both abstract and immediately familiar and intuitive. The sprite is composed at runtime from two images, one for each claw. The larger, lower claw is used to show the player's direction

[5] As of April 19th 2011, a new game has been announced that will also use high-resolution video. I hope to be able to complete and release mine first.

[6] Technically that is Fields Per Second, since I am outputting interlaced video. In the interest of readability, except when discussing specific issues relating to interlacing, I will not make the distinction.

of travel, while the upper claw shows the direction of fire for the player's weapon.

Their palettes are specifically chosen to merge in specific bit-patterns when combined together with a bitwise AND operation. There are six colours in the final image: edge and fill colours for each claw, the transparent background, and a "shadow" colour that gives a distinct edge between the lower and upper claws where they overlap, but does not appear between the claws and the transparent background.

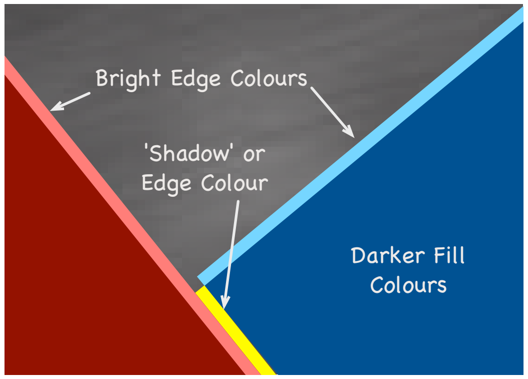


Figure 4: Mechanics of the player sprite compositing process.

Enemy Display

This took significantly longer than planned for, due in part to bugs in the OP, bugs in my code, the difficulty of debugging OPL generation, and the amount of time spent planning my code and data structures.

The method employed generates an OP object for each enemy in the mob-list, linking them together as it goes. This produces a list within a list which can be attached to the end of the general purpose list for other objects. While this technique is memory and bus efficient, it is moderately CPU intensive, and produces a non-constant workload for the OP, making performance hard to measure. For more discussion on this see Drawing Method Performance on page 18.

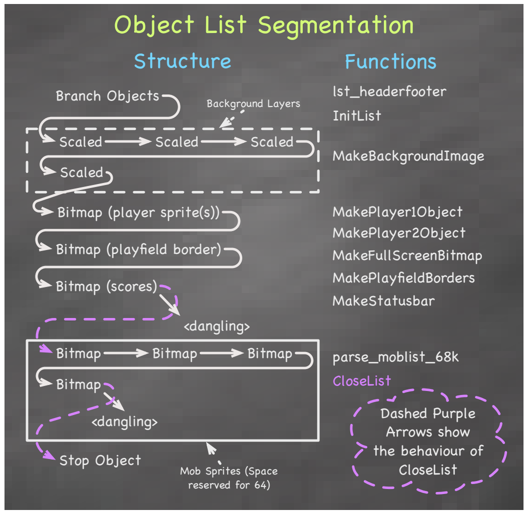


Figure 5: Updated diagram showing the structure of the OPL.

Enemy Update Mechanism

This relatively simple routine was easy to write in comparison to the OPL generating code, but has undergone very many revisions and changes since it was first created. It currently checks each object against the four bounds of the play-field, decrements the Artificial Intelligence (AI) timer, calls the AI framework when required, and updates enemy positions based on their speed and direction.

It exists as a separate call from the enemy display routine for two significant reasons. Firstly the enemy display routine must be called every frame, and before I decided to double-buffer the OPL, it had to complete in the brief period between the Vertical Blanking Interval (VBL) interrupt and the start of the first line of video. Secondly, by separating the routines, the updating of the enemies can be slowed to half rate (see page 17) or suspended entirely (for example, when the game is paused).

AI Framework

The AI framework provides a single entry point to the AI routines. It is basically a "trampoline" routine that does a simple look-up of which class of enemy is being processed, followed by a jump into a jump table[18], from which it bounces off into the appropriate AI routine for that enemy type. This technique can be used for all the AI types the game will support, and allows for a thin layer of abstraction between enemy type and AI behaviour.

Tightly coupled with this is the concept of an AI timer, which is just a counter that gets decremented each time the enemy is updated. When the counter reaches zero, the AI entry routine is called for that enemy. As part of the AI routine, the counter is reset to a sensible value plus a random element. This technique helps to ensure two things: firstly, that the system does not get slowed down by processing AI for every enemy every frame, and secondly, that the delays between updates produce a more natural-looking movement in the enemies without the need for complex movement interpolation.

There is a lot more information about my AI, both implemented and theoretical, on page 19.

Additional Features

Also implemented is a Pseudo-Random Number Generator (PRNG) based on a design for a 16-bit Linear Feedback Shift Register (LFSR) by David Hotz[17]. It provides a satisfactory level of randomisation for my needs, and is very fast in operation.

Code also exists to generate scaled bitmaps suitable for the background display, in both normal and RMW modes, and a simple routine for generating the enemies for the test level.

Dual Stick Controller

The game supports a type of controller that has, to my knowledge, never been produced by anyone for the Jaguar before. I plan on making a number of these to sell when the game is released, and have successfully produced one to prove the concept is sound.

The Dual Stick controller is built from entirely new components, with a circuit board of my own design. The circuit is a relatively simple affair and does the same job as the Atari design, but uses a different IC with a more sensible pin arrangement, allowing for the entire circuit to be produced on stripboard with a clean layout and few jumper wires[7].

The hardware consists of a pair of Seimitsu LS-33 joysticks, one large pushbutton and two smaller pushbuttons. The left hand joystick is connected to up, down, left and right, and exactly duplicates the Jaguar's D-Pad, while the right hand stick is connected to 2, 4, 6 and 8 on the numeric keypad. The single large pushbutton is mapped to 'C' and the two smaller ones are for 'Pause' and 'Option'.

This arrangement covers all of the inputs used in both TC and for my game[14]. The decision to only have one main button was taken since neither game makes use of 'A', and the functionality of 'B' is, in both cases, an alternative to having the second joystick, and not required when one is present.

[7]For the full technical details of the circuit, and the schematic, please refer to Appendix I

Software Tools

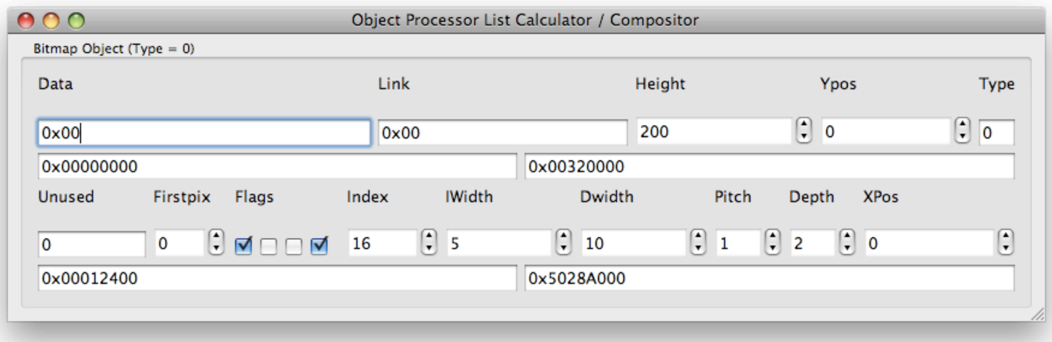
During the course of writing this game, I created a number of utilities to fill gaps in the Jaguar's toolchain. Once I am satisfied that they are mature and useful enough, I will release them to the community under some kind of open license.

