1 Introduction

This report is about studying what it would be a good idea to include in a game engine for the portable game console Game Boy® Advance (GBA) made by Nintendo®.

To do so, these items have been considered:

- What does a game engine consist of?
- What is the GBA capable of?
- What should a game engine for the GBA include?

I have also implemented a relatively simple, though perfectly usable, game engine, as well as provided a simple game, that uses this game engine. Furthermore, I have made a couple of straightforward programs that convert images of standard formats, such as Jpeg, Bmp, Png etc. to C-source code. That way images can easily be included in home-made games for the GBA.

The report was written by Otto Iohannes Monrad Kirk as a written paper at the Department of Computer Science at the University of Copenhagen, DIKU, in the period of February through July 2006.

In this paper, some registered trademarks are mentioned repeatedly and so to keep the record straight, it should be noted that Nintendo®, Game Boy®, Game Boy Advance®, Game Boy Advance SP®, Game Boy Micro® and Nintendo DS® are all registered trademarks of Nintendo of America and/or other countries. Also, Microsoft® and Windows® are the registered trademarks of Microsoft Corporation in the U. S. and/or other countries.

Throughout the report I mention three categories of people who are important in relation to game engines. They are

- **The game engine programmer**. In this case, it is yours truly.
- **The game programmer**. These are the people using the game engine to program games.
- **The game player**. These are the people playing the games that are made with the game engine.

1.1 Motivation and Perspective

One reason for making a game engine for the portable game console, Game Boy® Advance (GBA), is much like the reason for climbing a mountain, which is "Because it is there". Though, in the case of making a game engine for the GBA, the reason is: Because it is not there. Not a publicly available game engine, anyway.

However, there are several more down-to-earth reasons as well, which is what I will elaborate on below.

The reason why I have chosen to write a game engine for the Game Boy® Advance is to make it easier and faster to produce games for game programmers that are not affiliated with a major game development company. One of the main things that can assist in this area is a game engine. In fact, as most game development companies are probably aware of this, chances are that a lot of game engines for the GBA already exist. They are, however, not publicly available.

This means that games made for the GBA are typically made by one of the major game software development companies. The rest either take a huge amount of effort to produce or are very simplistic in nature, such as Tetris-clones and the like.

With a suitable game engine, games can be produced faster and in fewer man hours, thus resulting in lower production costs. Furthermore, a suitable game engine can hide the GBA specific code from the programmer, thus making it possible to program the GBA without possessing specific knowledge about the GBA hardware and how the GBA functions internally. Also, the lower production costs and fewer man hours could help make game development for smaller game companies or even single persons possible.

One of the results of a shorter production time for games, that is made possible with a suitable game engine, is that games have the opportunity to reflect current affairs, which means that some games might be just as up-to-date as e.g. newspapers. It stands to reason that this is only possible with relatively simplistic games, but naturally, the complexity of the games will increase with the abilities of an available game engine.

The benefit of making up-to-date games was illustrated recently. The very next day after the World Cup Final of Football 2006 was played; a simple game inspired by a single incident of the match was made public. The game was promptly mentioned in the Danish media; e.g. on the web site Game Section [GAMESECTION] the day after the match was played, and at the web page of Ekstra Bladet [EB] less than 36 hours after the match. The game is based on an incident in the match, where the football star Zinedine Zidane got a red card and was sent off because he butted an opponent, Marco Materazzi, in the chest. Thus, the object of the game is to butt as many opponents as possible. A scene from the game can be seen in Figure 1, below.

Velkommen til spillet -zidane-
By TonaZ alberto.zanot@alice.it

Figure 1: A scene from the Zidane game.

Even though the game is very simplistic in its nature, the fact that it is based on one of the major topics of conversation at the time has made sure, that there was a huge interest for the game. In fact, the interest for the game has been so immense, that of the two web-pages I have found hosting the game, one [WIDELEC] was completely down at the time and the other, Hyggestedet, [HYG], had to warn its visitors to be patient since the immense interest for the game had caused their server to be very busy. The latter web site, Hyggestedet, being a place that is plastered all over with advertisements; the extra number of visitors must almost unavoidably have meant more money for the people behind.

The reasons to study and make a game engine for the GBA are summed up in the table in Figure 2 below.

With a game engine	**Without a game engine**
The demand for hardware specific knowledge is kept at a minimum.	Advanced knowledge of the GBA hardware is necessary, which is hard to obtain, as hardware documentation from Nintendo® is not available for independent developers.
Less amounts of source code for the game programmers, which help making the source code for the game well-arranged, which will especially help projects with more than one developer.	More source code, as everything has to be made for every game. This might lead to badly arranged source code, resulting in extra difficulties, particularly in projects with more than one developer.
Fewer man hours needed, since some tasks are handled in advance. The result of this is lower production costs, a shorter time frame for the programming phase and/or the possibility for more current games.	Longer time frame for programming a game, since everything has to be made from scratch, resulting in unnecessarily many man hours needed and high production costs.

Figure 2: The reasons to study and make a game engine for the GBA.

The reasons for choosing to study and make a game engine particularly for the GBA, in contrast to any other game console or even a general purpose computer such as a PC, are rather subjective, as they evolve around the physical equipment I have access to, as well as which software development environments I have previously dipped into. And so, the GBA was chosen because I have the hardware equipment to transfer home-made games to an actual GBA unit, as described in section 2.4.2.2, and because I have previously made a small game for the GBA as a hobby project, using the programming environment Visual HAM, which is described in section 2.3.1. Furthermore, even though newer and more powerful handheld consoles have emerged from Nintendo®, i.e. the Nintendo® DS, and from other companies as well, it is my judgement that the choice of the GBA is a valid one, because the GBA is still going strong, which is shown by the fact that as of mid July 2006, on the Nintendo® web page, [NINTENDO2] it is stated that almost 100 games will be released in a foreseeable future as well as the fact that a new version of the GBA, i.e. the Game Boy® Micro®, was released as late as the year 2005. This game console is smaller than the original GBA, but plays exactly the same games.

1.2 Reading guide

Due to the special nature of this report, where it cannot be expected that the reader is familiar with programming the Game Boy® Advance or that the reader is too familiar with game engines, I find it in the right place to supply the reader with a guideline to finding the sections that appear most interesting:

Chapter 1 is this chapter, and is mainly an appetizer for the rest of the report.