OPLCalc

Since the majority of this game so far involved the creation of OPL objects, and the most difficult debugging tasks involved decoding those objects and understanding what was going on in them, I found it very helpful to have a simple converter / calculator I could use for that purpose.

The OP reads objects that are between one and three phrase lengths long, consisting of multiple fields of varying bit-lengths packed tightly together. Because of the extremely tight-packed nature of OP objects, they can be very awkward to read at a glance, or to generate pre-computed fields for compositing in code.

OPLCalc is written in C++ with Qt and is portable to any platform supported by Qt. In its current form it only supports Bitmap objects, but will be extended to support all of the Jaguar's objects before release.



Data	Link	Height	Ypos	Type
0x00	0x00	200	0	0

0x00000000 | 0x00320000

Unused	Firstpix	Flags	Index	IWidth	Dwidth	Pitch	Depth	XPos
0	0	✓ ☐ ☐ ✓	16	5	10	1	2	0

0x00012400 | 0x5028A000

Figure 6: Screenshot of OPLCalc utility.

PBM2CRY Suite

This is a number of small command line utilities with a shared codebase written in ANSI C for portability. They are built to automate the conversion of graphics in "classic" Portable Bit-Map (PBM), Portable Grey-Map (PGM) and Portable Pix-Map (PPM) formats into raw binary data suitable for inclusion into a Jaguar program.

PBM, PGM and PPM are very simple formats that use raw ASCII numerals to represent pixel values, with a simple header and support for comments. They are widely supported by graphics programs, and have the unique advantage of being human readable and manipulatable. Since everything is printable ASCII, graphics in these formats can be created and edited in any text editor.

Atari did supply a graphics conversion tool designed to read targra format images and convert them, however it offers relatively few options for processing the images during conversion and poor control over the colour mapping of low bit depth images. I hope that my suite of tools will provide an alternative to be used along side of Atari's tool, rather than to replace it entirely.

While the current suite of tools consist of image creation (blank images of a given size) and conversion only, I intend to add more complex image processing functions to them as time permits, and possibly a graphical front-end for interactive work. I have previously attempted to create a graphics program targeting the Jaguar and its CRY colour model, but abandoned the projects due to the complexity of the task. By using the PBM2CRY suite of tools to do all the main processing tasks, I get the benefit of modular code the is easy to test, and that can be integrated into users' build scripts easily, providing a range of filters and processing tools that can be called automatically at compile time.

Shortfall

Despite my best efforts, I did not manage to get as far along as I had hoped. I did not manage to complete several fairly significant elements of the basic game, and managed very little towards the advanced features of the game that I had hoped to get to.

Collision Detection

The next significant thing to implement should have been collision detection between the player and enemies. This would be part of the enemy update routines and should not, I believe, actually take very much work. The update routines already perform collision detection between each enemy and the edges of the play-field, so with their positions already in memory it would take very little extra time to also check them against the players' bounding box(es).

Pixel-accurate collision detection will be easy to implement if I decide to switch to using a multiple-framebuffer approach (as described on page 19). The blitter has hardware assistance for collision detection. After drawing all of the enemy sprites, the blitter can be set to draw the players' sprite(s) onto the same image, with the data comparator set to interrupt processing if any part of the player sprite is drawn over the top of a non-transparent pixel.

Pixel-accurate collision detection would be difficult to achieve under the current method of drawing all enemies with the OP.

Shooting

Another significant feature absent from the game as it stands is the ability to shoot. This covers a number of related functions. Projectile creation, projectile movement, projectile display and collision detection between projectiles and enemies. Each of which is a significant task in it's own right.

I do not believe I ever gave these issues as much attention as I should have, and as a result I am still somewhat unsure as to how they could have been implemented under the current design of the game engine, given how much I now know about the amount of time available to the system each frame (see Drawing Method Performance on page 18).

Bitmap AI

Although it was always designated an optional component, to be completed if I had time, I had strongly hoped to be able to implement the more advanced AI methods I had planned.

Controlling State Machine

I had also hoped to implement a Finite State Machine (FSM) to keep track of the global state of the game, however this was a low priority compared to the other basic features. It would keep track of all the basic states of the game, including temporary states such as animated transitions between menu items and at the start and end of each level.

Problems

As previously mentioned, I did not achieve quite as much as I had hoped for. Part of this was due to under-estimating how long things would take, but a significant part of my time was spent trying to diagnose and fix bugs, either in my code, or as a result of the Jaguar's numerous hardware bugs.

OP Interlace Bugs

The OP was supposed to be capable of handling interlaced video and adapting its behaviour to accommodate the changes it requires[4]. However, this is sadly not the case, and does not appear to be an issue Atari were aware of[3].

Height Decrement Bug

The OP was supposed to decrement the height field of each object by 1 per line in non-interlaced mode, or by 2 per line when drawing interlaced video[4], to account for the extra line that will be drawn in the next field.

In fact, it decrements by 1 regardless of the video mode. This has the result that objects get drawn at double their full height, with each line of bitmap data appearing on both fields of the display.

In order to work round this bug, the easiest way is to use the clipping behaviour of the OP and to define the object as being twice as wide, and half as tall as it actually is, and then tell the OP to clip the image to its actual width, as illustrated in Figure 7. In order to draw the odd-numbered rows of the object on the odd field of the display, the (actual) image width must be added to the data address when building the list for the alternate field.

It is possible to use this bug as a feature when creating elements such as thin horizontal lines without flicker, or scaled objects to effectively get a $\times 2$ vertical scale factor for free.

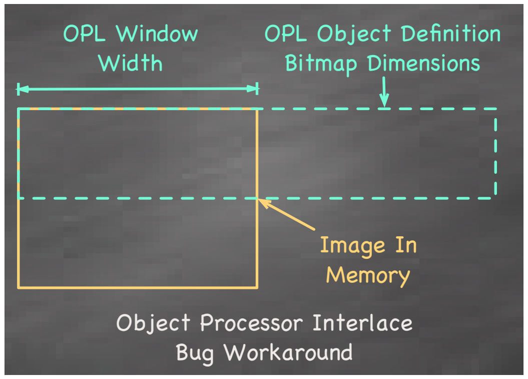

OPL Window Width
OPL Object Definition Bitmap Dimensions
Image In Memory
Object Processor Interlace Bug Workaround

Figure 7: OP Interlace Bug Workaround.

Y Precision Bug

Another artefact of the OP not adequately understanding interlaced video is that because the Vertical Count (VC) register always updates in intervals of two, objects were only being drawn on even scanlines. The correct behaviour of VC would be to reflect the actual scanline being drawn, which would increase by two at all times, but would be an even value when drawing the first – or upper – field of the display, and odd when drawing the second or lower field. This is sadly not the case and the VC register remains even at all times.

In order to work around this, and align objects correctly on the odd fields, it is necessary to perform some moderately complex checks. If the object is on an even alignment, and the display is drawing the odd field, the image width must be added to the image data address, but if the object is on an odd alignment, and the display is drawing the even field, the image width must be added to the data address, and the y-position of the object must be increased by one.

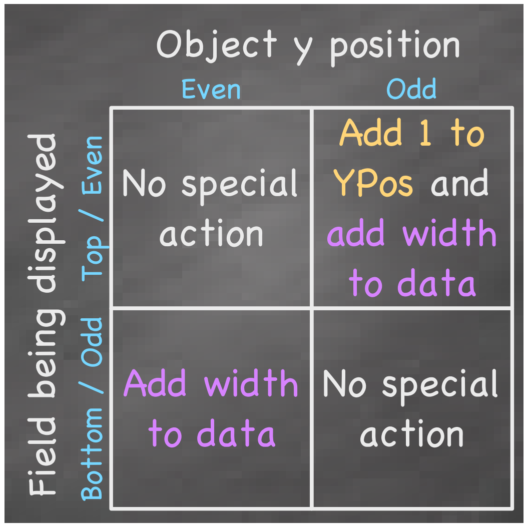


Figure 8: Object Alignment Checks.

Crashes and Lockups

One of the most annoying aspects of the Jaguar is the lack of proper diagnostic output from the custom processors. The OP will typically just shut down completely when presented with input it does not like, and frustratingly, this quite often will also cause the entire system to crash in such a way that the remote debugging link provided by the official development kit will cease functioning entirely. The only way to restore the unit to a working state is to reset the system, but this effectively removes all trace of what caused the error.

This one issue alone is responsible for consuming a vast amount of my time. It required me to initiate elaborate work-rounds to work out what was going wrong, such as changing the value of the background colour register at the start and end of each function, in order to see which one failed to terminate properly — a technique that works in most cases only because the background colour register is read by the graphics hardware directly, and used to fill the line buffers before the OP starts writing to them.

Performance Issues

I ran into a couple of performance issues during development, where I was attempting to get more out of the Jaguar than it was capable of.

By increasing the number of enemies drawn, I ran into problems, which, it turned out were not the limits of the OP's performance, but of the CPU's. The problem arose because in the time between the VBL Interrupt firing, and the OP starting to draw the first line of video, there was not time for the CPU to finish creating the entire OPL. I was able to fix this by double-buffering the OPL, although porting the code to run on the GPU (as I originally intended) would probably have fixed the problem too, or at least significantly increased the length the OPL could reach before it became an issue[2].

The second timing problem I encountered, however, was due to the OP itself running out of time. Under ideal conditions, the OP can write two pixels (at 1–16 bits per pixel) per clock tick. Scaled images, as well as 24-bit images, are written at one pixel per tick, and RMW objects are written at half that rate[4]. When, towards the end of time I had available for programming, I attempted to create several scaled RMW objects (of 60×80 pixels scaled up to 640×480), and experienced distorted video and tearing away of some sprites as the OP ran out of time to draw them.

This forced me to conclude that the third of my three technical objectives (the use of multiple RMW objects to create a multi-layered background) is technically impossible. I still plan to use the effect, and graphically the backgrounds will look the same, but they will have to be pre-combined by the blitter or GPU, and displayed as a single scaled bitmap.

OP Overrun Behaviour

Interestingly, it appears that the OP will continue to process the remaining elements from the OPL even after the video hardware has switched over the line-buffers to start the next line. It will then start to draw the next line of video, which, since my test cases began with a RMW background image, meant that the enemy sprites from the previous line were visible underneath the background.

I cannot think of any use for this behaviour, and the workaround is obviously just to not place such heavy demands on the OP.

Sprite Aliasing Issue

During the later stages of development I discovered one additional issue with interlaced video that was not immediately apparent, but with hindsight is obvious. Any sprite which moves vertically at a rate at or near to 1 pixel per update will only have every other line drawn. This is because in the time it takes to draw the alternate field of the display, the sprite will have moved 1 scanline vertically, resulting in the same rows of image data as in the previous update being drawn in their new location one line above or below their previous location.

This is only likely to be a problem if the sprite has a lot of detail, or if significant details lie on interleaved scanlines. If the sprite is moving at a speed close to, but not quite one pixel per update vertically, it can be observed to toggle between display of the odd and even scanlines at a rate proportional to the fractional part of its speed. This issue can easily be worked around by either halving the rate at which updates happen, or by ensuring that sprites do not get assigned vertical speeds close to one pixel per update, however I believe it to be a minor issue and not obvious enough to be noticed. It was only visible to me because I had a sprite consisting of alternating lines of colour, designed to test the interlaced drawing routines.

Observations

I have learnt quite a lot during the course of this project, and my ideas regarding various aspects of game programming have evolved significantly. A game of this type is most heavily influenced by its AI from the players perspective, and by the graphics routines from the programmers perspective.

Drawing Method Performance

One of the Jaguar's strengths is the flexibility it offers developers when it comes to getting things drawn on screen. Both the OP and blitter are very fast and very flexible, however each one leads to a different methodology. The OP is essentially a sprite engine, and can draw images of varying sizes and bit depths, compositing them into a single line buffer immediately before each scanline of video is sent to the display. The blitter, on the other hand is designed to move and combine graphics at high speed, leaving a complete image in memory that can be processed further or drawn to screen by the OP.

The strength of the Jaguar is that you can easily combine both techniques to a greater or lesser degree, and it is entirely up to the programmer how they want to draw things. However that flexibility also comes at the cost of making it hard to estimate exactly how much work the system can do, or the best technique to achieve a given task.

This combination of the two approaches came about since the OP was designed for the Panther, while the Jaguar design added the GPU and blitter to do advanced graphical manipulation and 3D rendering[23].

Measuring OP performance is incredibly difficult. It cannot be run in single-step mode, it does not produce any kind of indication of the time taken or remaining for each line, and it is not possible to profile how long each step of any given processes takes without the use of a logic probe.

The developers' manual[4] gives some indications of the kind of performance one can expect from it on a per-object and per-clock-cycle basis, but there are too many variables to easily extrapolate a full estimation of how much it can do, given the hard limits imposed by the video standards timing, and the nature of a multi-processor system using DRAM with variable timing.

The blitter is easier to measure, since its operation is entirely under programmer control, and it needs to be reset after performing each task. It is however given a low bus priority (only the CPU is lower), which means other tasks running in the system can impact blitter performance.