Chapter 2 is mainly concerned with the way the GBA works and since this report revolves around a rather specialized subject as mentioned above, namely the Nintendo® Game Boy® Advance, I find it reasonable to have a chapter solely devoted to going through some of the characteristics of the GBA. Here the hardware supported features of the GBA are described, such as the display, sprites, user input, sound handling as well as some software aspects, including which software development systems to use. The chapter also includes sections about transferring a game to the GBA. For a reader that is not too interested in the possibilities of the GBA, this chapter may be skipped.

Chapter 3 is concerned with 2D game engines in general, and is mainly a discussion of what I believe to be important features of a 2D game engine and how that might be achieved. The chapter is rather important in order to understand the line of thought in the report.

Chapter 4 combines the two chapters, 2 and 3. It does so primarily by discussing what could be included in a game engine for the Game Boy® Advance. This is what the report is really all about, and the chapter is essential to all readers of the report.

Chapter 5 is a about measuring performance and scalability of the programmed game engine. The chapter should be read by game programmers, as it offers some guidelines to choose a reasonable compromise between the frame rate of a game and the number of concurrent sprites.

Chapter 6 and 7 are about programs I have written; the test game Space Invaders 2 and tools for converting graphics for use with the GBA, respectively. These chapters are probably mostly interesting to game programmers.

Chapters 8 and 9 provide a description of the data flow, a brief program description and instructions for the game programmer on how to use the game engine, and they are mainly for the benefit of those who want to use the game engine to make their own game.

Chapter 10 is the conclusion. It should be read by everybody with the slightest interest in this project.

2 Introduction to programming the Game Boy® Advance

As I do not expect the readers of this report to be too familiar with the Nintendo® Game Boy® Advance (GBA), I will here give an introduction to its key features.

A great deal of effort has been put into finding out just how the GBA works and how to program it, as no official guide to GBA programming is publicly available. The reason for this is that the GBA is a proprietary system, so in order to get any support from Nintendo® for programming this system, it is necessary to become a licensed developer, which is only possible by being affiliated with one of the major game development companies and by signing a nondisclosure agreement.

Thus, it has been necessary to find information about programming the GBA elsewhere, namely the Internet, which has numerous web sites dedicated to that purpose. This means that in order to gather the information presented in this chapter, a very time-consuming information hunt has had to be performed. In addition to this, as most of these web-sites, if not all of them, are maintained by amateurs, it means that the quality is very varying going from excellent to decidedly defective, and even the best guides I have come across contain some defects, making the process of programming a long-drawn-out affair, as well.

Nevertheless, one unofficial book has been written about the subject of programming the GBA, which is *Programming The Nintendo® Game Boy® Advance: The Unofficial Guide* by Jonathan S. Harbour [HARBOUR1]. However, that book was never published, allegedly due to legal issues with Nintendo®, which means that the only place to find the book is on the author's personal web-page [HARBOUR2]. Throughout this project, that book has been the most important source of information about how the GBA works and how to program it, even though web sites such as gbadev.org [GBADEV], the GameBoy Advance Dev'rs [GBADEVRS] and Gameboy Advance Technical Info [GBATEK] also have provided me with invaluable information on those subjects; the two former being sites where developers share GBA relevant information and the latter being a collection of GBA hardware specifications.

2.1 Console vs. general purpose computer

The Game Boy® Advance is a game console, as opposed to e.g. a PC that is a general purpose computer. What this means for the GBA is that:

- It is proprietary, which means that available documentation is at best imperfect and in other cases incorrect.
- Every unit is identical to others, which means that if a program runs on one GBA unit, it will run on all GBA units, as opposed to e.g. a PC, where it is perfectly possible to make a program that runs on one PC, but not on another, because the hardware on PCs differ.
- It is hardware optimized for games. What this means, for example, is that effects such as moving, rotating and scaling backgrounds and sprites, etc. demand much less CPU power than they do on a system where these effects have to be achieved through software means. On a PC, for instance, in a simple game that consists of a static background with some sprites moving around, each sprite has to be drawn on top of the background, and when the sprite is moved, the revealed piece of background has to

be redrawn. On the GBA, with hardware controlled sprites, this is done automatically, as the sprites do not overwrite the background, but are merely shown on top of it.

- There is no operating system, which means that everything has to be made from scratch. For example, there is no built-in text system, so if text is needed, then graphical images of the alphabet as well as code to handle text must be provided by the programmer or the game engine.
- There is no file system, which means that every piece of data must be located in RAM, ROM, EPROM or Flash RAM. For games this means that data, such as images and sounds, must be placed in the ROM, EPROM or Flash RAM together with the game program.

2.2 Presentation of the Game Boy® Advance

The Game Boy® Advance is a handheld game console that was made by Nintendo® as a successor to the Game Boy® Color® and the original Game Boy®. There are three different versions of the Game Boy® Advance, namely the Game Boy® Advance (released in 2001), The Game Boy® Advance SP® (released in 2003) and the Game Boy® Advance Micro® (released in 2005). Images of the various versions of the GBA are shown in Figure 3, Figure 4 and Figure 5, below. Please notice that the images are not on the same scale.

Figure 3: The Game Boy® Advance.

Figure 4: The Game Boy® Advance SP®.

Figure 5: The Game Boy® Micro®.

Technically, they have the same specifications, except that the GBA Micro is not backwards compatible with Game Boy® and Game Boy® Color games. Furthermore, the latter two comes with a backlit display, which was one of the main things missing from the original GBA. Finally, the GBA SP is foldable, much like a laptop computer, and both the GBA SP and the GBA Micro are smaller in size than the GBA.

The primary goal for the Game Boy® Advance is games, mainly in 2D, as 3D graphics is not supported by the hardware. This has resulted in both the external and internal design of the console:

- On the outside, is a 61.2 * 40.8 mm screen with a resolution of up to 240 * 160 pixels with 32768 colours as stated on the Nintendo® web page [NINTENDO1], an eight way directional pad and six buttons, a built in speaker (mono) and a headphone jack (stereo), a plug for communication with other GBA units and a slot for cartridges.
- Inside it has 32-bit ARM processor embedded with 32 KB RAM and 96 KB video RAM and an additional external 256 KB RAM. It supports six different video modes, digital sound and up to 128 sprites and has hardware support for fast copying of memory. Sprites as well as some backgrounds in some video modes are supported by hardware optimized translation, i.e. movement, rotation and scaling.

The appearance of the various versions of the Game Boy® Advance can be viewed below in Figure 3, and the features of the GBA are described in more detail below:

2.2.1 The GBA Screen

As mentioned above, the screen on the GBA consists of 240 * 160 pixels with 32768 colours. This, however, is only the half truth, as there are six different video modes to choose from, each with its pros and cons, which will be described below.