Linear OPL

The approach I initially took was to use the OP for all drawing, with a long, linear, OPL, and it worked well, despite the difficulties inherent in debugging object-creating code. However, it became apparent that, whilst the game was only partially complete, performance was becoming an issue, and I was approaching the limit of how much could be drawn to screen.

Fragmented OPL

The OPL is not technically a list at all, but rather a graph. Through the use of branch objects, it is possible to fork the list into two possibilities, based on a number of variables but most commonly the current scanline. Similarly, since the link field is just a memory address, it is possible to merge multiple paths back together.

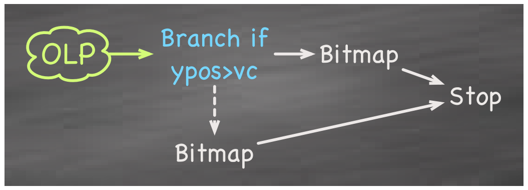


Figure 9: Branching and merging in the OPL.

This can be used in situations where there are a huge number of objects to process, such as a character-mapped display with each character being a separate object[1].

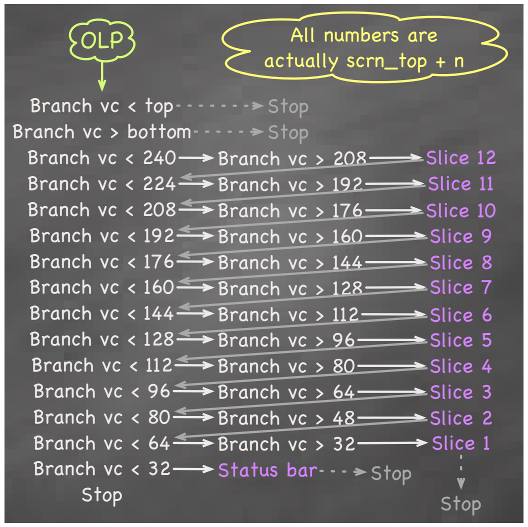


Figure 10: Breaking the OPL into slices, note the use of branch object fall-through (down the columns).

I spent some time formulating an approach that would allow me to divide the screen into thirteen horizontal slices, and point the OP at two slices on each pass (Figure 10). It is necessary to process two slices per scanline in order to account for the height of the sprites, and the fact that they do not neatly line up in a grid (as would be the case with a character-mapped display).

In the end I did not implement this, since I was not convinced the performance gains outweighed the added complexity of the list structure. It may be useful in a "bullet-hell" type game, with hundreds of small bullet sprites being drawn simultaneously, but did not seem necessary for this game.

Single Framebuffer

A very common technique on computers and older games consoles, is the use of a single frame buffer, into which everything is drawn, either by the CPU or with the assistance of a dedicated co-processor or blitter. By using two frame buffers, one of which is displayed while the other is being written to, a game can largely de-couple itself from the underlying video rate, with each of the buffers being displayed for as many frames as needed to update the entire display.

This technique places less critical demands on bus time, but can be considered inefficient due to the need to clear and re-draw the entire display (including backgrounds and other elements that may not have changed) each frame. It also uses significantly more memory than the previous methods.

Multiple Framebuffers

Given the flexibility of the Jaguar's OP, another approach becomes practical. By creating multiple frame buffers in memory each one can be used for a different part of the display (backgrounds, enemies, bullets, graphical overlays, etc) and then combined together at display-time by the OP.

This places a more significant burden on bus time and memory than a single framebuffer, since the OP must fetch multiple full-screen images each frame, but that demand is more consistent and predictable than using the OP to draw each individual sprite. The advantage is that it allows for each "layer" of the display to be maintained and updated independently (and at differing refresh rates if desired).

Artificial Intelligence

Due to not getting quite as much progress done as I had hoped for, the game in its current state features rather little AI. Since the AI is a rather important consideration of this game, and has been the subject of considerable thought and planning, I feel I should go into some detail about my plans here.

In the following sections it is worth remembering that my game utilises fixed-point maths with four fractional bits for both speeds and positions, allowing for smooth movement in all directions[32].

Simple AI

The most basic chase AI used in this style of game consists of just two tests: is the player above, below, or on the same level as the enemy, and is the player to the left, right, or central to the enemy. For each test, if the player and enemy are not on the same row/column as each other, move in the appropriate direction. This yields a rather crude 8-way chase AI, however I did not, and do not plan to implement this system in this game. It is worth mentioning as it is both very common and computationally cheap, it has two obvious flaws.

The first flaw (although it is arguably a feature) is that it creates a very predictable AI, which has a strong bias to diagonal movement. This can be used — in games where the AI enemies interact with the environment — to the player's advantage[8]. When moving diagonally, enemies using this AI scheme will actually move faster than when moving along the cardinal directions[20]. This is because the two checks (move up/down, move left/right) are done independently, and each just sets the vertical or horizontal speed.

Implementing fully unconstrained movement in a chase AI is very simple in theory, but prohibitively computationally expensive. The simple, and idealised solution would be to subtract the players position from that of the enemies, yielding a vector from the enemy to the player, and then to normalise that vector and multiply the result by the desired enemy speed. The difficulty comes from the fact that vector normalisation requires the calculation of one square root plus one division for each dimension (in this case two), all of which are prohibitively expensive operations.

A method I have come up with, but not yet had time to implement[9] is to implement 16-way movement through a series of boolean checks. While it will not be quite as effective as fully unconstrained movement, it should provide smooth enough movement to look indistinguishable when the player is being chased, and ensures that the units speed is in all directions.

The method involves at most two jumps and should execute very quickly. It works by taking the differences in position between the player and the enemy, separating the values into their sign and absolute values, then comparing if the larger component is more than double the smaller component, as illustrated in figure 12.

Although eventually destined to have a bitmap based swarming AI, the Grid Bug enemies (see page 9) currently have a chase AI that gives them a rather distinct behaviour. It operates on a similar principle to the 16-way AI previously described, first taking the differences between the player and Grid Bug positions, then separating them into sign and magnitude. After that, it simply tests which component is larger, and moves in that direction, adjusted for the sign of the values. This ensures the Grid Bugs maintain their characteristic movement strictly along the cardinal directions, while also pursuing the player.

[8]In a maze-type environment where enemies only move along fixed paths (e.g. platforms and ladders) learning to manipulate the crudeness of the AI to send enemies down the wrong paths is often a key part of the challenge.

[9]Hopefully it will be implemented by the time of the project demonstration.

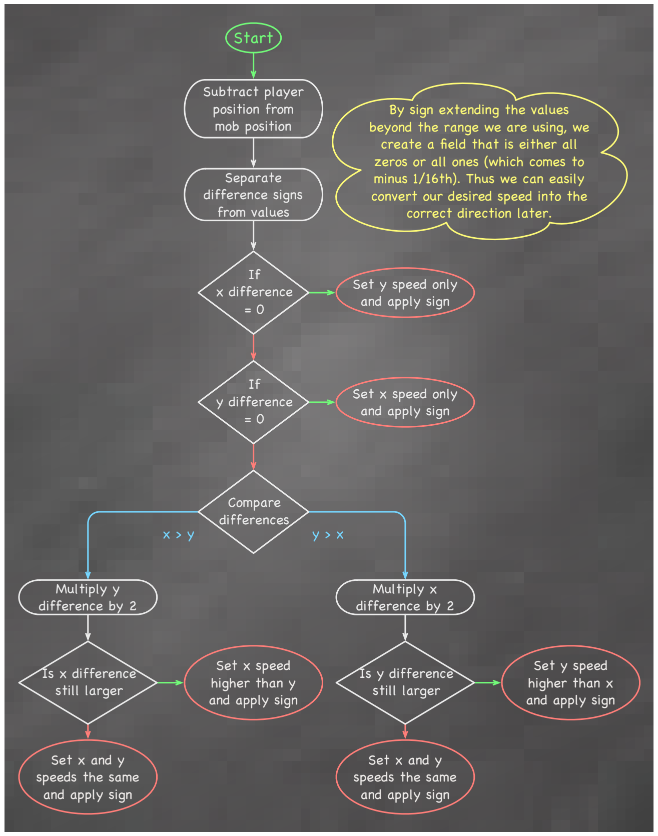

Start
Subtract player position from mob position
Separate difference signs from values
By sign extending the values beyond the range we are using, we create a field that is either all zeros or all ones (which comes to minus 1/16th). Thus we can easily convert our desired speed into the correct direction later.
If x difference = 0
Set y speed only and apply sign
If y difference = 0
Set x speed only and apply sign
Compare differences
x > y
y > x
Multiply y difference by 2
Multiply x difference by 2
Is x difference still larger
Set x speed higher than y and apply sign
Is y difference still larger
Set y speed higher than x and apply sign
Set x and y speeds the same and apply sign
Set x and y speeds the same and apply sign

Figure 11: Flowchart of the 16-way simple AI.

Bitmap AI

The simple AI schemes are all inherently limited to operating on the relationship between the player and each individual enemy unit, and while that is fine for some enemies, more advanced behaviour requires a more advanced understanding of the entire game state. This can be achieved with bitmap based AI schemes. Unfortunately due to time constraints none of these were implemented.

The basic premise of the bitmap AI schemes is that by computing a number of maps of the playfield, information becomes available that would otherwise be expensive to calculate each time it is required. That information can then be read cheaply and used to create complex behaviours with low per-unit computational overhead.

Taking as an example the case of the well known flocking algorithm. It is generally implemented by — for each unit in the flock — finding the n closest neighbours of that unit, calculating the average of their positions and headings, and adjusting the unit's own position and heading to match[7].

Even ignoring the maths involved in calculating headings, the basic algorithm requires that, for each unit, the program must examine every other unit in the flock, just to determine if they are of further interest. This is woefully inefficient, and impractical on a system as constrained as the Jaguar.

If instead, each member of the flock added[10] a Gaussian 'bump' bitmap to an image of the playfield, centred at its own co-ordinates, then the cumulative effect would be to create a distribution map of the flock, getting brighter in areas where there were more members, and dimmer where there were fewer. It is then a simple matter for each unit in the flock to examine the four or eight tiles surrounding its position and head toward the brightest one.

The playfield-maps produced by this method need not be the same resolution as the playfield itself, and in fact doing so would be prohibitively expensive to calculate. Similarly, they do not need to be updated as often as the rest of the game, and data calculated in one frame will still be useful for a number of frames to come.

[10] Using simple saturating pixel addition, a feature of the blitter but easily implemented in software.

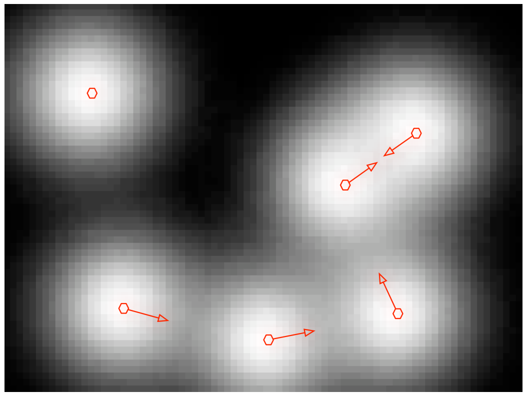


Figure 12: Bitmap AI based flocking example.

Flocking normally also includes an element of avoiding nearby units, which should be easily achieved by adding a 'dent' in the centre of the 'bump' bitmap, which is not as bright as the surrounding values. By the same method various other formations could be constructed, still with the same naive hill-climbing function, but different patterns added to the underlying map.

With this basic format it is easy to envisage a large range of different AI behaviours. Inverting the hill-climbing behaviour to prefer the darkest areas would create a solitary enemy, that try to scatter and spread themselves as evenly as possible. Or by combining a preference for lower density areas with a seeking behaviour, an enemy can be imagined which avoids other enemies but seeks out the player.

However enemies are not the only things of interest on the playfield. If we project a similar Gaussian 'bump' under each and every bullet, we get an accurate map of areas which are most dangerous to the enemy units, and should be avoided. An AI designed to avoid the players fire would provide an entertaining challenge and require different tactics to defeat.

The bullet-map image can be further enhanced by only dimming it on each refresh, rather than wiping it completely and starting afresh. This would allow the traces of each shot to linger for some time, slowly diminishing.

Similar Approaches

Although developed independently, my approach to this kind of AI has a lot in common with Alexander Repenning's paper on Collaborative Diffusion[24]. It functions by moving the processing of AI away from the individual units / enemies, and into the environment, such that the bulk of the work is done only once every update, rather than once per unit.

Where our approaches differ however is in the implementation details. Repenning's approach shows a clear bias toward an Object-Oriented Programming (OOP) mindset and introduces needless complexity and un-used flexibility. He proposes a network of floor tiles or patches, in which each one communicates its own state with its neighbours, and is in turn influenced by their states. Given that most games, and all of his example games, can be represented as a 2-dimensional grid, one can apply standard 2D image processing techniques on the data and save considerable time. Even a complex three-dimensional world could be represented adequately as a series of 2D bitmaps stacked above each other.

The AI Timer

As mentioned on page 12, the game uses a timer to give a variable delay between calls to the AI routines. During the intervening time, straight-line movement is carried out the update routine, which need only add the speed values to the position values (both of which are fixed point numbers for smooth movement[32]). When the timer reaches zero (or if the enemy collides with the play-field boundary), the appropriate AI routine is called, and a new set of speed values, as well as a new value for the AI timer, are calculated.

Even with the simple AI routines currently implemented, it is clear that the values used to set the timer have a significant impact on the behaviour of the enemies. Setting it too high results in their wandering far away from their ideal behaviour, setting it too low results in highly aggressive AI as well as, in the extreme cases, noticeable system slowdown.

More subtle is the range of AI timer values. In order to avoid bottlenecks in performance as many enemies calculate AI at once, the values given to the timer should always be randomised somewhat. The effect of a small range of values is to create a fairly consistent behaviour, which, when generalised over a group of enemies, creates an orderly appearance amongst them. A large range however can create a more diverse, less ordered behaviour, with enemies sometimes keeping their heading for a long time, and at other times appearing indecisive or even chaotic.