2.2.1.1 Tile based video modes

In the tile based video modes the screen is made up from "tiles" which are blocks of 8 * 8 pixels, where the colour of each pixel comes from a palette with 256 colours, chosen from the 32768 colours available on the GBA. The palette is the same for all the tiles and must be specified elsewhere. This means that each tile consists of 64 bytes, each being an index that refers to a colour in the colour palette.

There are three tile based video modes on the GBA, which means that the screen image in these video modes is made up of three things: A number of tiles, a colour palette and a "map", showing which tiles are placed where.

Each of the tile based video modes consists of a number of backgrounds that are shown on top of each other. Depending on the video mode, each background has different properties. The number of tiles in each background exceeds the screen limitation, as a tile based background consists of up to 1024 * 1024 pixels, as opposed to the size that is 240 * 160 pixels, as mentioned earlier. With a simple instruction, it is then possible to decide which part of the background is to be shown on the actual screen, also called translation. Furthermore, some of the backgrounds can easily be scaled to practically any size and rotated by practically any angle. The following table shows which backgrounds has which properties, and what the maximum size of each background is:

Background number	Maximum resolution	Rotation/scaling
0	512 * 512	No
1	512 * 512	No
2	1024 * 1024	Yes
3	1024 * 1024	Yes

Figure 6: Backgrounds in tile based video modes

Which backgrounds are available in which video mode is shown in the following table, in Figure 7:

Video mode	Available backgrounds
0	0, 1, 2, 3
1	0, 1, 2
2	2, 3

Figure 7: Available backgrounds

There is one exception to this, and that is, that in video mode 0 neither scaling nor rotation is available - not even in background 2 and 3.

The available backgrounds will be shown on top of each other, and here it useful to know that the colour with index 0 is transparent, thus making it possible to see the backgrounds behind. Each background can be translated individually of the others, and the same goes for scaling and rotation for those backgrounds that support these features.

2.2.1.2 Bitmap based video modes

Three video modes are bitmap based, i.e. each pixel can be controlled individually of the others. Here is an overview of them:

Video mode number	Resolution	Colours	Back buffer
3	240 * 160	32768	No
4	240 * 160	256	Yes
5	160 * 128	32768	Yes

Figure 8: The bitmap based video modes

The fact that a video mode has a back buffer means that two individual buffers are present, where the graphics can be drawn. What buffer is shown on the screen can be changed by a single instruction, which makes those video modes (modes 4 and 5) suitable for situations, where you generate an image, and want to show it in an instant. The limitations of either the number of simultaneous colours (mode 4) or the resolution (mode 5) must be taken into consideration, when choosing which of the two video modes to use. Due to a hardware limitation, that states that the video buffers must be updated 16 bit at a time, in video mode 4, where each pixel consists of only eight bits, two pixels must be drawn at a time. This means that video mode 4 is relatively slow, if you want to draw one pixel at a time, as it involves reading two pixels, combining the result with a value for the pixel you want to draw and finally drawing two pixels.

As shown in the table in Figure 8, video mode 3 does not have a back buffer. On the other hand, it uses the full potential of the screen when it comes to the resolution and number of simultaneous colours, thus making video mode 3 suitable for still images and backgrounds in games where the background does not change during game play.

2.2.1.3 Considerations, when choosing video mode

The concept of tile based video modes is a bit harder to comprehend than the concept of bitmap based video modes. This is due to the fact that in order to show an image in a tile based video mode it is necessary to divide the image into tiles, and make an appropriate palette and an appropriate tile map. In the bitmap based video modes 3 and 5, the colour of each pixel is determined directly. In video mode 4 a palette is used to determine the colours available.

However, it is substantially faster to change the graphics in the tile based video modes, as writing a 16-bit value to tile map can result in a different tile being shown, which means that a block of 8 * 8 pixels can be changed by one instruction, as opposed to the bitmap based video modes, where writing a 16-bit value to the video memory can only result in one or two pixels being changed. Furthermore, translation, scaling and rotation of the background, as well as using several backgrounds on top of each other, are only possible in the tile based video modes.

2.2.1.4 Window feature

The GBA hardware supports windows, which means that two windows can be defined. This way the screen can be split up in different regions, which can be treated independently, when it comes to graphical effects, such as scrolling and rotation. The appearance of windows on the GBA is illustrated in Figure 9 below. This effect is only possible in the tile based video modes.

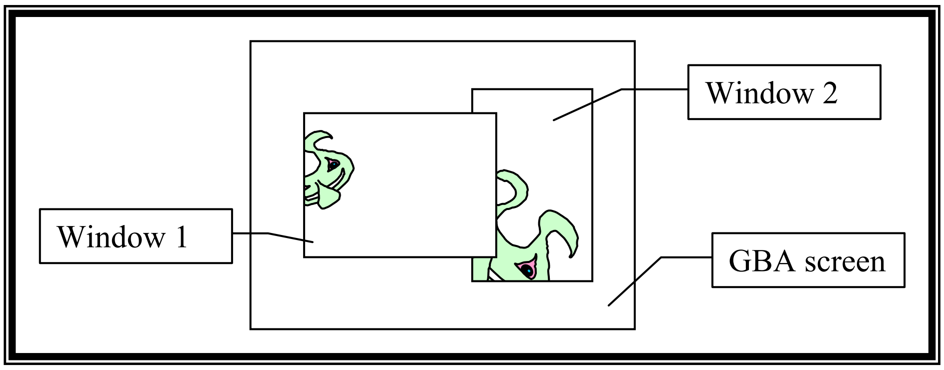


Figure 9: Hardware controlled windows on the GBA screen.

2.2.2 Sprites

As mentioned earlier, the Game Boy® Advance has support for up to 128 sprites. Each sprite has the following properties, according to Jonathan S. Harbour, [HARBOUR1, p. 230]:

- **Tile Index**, which is a reference to the sprite image.

- **Size**, indicating one of twelve possible sprite sizes in the range of 8*8 to 64*64 pixels; see the table in Figure 10 below.
- **Position** on the screen
- **Palette Information**, indicating 4 or 8 bits per pixel along with a reference to a palette of colours. Each sprite either uses a 256 colour palette or one of 16 palettes with 16 colours each.
- **Mosaic Effect**, used to decrease the sprite resolution while retaining its size. For an illustration of this, please take a look at Figure 11 below. This effect is primarily used as a "cool" feature.
- **Flip**, used to flip the sprite image left/right, up/down or both.
- **Rotation**, used to rotate the sprite in any angle.
- **Scaling**, used to scale the sprite up or down.