Conclusions

This module of work is drawing to a close, but I intend to continue with this project for another six to twelve months, by which time I hope to have the game complete and ready for release. Thus far my focus has been to demonstrate as many features as possible, and to effectively prove the concept of my initial design. As I continue with the project I will re-write large sections of it for increased speed and add the remaining features not yet implemented.

Lessons Learnt

One of the key lessons I have learnt from this project is the awkwardness of debugging one thread of a multi-threaded system. Specifically, the difficulty encountered in single-stepping through the process of OPL creation when the OP is busy processing the head of the list.

Had I been more aware of the extreme difficulty of diagnosing and repairing those sort of bugs from the start, I would have designed the game in such a way as to minimise the amount of OPL construction needed – either by producing generic subroutine calls to create objects (marginally less efficient in terms of CPU usage), or by entirely re-designing the game to use a (multiple) frame buffer design.

I have also done considerable research into the history of the Jaguar and learnt more about the people and events surrounding its creation. I have come to have a better understanding of what the system is capable of and of how much effort can go into making even a relatively simple game like this one. I have also discovered new bugs in the Jaguar silicon, and explored workarounds for them.

I also learnt a great deal about LaTeX. It has been used for all of these reports, which to me count as a significant technical achievement in and of themselves, due to the programatic nature of LaTeX documents and the amount of work put into programming the structure and presentation of them. Prior to this project I had never used TeX or LaTeX before.

Overall Self Assessment

On reflection, I am proud of what I have produced for this project. Of the three key objectives I set out to achieve (see page 9), I have achieved two of them, and shown that the third is technically impossible (page 16).

I have also created a range of new software tools for the developer community and demonstrated a new kind of hardware controller for the Jaguar that I intend to produce.

Far more of my time than I had ever anticipated was spent writing OPL generating code, and debugging it when things went wrong. I had also not anticipated quite how much real life — as well as other project related tasks such as planning and report writing — would impact on the amount of time I could spend programming.

Future Plans

I plan on continuing work on this game, with a view to an eventual release. To that end my immediate goals are to implement one or two additional AI schemes, get bounding-box based collision detection working, and a basic state machine. After that, I will start to port the code over to the GPU, possibly changing the drawing style to use a multiple framebuffer approach at the same time.

Appendices

Appendix I Joystick Controller Circuit

The Jaguar uses a 15-pin high density D-shell plug for its controller interface. Two of those pins were used for analogue inputs initially but that usage was quickly depreciated and later models of the console do not feature the Analogue to Digital Converter (ADC) chips necessary for analogue input. Of the remaining pins, two are used to carry +5V and ground, and one is not connected. This leaves the 10 data pins.

On each controller socket, there are 4 output pins, and 6 input pins. In typical usage a Jaguar game will poll each of the 4 output lines one at a time, then read back from the 6 input lines. A matrix of diodes and switches inside the controller will connect each switch to one input and one output line. This is complicated somewhat by the use of negative logic throughout, and by the addition of a non-inverting buffer IC to ensure the signals returned to the Jaguar are at a uniform voltage.

For more details of the interface see [6].

I set out to create a new universal circuit design over the summer shortly before beginning this project. I had previously created a line of Rotary Controllers for T2k based upon original Atari control pads. My first one was produced in 2002[34], with production and sale continuing for several years after that. I sold the last few Rotary Controllers in 2010, and, since there seemed to be demand for a new line of them, set about creating a new circuit design that would not require the modification of original control pads, and with all components being easily sourced. To this end I designed it such that it could be produced on strip-board.

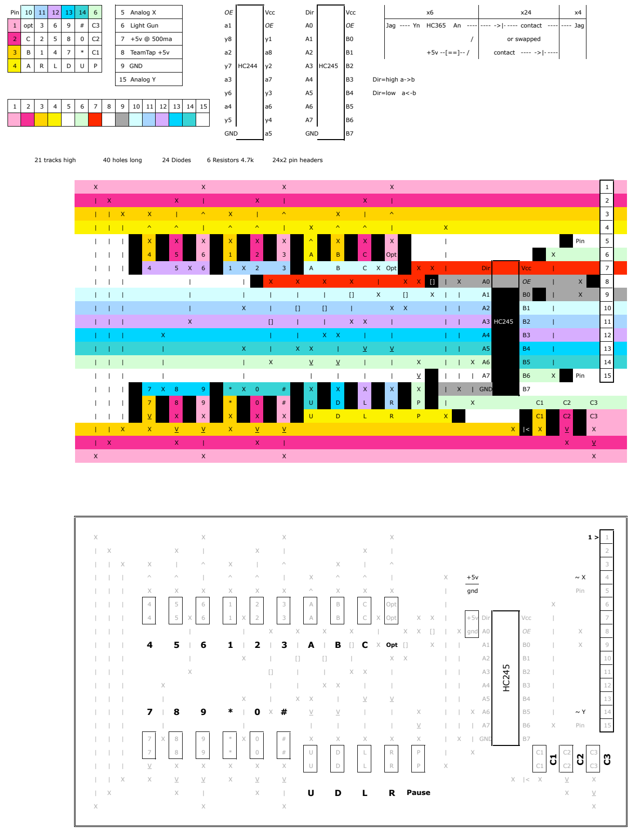


Figure 13: Jaguar controller circuit board schematic. Solder side with colour-coded traces (above), and component side (below).

Appendix II References

[1] Atari. Tree Demo. Example Jaguar program, 1993.

[2] Atari. *Appendices to the Jaguar Manual*, April 26th 1995.

[3] Atari. *Hardware Bugs & Warnings*, April 26th 1995.

[4] Atari. *Jaguar Software Reference Manual*, June 7th 1995.

[5] Atari. *Jaguar Technical Overview*, April 10th 1995.

[6] Atari. *Jaguar Technical Reference Manual*, April 26th 1995.

[7] David M. Bourg and Glenn Seemann. *AI for game developers*. O'Reilly Media, Inc., 2004.

[8] Mark Campbell. Interview with Martin Brennan. 2011. http://www.konixmultisystem.co.uk/index.php?id=interviews&content=martin.

[9] Edge. Atari: Cat among lions? *Edge 1996 Essential Hardware Guide*, Autumn 1995.

[10] Edge. Atari: From boom to bust and back again. *Edge Magazine*, March 1995.

[11] Edge. Atari. *Edge Magazine*, February 2002.

[12] Edge. What Atari did next. *Edge Magazine*, February 2002.

[13] Edge. The making of: Tempest 2000. *Edge Magazine*, April 2005.

[14] Carl Forhan. *Total Carnage Game Manual*. Songbird Productions, 2005.

[15] Next Generation. Atari's president talks back. *Next Generation Magazine*, July 1995.

[16] Hasbro Interactive. Press release, May 1999. Available at www.atariage.com/Jaguar/archives/HasbroRights.html.