Shape (attribute 0)	Form (attribute 1)	Resulting sprite size (in pixels)
00	00	8 * 8
00	01	16 * 16
00	10	32 * 32
00	11	64 * 64
01	00	16 * 8
01	01	32 * 8
01	10	32 * 16
01	11	64 * 32
10	00	8 * 16
10	01	8 * 32
10	10	16 * 32
10	11	32 * 64

Figure 10: Table showing the possible sprite shapes and forms.

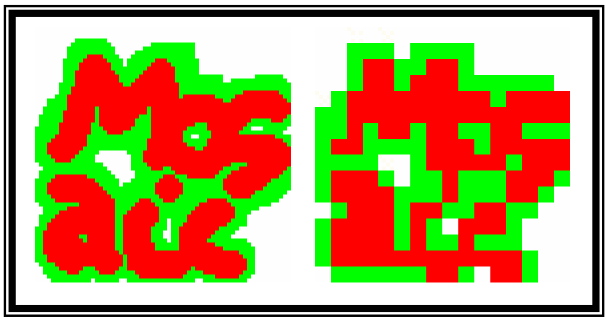


Figure 11: Two sprites with the only difference being that the Mosaic Effect has been applied to the rightmost one. Note that the size of the sprites is the same, even though the shown resolution is not.

To use these attributes they must be cramped into eight bytes per sprite, but unfortunately [HARBOUR1] does not specifically mention how this is done. An overview of this, however, is made by e.g. [GBATEK], which is a thorough specification of GBA hardware. The following is a short, yet highly paraphrased summary about its section about sprites.

Each sprite uses 3 16 bit words, usually referred to as attribute0 through attribute2. The layout of these attributes is as follows:

Attribute 0:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shape		C	M	Alpha		SD	R	Y-coordinate							

- **Shape** can be either square (= 00), wide (= 01) or tall (= 10). The last bit combination (= 11) is undefined.
- **C** determins the type of palette used. If C is 0, one of 16 16-colour palettes is used, depending on the value of bits 12-15 in attribute 2. If C is 1, the sprite uses a 256 colour palette.
- **M** is a mosaic flag.
- **Alpha** is used to set alpha blending
- **SD** is used to make the sprite double in size. This is useful when rotating a sprite, as it might otherwise be cut at the edges.
- **R** is used to set sprite rotation.
- **Y-coordinate** is used to set the vertical position of the sprite.

Attribute 1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Form		Rot. data index					X-coordinate								
		VF	HF												

- **Form** is used together with **Shape** of attribute 0 to decide the proportions of the sprite.
- If the **R** flag of attribute 0 is set, **Rot. data index** is used to rotate the sprite at any angle. Otherwise **VF** and **HF** is used to mirror the sprite along a vertical and horizontal axis respectively, causing the sprite to “flip”.
- **X-coordinate** is used to set the vertical position of the sprite.

Attribute 2:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Palette number				Priority		Sprite graphics index									

- **Palette number** is used to choose which 16 colour palette to use, when the 16 colour palette mode is chosen by setting the **C** of attribute 0 to 0. Otherwise the four bits are unused.
- **Priority** determines which sprites should be in the front and which should be in the back of each other and of the backgrounds in the tile-based video modes.
- **Sprite graphics index** is used to chose where in memory the sprite image is located, i.e. which sprite image to use.

2.2.3 User Input

User input is handled through an eight way directional pad and six buttons. The eight way directional pad has the shape of a + sign, and functions much like four buttons, where the diagonal directions are achieved when combining two non-diagonal directions. Of the six buttons, two are typically used in game menus between game play or to interrupt game play, namely the Start and Select buttons. Another two, the A and B buttons, typically control the most used functions in a game. E.g. in a shoot'em up game, one or both of these buttons would typically be used to fire a shot. The last two buttons, the so-called shoulder buttons, are named L and R (for "left" and "right"), and in games that use them, they typically control the more specialized functions of the game. The layout of the buttons can be seen in Figure 12, below.

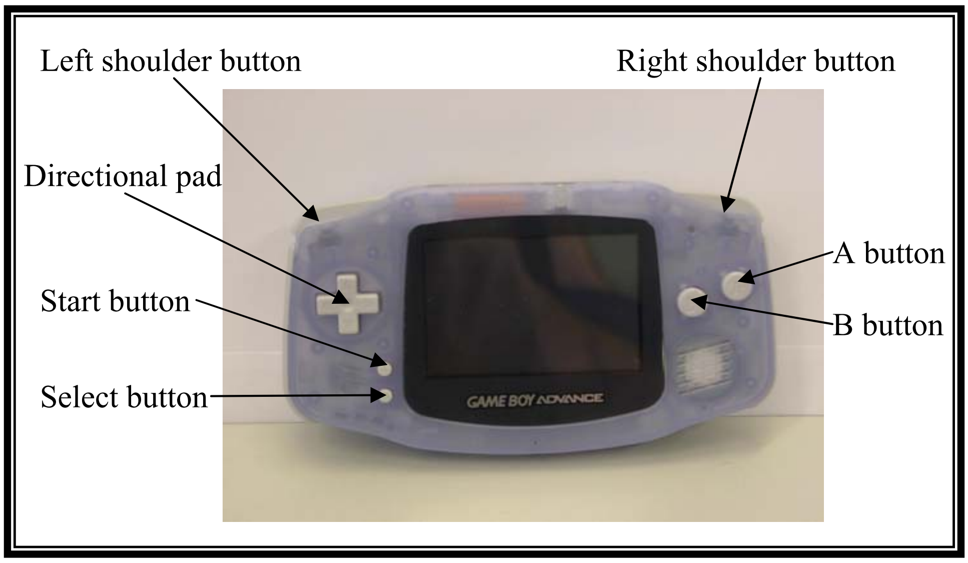


Figure 12: The buttons of the GBA

Access to reading the values of the buttons happen through a 16 bit word at a particular location in the memory, where each button is represented by one bit.

2.2.4 Game Cartridges

The games for the GBA come on game cartridges that are to be inserted in the GBA and typically have 4 or more megabytes of ROM and often a small amount of RAM, typically 32 or 64 KB, for game saves. Special cartridges are available with Flash RAM in stead of ROM, which makes it possible for non-licensed programmers to develop games etc. for the GBA. As these cartridges can also be used to make pirate copies of games, they are obviously not endorsed by Nintendo®. That kind of cartridge will be discussed in more detail in section 2.4.2.2.