[17] David Hotz. Small fast 16-bit PRNG, May 2009. http://codebase64.org/doku.php?id=base:small_fast_8-bit_prng.

[18] Randall Hyde. *The art of assembly language*. No Starch Press, 2003.

[19] Frans Keylard. Catnips... Jaguar tidbits from Don Thomas. Usenet news posting, October 1995. http://groups.google.com/group/rec.games.video.atari/msg/f0b5572a76d656de.

[20] Michael Molinari. Lag, control, and you. http://shmuptheory.blogspot.com/2010/01/lag-control-and-you.html, 12th January 2010.

[21] Bloomberg Business News. Atari agrees to merge with disc-drive maker. *The New York Times*, February 14th 1996. http://query.nytimes.com/gst/fullpage.html?res=9A0CE5DB1239F937A25751C0A960958260.

[22] Andy Nuttall. Jaguar developers revealed. *ST Format Magazine*, pages 8–9, December 1993.

[23] QueST. Interview with John Mathieson - the “father” of the Jaguar, August 2002. http://www.landley.net/history/mirror/games/mathieson.htm.

[24] A. Repenning. Collaborative diffusion: programming antiobjects. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 574–585. ACM, 2006.

[25] Paula Richards. Atari launch jaguar to uk press. *ST Format Magazine*, pages 8–9, November 1993.

[26] Paula Richards. Atari launch their “billion dollar baby”. *ST Format Magazine*, pages 8–9, August 1993.

[27] Paula Richards. Atari launch Jaguar. *ST Format Magazine*, pages 7–9, January 1994.

[28] Paula Richards. Jaguar news. *ST Format Magazine*, pages 45–48, March 1994.

[29] Paula Richards. Jaguar news. *ST Format Magazine*, May 1994.

[30] Paula Richards. Jaguar news. *ST Format Magazine*, pages 47–50, April 1994.

[31] Paula Richards. Jaguar the next big thing. *ST Format Magazine*, pages 83–98, February 1994.

[32] TAD. Fixed point maths. *HUGI Demo-Scene Disk Magazine Issue 15*, May 1999. Available at http://hugi.scene.org/.

[33] New York Times. Sega investing $90 million in Atari. *The New York Times*, September 1994. http://www.nytimes.com/1994/09/29/business/company-news-sega-investing-90-million-in-atari.html.

[34] Nick Turner. Acme co. patented ass whomping machines. *AtariAge Forum*, May 2002. http://www.atariage.com/forums/topic/4154-acme-co-patented-ass-whomping-machines/.

[35] Curt Vendel. Jag encryption - round 2. *AtariAge Forum*, November 2003. http://www.atariage.com/forums/topic/37617-jag-encryption-round-2/.

[36] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[37] J. Zagal, C. Fernández-Vara, and M. Mateas. Gameplay Segmentation in Vintage Arcade Games. *Ludologica Retro*, Vol 1:1971–1984, 2007.

[38] Zerosquare. Hi-res on Jaguar (and interlaced mode), April 29th 2006. http://www.jagware.org/index.php?showtopic=230.

Appendix III Glossary & Index

ADC *Analogue to Digital Converter.* A chip designed to convert an analogue voltage into a binary representation. 25

AI *Artificial Intelligence.* The act of creating behaviours which seem to show intelligence, or are related to learning or improving over time. 1, 11, 12, 14, 17, 19–24

ALU *Arithmetic Logic Unit.* The portion of a microprocessor IC responsible for mathematical and logical processing. 6

ANSI *American National Standards Institute.* Standardisation body in the United States of America. 13

ASCII *American Standard Code for Information Interchange.* The standard format for English text. A 7-bit format that, while largely considered obsolete, is still widely used. The 128 ASCII characters form the lowest portion of most other character encodings, for backwards-compatibility reasons. 13

Atari *Atari Inc. Atari Corp.* Influential company responsible for many early consoles, arcade machines and home computers. 1, 3–6, 8, 10, 12, 14, 15, 25, 29, 30

ATD *Attention To Detail.* Small development studio responsible for several Jaguar games including Cybermorph, Battlemorph, and Blue Lightning. 3

BJL *Behind Jaggy Lines.* Replacement boot code for the Jaguar supporting code upload into memory via a specially constructed parallel cable. 5

Blitter *Bit-Block Transfer Engine.* A dedicated processor designed to rapidly copy blocks of memory. The Jaguar's Blitter has a number of advanced address generation features allowing it to do image rotations, line drawing and shading, in addition to simple rectangle moves and fills. 3, 6, 7, 10, 14, 17–19, 21

Bus *Data Bus.* The main data path of a computer system. 5, 6, 8, 11, 18, 19

C *The C Programming Language.* Probably the most widely deployed and most prominent programming languages of all time. 13, 28

C++ *C Plus Plus.* Object Orientated language based on C and originally created as an extension of it. 13

CES *Consumer Electronics Show.* A large trade show traditionally held in Las Vegas. 3

CISC *Complex Instruction Set Computer.* A design philosophy encouraging more powerful and programmer-friendly machine-code instructions. Compare with RISC. 6, 10, 28, 31

CPU *Central Processing Unit.* A Motorola 68000, 32 bit CISC processor with 16 bit data bus. 6, 10, 11, 16, 18, 19, 23

Cross Products *Cross Products Ltd.* Creators of the SNASM cross-platform development system, designed to help developers produce code that ran on a wide range of systems. 4

CRT *Cathode Ray Tube.* A traditional design for televisions and monitors. 31

CRY *Cyan, Red & Intensity.* A somewhat unconventional colour model used in the Jaguar, featuring two 4-bit colour vectors and an 8-bit intensity value. 7, 9, 14, 31

D-Pad *Directional Pad.* A cross-shaped button able to recognise input in four or eight directions. 8, 12

DRAM *Dynamic Read Only Memory.* A type of RAM that requires periodic refreshing to prevent data loss. 6, 18

DSP *Digital Signal Processor.* A custom-designed RISC processor intended for sound generation and music playback. 6, 8, 10

Field *Half-Frame.* One half of an interlaced video signal, two fields make up one frame. 10, 15–17, 29

FPS *Frames Per Second.* A measure of a game's performance. The number of times it updates the display each second. 10

FSM *Finite State Machine.* A simple form of control system that can be used to select between a finite number of states that the system can be in, with clear transitions between states. 15

GPU *Graphics Processing Unit.* A custom-designed RISC processor intended for graphics manipulation. 3, 6, 7, 10, 16–18, 24

Hasbro *Hasbro Interactive.* A division of Hasbro Inc. that dealt with video game production and publishing. 5

HSL *Hue, Saturation, Luminance.* A colour model that attempts to model the way artists think of colour. 7

IBM *International Business Machines.* Giant of the computing world, making everything from cash machines to mainframes and responsible for the PC as we know it today. 4