2.2.5 Timers, interrupts and Direct Memory Access

The GBA is equipped with much of the same hardware as many other computers. This includes four timers, a number of interrupts and hardware optimized Direct Memory Access (DMA). These kinds of hardware are the topics of this section, and will be described below.

2.2.5.1 Timers

The Game Boy® Advance has four built in timers, each represented by a 16-bit value in memory. At every given time period (frequency), the value of each timer will increase by one, hence referred to as a tick. If so set, a counter can be made dependant on another, and only count up, when that other timer has an overflow, thus making timers of greater length than 16-bit possible. Each timer can, individually of the others, be set to one of the frequencies shown in the table below in Figure 13. In my sources, however, there seems to be some disagreement as to the exact frequency of the GBA clock, although the variation is so small, that it practically makes no difference. The sources all state that the frequency is 16.78 MHz, but [HARBOUR1, p. 306], among others, take that to mean 16,780,000 Hz, while [GBATEK] states that it means 16 * 1,024 * 1,024 = 16,777,216 Hz. Obviously, this little dispute could be settled simply by making a program to measure the exact frequency. This, however, is beyond the scope of this report, so instead I will use the latter definition in the table below and henceforward.

Frequency number	Frequency	Ticks per second	Interval between ticks	Overflow frequency (time between every overflow)
0	Every clock cycle	16,777,216	59.605 ns	256 times per second
1	Every 64 clock cycles	262,144	3.815 µs	4 times per seconds
2	Every 256 clock cycles	65,536	15.259 µs	Every 1 second
3	Every 1024 clock cycles	16,384	61.035 µs	Every 4 seconds

Figure 13: Timer frequencies. 1 µs = 1 microsecond = 10^{-6} second. 1 ns = 1 nanosecond = 10^{-9} second.

2.2.5.2 Interrupts

The Game Boy® Advance is capable of handling several interrupts, which means that every time a specific event occurs, normal program execution is interrupted, and a jump is made to an address, which must be specified beforehand. The events that can cause an interrupt include:

- **Vertical Blank**, which occurs every time the screen has been refreshed.
- **Horizontal Blank**, which occurs every time a line on the screen has been refreshed.
- **Vertical Scanline Count**, which occurs when a specified scanline is being drawn
- **Timer overflow**, which occurs when one of the four timers overflow.
- **Serial Communication**, which is used for communication with other GBA units via the so-called game port.
- **DMA finished**, which occurs when one of the four DMAs has finished. The DMAs are discussed in section 2.2.5.3 below.
- **Button pressed**, which occurs, when one of a specified range of buttons is pressed.
- **Cartridge**, which occurs, when a game cartridge is inserted or removed.

I will not go into detail with every one of these events, yet the Vertical Blank, deserves some study, as it is the key to getting a correct timing in games. As mentioned, the vertical blank interrupt is generated each time the screen has been refreshed, which is approximately 59.73 times a second. The period of time between the end of a screen refresh and the beginning of the next screen refresh is called the vblank period and is equal to 83,776 clock cycles. The refresh rate, which is the period between the start of two consecutive screen refreshes, is equal to 280,896 clock cycles [HARBOUR, p. 305]. There are two main reasons why the vertical blank interrupt is important in game programming:

- It can be used to ensure a consistent frame rate during game play.

- It can be used to ensure that the graphics are displayed properly, i.e. without "tearing", that can occur when e.g. a sprite is moved while being drawn. An illustration of the undesired tearing effect is shown in Figure 14, below.

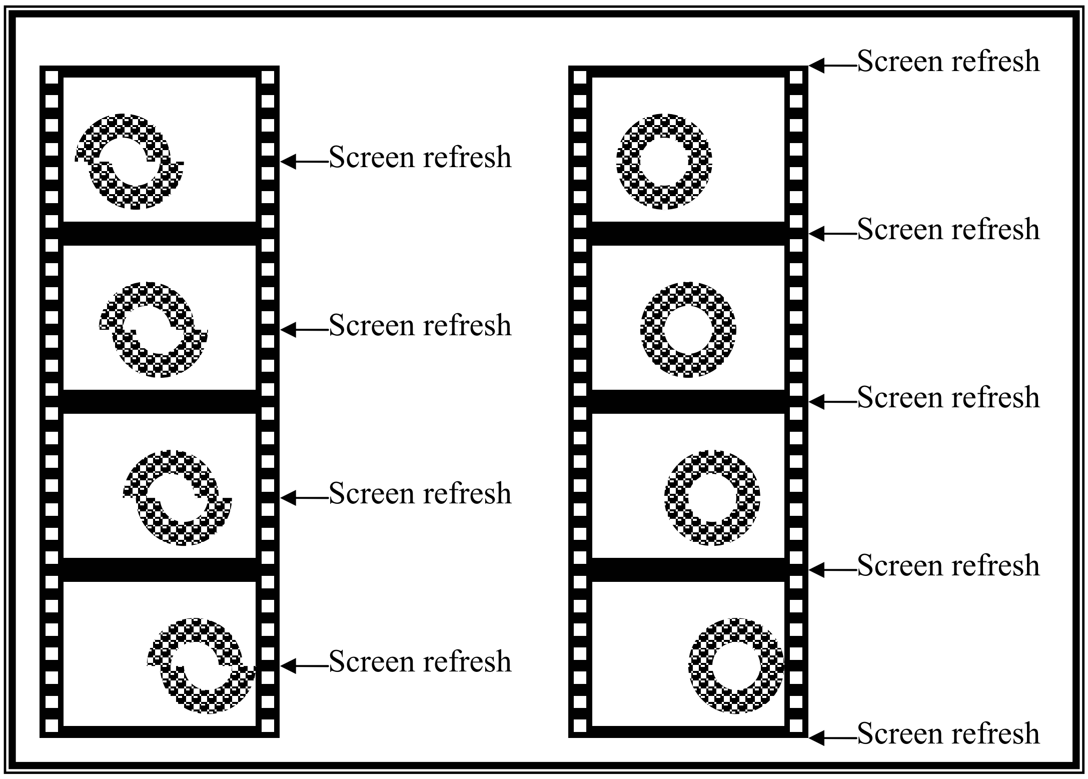


Figure 14: The left reel illustrates the tearing effect that can occur when a sprite is moved in the middle of a screen refresh. Please notice that the upper part of the sprite appears to be disconnected from the lower part. The right reel illustrates the same sprite movement, but here care has been taken only to move the sprite between screen refreshes. Please notice that the sprite is not disconnected.

All interrupts will generate a call to the same function, which is then responsible for taking the proper actions, depending on which event caused the interrupt, thus making it necessary to make an interrupt handler. The way to handle interrupts is described in [HARBOUR, pp. 299-305] and is here boiled down to the following necessary steps:

- Interrupts must be disabled, in order to keep further interrupts from disrupting the handling of the current one.
- Certain interrupt flags must be backed up in order to restore them later, as they might otherwise be changed, possibly resulting in a system crash.
- Actions must be taken, depending on which event caused the interrupt.
- The saved flags must be restored.
- Interrupts must be re-enabled.

During the handling of an interrupt, further interrupts are usually disabled. Otherwise, if interrupts were generated at a faster rate than the handling of them, it could lead to a stack overflow, as a return address is stored at every interrupt, and eventually cause a system crash.

2.2.5.3 Direct Memory Access (DMA)

The GBA is equipped with four Direct Memory Access (DMA) channels, which are hardware supported ways of copying an area of memory from one address to another. Obviously, using hardware supported means of copying is much faster than a loop doing the same thing – even when written in assembler. Speed, however, is not the only reason to use the DMA channels. Other reasons are timing, as it is possible to set up the DMA channels to copy memory according to one of the timers, such that e.g. one word is copied every given interval. Another timing aspect is the fact that the DMA channel can be set up to wait for the next Vertical Blank to occur before starting copying, which is useful, when using the DMA to change the graphics on the screen. Also, the DMA channels can be set up to using the same destination address for all data being copied. At first this might seem useless, but in combination with a timer, this is exactly what is needed when playing digital sound, as playing digital sound is done by changing the value of certain memory addresses at given time intervals, thus determining the frequency of the sound being played.

2.2.6 Sound

The GBA has six sound channels, of which four are primarily present for backward compatibility reasons with the Game Boy® Color® and the original Game Boy®. The sound quality of these four channels is not the best, which is why the GBA has an extra two sound channels, capable of handling digital sound, or sampled sound. Each of these two channels is composed of a Digital-to-Analog Converter (DAC), and can thus play digital sound samples, as mentioned in [HARBOUR, p 333].

The way to generate sound on the GBA is to send wave samples to one of the two DACs at a given interval, depending of the frequency of which the sound should be played, which is usually the same frequency at which the sound was originally sampled. The DACs are available through a specific memory address, to which the sound samples must be written. As mentioned in section 2.2.5.3 about DMAs, a DMA can be made to copy one byte after another to the same address in memory at intervals defined by the overflow of a given timer.

However, the timers are dependant on clock frequency, rather than of sound sample frequency, so the number of clock cycles between each sample must be calculated, and used as the timer's initial value, which it must start with after every overflow. As the timers overflow at a value of 65536, that initial value can be calculated with the following formulae:

$$65536 - \frac{clock\ frequency}{sample\ frequency}$$

The clock frequency is a constant at 16777216, which means that if, for instance, the sound to be played has a sample rate, or frequency, of 44000, the number of clock cycles between each sample is

$$\frac{16777216}{44000} \approx 381$$

which means that the initial value of the timer must be

$$65536 - 381 = 65155$$

After that, only one problem remains, and that is to stop the DMA from playing the sound, when it has finished. According to [HARBOUR], the way to set up the DMA and the timer, will cause the DMA to continue playing the sound until it is stopped manually, and thus it is necessary to calculate the number of screen refreshes a sound should be played, and when that number has passed, stop the sound manually.

2.2.7 Communication

Up to four Game Boy® Advance units can be joined through a so-called Game Link Cable, which is connected to the External Extension Connector on the top of the GBA as shown in Figure 15, below.

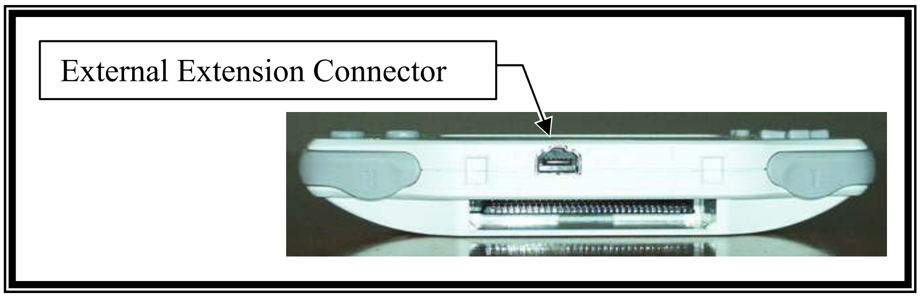


Figure 15: The top side of a GBA unit – showing the External Extension Connector.

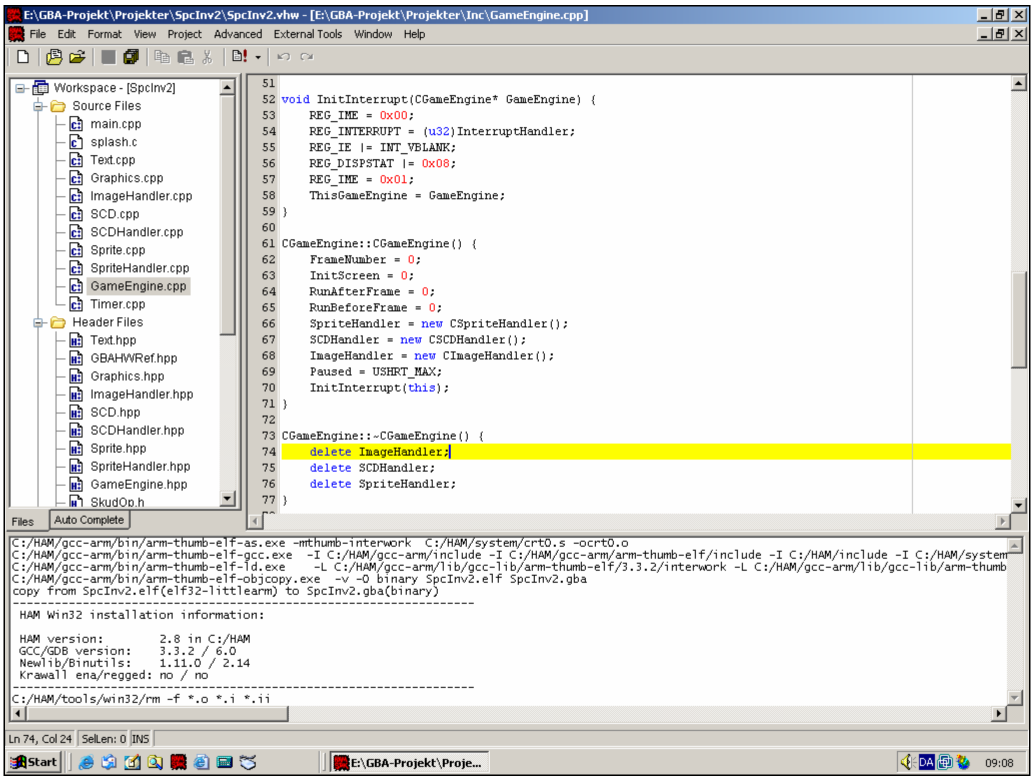
Games can send and receive data through the Game Link Cable, which makes it possible for up to four players to participate in the same game simultaneously, provided they each have a GBA unit and a suitable Game Link Cable.