IC *Integrated Circuit.* An electronic circuit designed to do a specific, well documented task, mass produced on a microchip and designed to be used as a component in a larger circuit. 6, 12, 25, 28, 31

Infogrames *Infogrames Entertainment, SA.* French publisher and distributor of computer and console games. 5

Interlace *Interlaced Scanning.* The technique of transmitting video in alternating fields of alternating scan-lines. 10, 15, 17, 29

Jaguar *Jaguar 64-Bit Interactive Multimedia System.* Ill-fated games console that was the final product Atari were to release before their eventual demise in 1996. 1, 3–10, 12–16, 18, 19, 21, 23–25, 28, 29, 31

Jeff Minter *aka Yak.* Programmer, game designer, hippy, and founder of Llamasoft. Responsible for innumerable classic and psychedelic games, as well as most of the techniques used in modern audio visualisers and sound-to-light systems. 3, 7, 31

JTS *JT Storage Corporation.* A little-known manufacturer of hard drives and storage solutions, merged with Atari in a reverse-takeover in 1996. 3, 5

LFSR *Linear Feedback Shift Register.* A type of PRNG that uses shifts and exclusive or operations to permute it's state. 12

NTSC *National Television System Committee.* The standard for television transmission used in the USA and other territories. 10, 31

OLP *Object List Pointer.* The register in the OP which contains the start address of the OPL. 18, 20, 22

OOP *Object-Oriented Programming.* A programming and design methodology promoting the idea of systems being built out of self-contained and largely independent objects. 22

OP *Object Processor.* A complex sprite engine capable of drawing scaled and un-scaled bitmaps to screen in a number of bit-depths, colour models, and drawing modes. 3, 6, 7, 10, 11, 13–19, 23, 30

OPL *Object Processor List.* A list of tightly packed "object" definitions, used as instructions for the OP. 6, 10, 11, 13, 16–18, 23, 24, 30

PAL *Phase Alternating Line.* The standard for colour television transmission used in the UK and most of Europe. 10, 31

Panther *Atari Panther.* A 32-Bit games console partially designed then scrapped by Atari in favour of the Jaguar. It was to have been launched in 1991. 3, 18

PBM *Portable Bit-Map.* A very simple file format for storing 1-bit per pixel images. 13

PGM *Portable Grey-Map.* A very simple file format for single channel images. 13

Phrase *64 bit.* A slightly non-standard term used in the Atari documentation for a 64-bit value. 6, 13

PPM *Portable Pix-Map.* A very simple file format for storing images with three separate channels per pixel. 13

PRNG *Pseudo-Random Number Generator.* A software routine designed to produce a seemingly random series of values. 12, 30

Qt *Qt framework.* Cross platform open source window and widget toolkit, libraries and frameworks. Produced by Trolltech and currently owned by Nokia. 13

RAM *Random Access Memory.* A type of memory which can be written to and read from in any order. 5, 6, 29

RGB *Red, Green, Blue.* A colour model that uses three values of roughly equal size to represent the levels of Red, Green and Blue that make up a given colour. 7

RISC *Reduced Instruction Set Computer.* A design philosophy encouraging more simple and compiler-friendly machine-code instructions. Compare with CISC. 6, 10, 28, 29

RMW *Read, Modify, Write.* The process of blending a CRY image into the line buffer and thus all the images previously drawn that frame. 7, 9, 12, 17

ROM *Read Only Memory.* A kind of IC that stores a fixed block of data that cannot change. 5, 6

SDTV *Standard Definition Television.* The standards (PAL and NTSC) that relate to traditional television signals. 9

Sega *Sega Corporation.* Large Japanese video game manufacturer responsible for a long line of arcade and home consoles, including the highly successful Master System and Mega Drive systems, as well as innumerable software titles. 4, 6

T2k *Tempest 2000.* 3D shooting game by Jeff Minter, widely regarded as being one of the best games on the Jaguar. 5, 7, 9, 25

TC *Total Carnage.* Top-down dual-stick shooting game produced by Midway for a number of systems and released by SongBird Productions for the Jaguar in 2005. 9, 10, 12

VBL *Vertical Blanking Interval.* The time during which a Cathode Ray Tube (CRT) turns off the electron beam and re-aligns it to begin the next frame. 12, 16

VC *Vertical Count.* The register in the Jaguar video hardware which contains the current line number being drawn. 15, 16, 18, 20, 22

Appendix IV Updated Gantt Chart

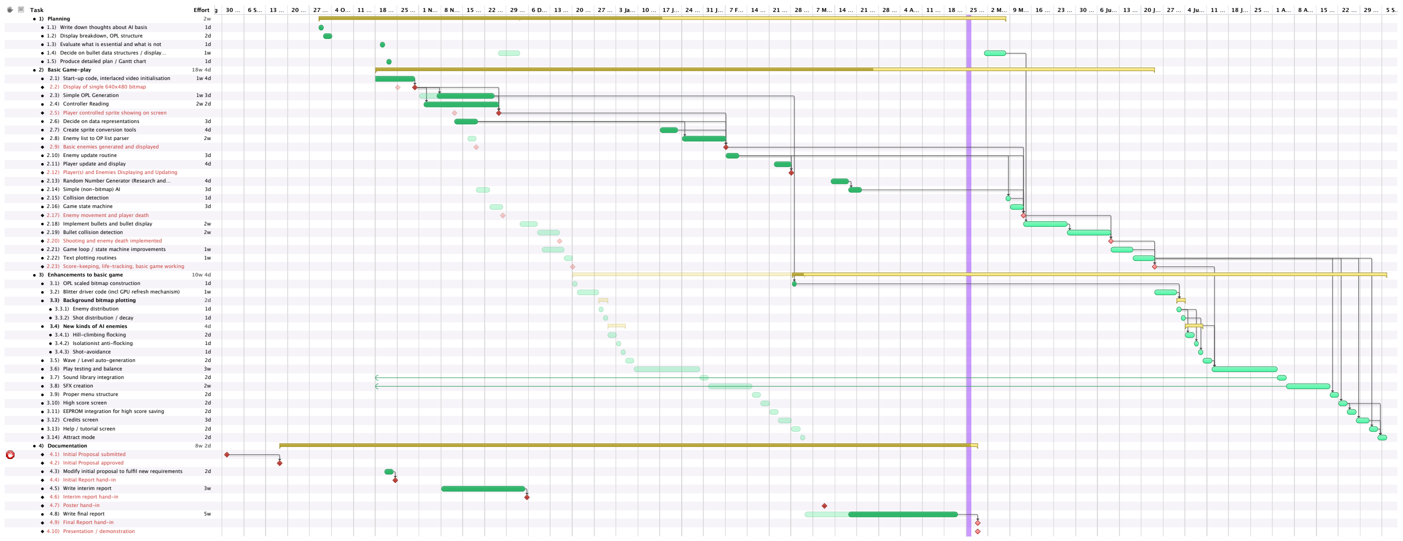


Figure 14: Updated Gantt Chart. Apologies for any illegibility due to scale issues. The electronic submission of this report can be zoomed in.