2.3 Development systems

A number of GBA development systems, i.e. a system that can compile and link source code of some programming language to make it executable on the Game Boy® Advance, have been developed and made available to the public. These systems are not necessarily Public Domain, but are, nevertheless, systems that do not require a special agreement with Nintendo®, such as a non-disclosure agreement. So far, the systems I have come across can be used to develop Game Boy® Advance programs with programming languages such as ARM assembler, C, C++ and a variant of Basic. Also commercial systems are available, e.g. the "Catapult Development System", which "includes an editor, debugger, simulator and an extensive set of tools [and] is based around a custom language", according to [NOCTURNAL]. Links to these systems can be found at [GBADEV]. Two of the more interesting of the systems mentioned above are DevKit Advance and HAM. Both include C/C++ compilers and are discussed briefly below:

2.3.1 Visual HAM

HAM is a development system that comes integrated in a graphical shell called Visual HAM. It is a complete development environment that runs on a PC, either under Windows or Linux. It includes compiler, linker, editor, debugger, GBA emulator, a number of conversion tools for converting graphics and sound files to GBA compatible C/C++-arrays, as well as the so-called library HAMlib, which is a large collection of functions and macros particularly suitable for game development on the GBA. As HAM is quite easy to install, it makes a good choice for the beginning Game Boy® Advance developer. The appearance of Visual HAM bears a resemblance to Microsoft® Visual Studio® and can be seen in Figure 16, below.



```

void InitInterrupt(CGameEngine* GameEngine) {
    REG_IME = 0x00;
    REG_INTERRUPT = (u32)InterruptHandler;
    REG_IE |= INT_VBLANK;
    REG_DISPSTAT |= 0x08;
    REG_IME = 0x01;
    ThisGameEngine = GameEngine;
}

CGameEngine::CGameEngine() {
    FrameNumber = 0;
    InitScreen = 0;
    RunAfterFrame = 0;
    RunBeforeFrame = 0;
    SpriteHandler = new CSpriteHandler();
    SCDHandler = new CSCDHandler();
    ImageHandler = new CImageHandler();
    Paused = USHRT_MAX;
    InitInterrupt(this);
}

CGameEngine::~CGameEngine() {
    delete ImageHandler;
    delete SCDHandler;
    delete SpriteHandler;
}
```

```
C:/HAM/gcc-arm/bin/arm-thumb-elf-as.exe -mthumb-interwork  C:/HAM/system/crt0.s -ocrt0.o
C:/HAM/gcc-arm/bin/arm-thumb-elf-gcc.exe  -I C:/HAM/gcc-arm/include -I C:/HAM/gcc-arm/arm-thumb-elf/include -I C:/HAM/include -I C:/HAM/system
C:/HAM/gcc-arm/bin/arm-thumb-elf-ld.exe    -L C:/HAM/gcc-arm/lib/gcc-lib/arm-thumb-elf/3.3.2/interwork -L C:/HAM/gcc-arm/lib/gcc-lib/arm-thumb
C:/HAM/gcc-arm/bin/arm-thumb-elf-objcopy.exe  -v -O binary SpcInv2.elf SpcInv2.gba
copy from SpcInv2.elf(elf32-littlearm) to SpcInv2.gba(binary)
-----------------------------------------------------------------------
 HAM Win32 installation information:

 HAM version:        2.8 in C:/HAM
 GCC/GDB version:    3.3.2 / 6.0
 Newlib/Binutils:    1.11.0 / 2.14
 Krawall ena/regged: no / no
-----------------------------------------------------------------------
C:/HAM/tools/win32/rm -f *.o *.i *.ii
```

Figure 16: The appearance of VisualHAM. The window is divided in three. At the bottom is shown some compiling information. At the top left is shown a list of files, that together make up a project. At the top right is a text editor.

HAM along with VisualHAM can be downloaded from [NGINE].

2.3.2 DevKit Advance

DevKit Advance is a command line based compiler, and as such it lacks the usability of VisualHAM. It is, however, more flexible in regards of using it with a third party environment, such as, for instance, Microsoft's Visual Studio. In fact, various guides describe how to set up DevKit Advance to work with Microsoft® Visual Studio®. These can be found on the Internet, e.g. at [GBADEV1]. DevKit Advance can be downloaded at [SF1].

2.4 Playing the game

When compiling a game or any other program, for that matter, for the GBA, the result is a single file. Since games for the GBA are meant to be sold on cartridges, as mentioned in section 2.2.4 about Game Cartridges, the GBA does not directly provide means for receiving such a file from a PC. Thus, special means are required to play a home made game. The simplest way is to use an emulator, which can often suffice for testing purposes. However, there is not much point in making a game for a Game Boy® Advance, if it is only to be played in an emulator. Thus, means for playing the game on a real Game Boy® Advance are also necessary. Both ways are described in the following:

2.4.1 Emulation

Several Game Boy® Advance emulators are available. For Windows alone, [ZOPHAR1] lists a whole 22. I will not mention them all here, but instead bring up two that stand out from the rest.

2.4.1.1 Visual Boy Advance

As far as I can judge, having tried some emulators myself, and read what others have written on the subject, the best Game Boy® Advance emulator is Visual Boy Advance. Not only can it play the games almost perfectly, it also comes with a lot of debugging facilities, such as a disassembler, logging, viewing of about 100 control registers, graphics viewers (maps, sprites, palette and tiles) and memory viewer. What's more, it allows for single stepping one frame at a time, as well as recording of audio and video. The Visual Boy Advance can be downloaded from [NGEMU], and an image of it in function can be seen below, in Figure 17.

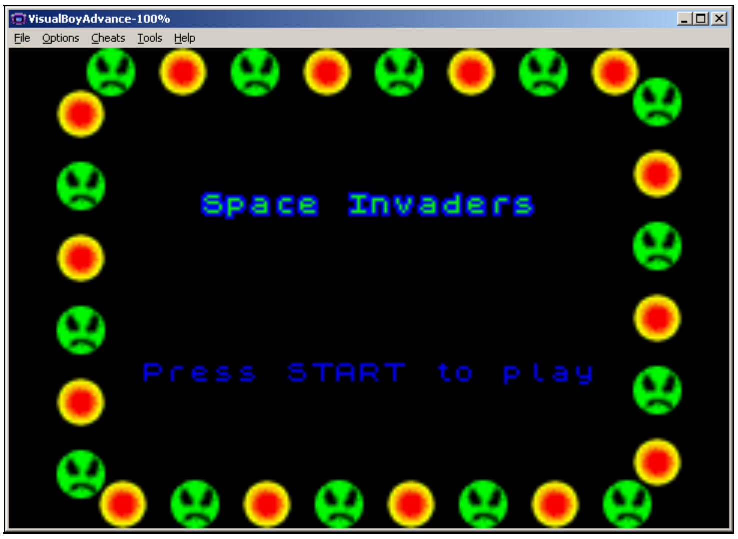


Figure 17: Screen dump of VisualBoyAdvance

2.4.1.2 No$GBA

One thing is missing from the Visual Boy Advance emulator, though, and that is simulation of several GBA units connected via a Game Link Cable. The only GBA emulator, where I have noticed this feature, is No$GBA. However, even though No$GBA can play games almost flawlessly, the freeware version of No$GBA is next to useless at aiding programming, as all debugging facilities has been removed from it. The freeware version of No$GBA can be downloaded at [EMUBASE]. The full version of No$GBA includes a number of debugging facilities, but as it is rather expensive, I have not had the opportunity to try it myself and thus I cannot comment on the quality. An image of the No$GBA emulator is shown below, in Figure 18.

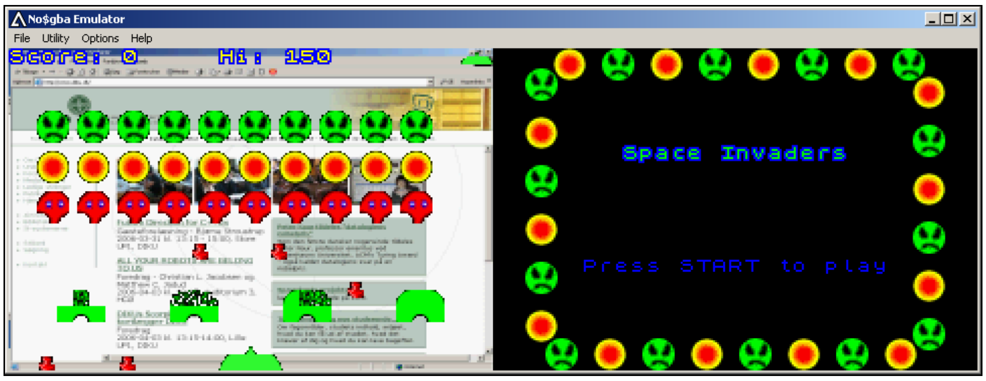


Figure 18: Screen dump of the No$GBA emulator, emulating two GBA units simultaneously.

2.4.2 Transferring the game to the Game Boy® Advance

There are two ways to go about when wanting to try a game on the real hardware; either via a cable connecting the GBA and a PC or by using a special Game Cartridge. Both methods are described below.

2.4.2.1 Multi Boot Version 2 Cable

As mentioned earlier in section 2.2.6, the Game Boy® Advance can communicate with other GBA units by use of the External Extension Connector port. But, that port can be used to more than merely sending and receiving game information from one GBA unit to the other. If a GBA is turned on without an inserted Game Cartridge, it will try to receive a program via the External Extension Connector from another GBA unit. The purpose of this is that it makes it possible to play some two or even four player games on separate GBA units without having to buy two copies of the same Game Cartridge. However, this piece of functionality can also be used in game development, as the GBA does not know from where it receives its program. The MBV2 (Multi Boot Version 2) takes advantage of this fact by making a PC send the boot program to a GBA. In fact, the MBV2 cable is even supported by Visual HAM, in that the program being developed can be compiled and sent to a GBA by the press of a single key. Below is an image from [LIK-SANG] of a MBV2 cable.

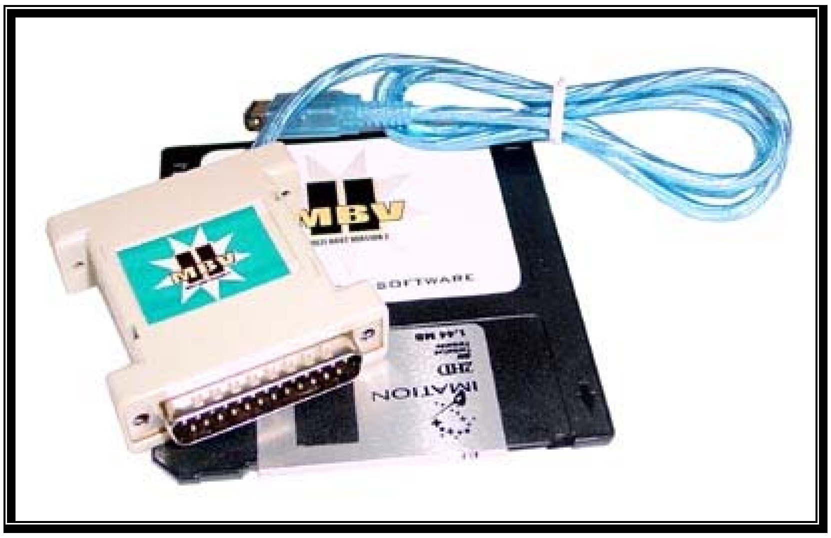


Figure 19: Picture of the Multi Boot Version 2 cable

The disadvantage of the MBV2 cable is that the whole program must be placed in RAM, which only allows for programs at the size of up to 256 KB to be transferred to the GBA. Also, with the MBV2 cable, a near by PC is necessary, every time a game is to be played.

2.4.2.2 Flash linker

As mentioned earlier in section 2.2.4 about Game Cartridges, special cartridges are made, that instead of ROM has Flash RAM. These cartridges usually come with a card reader/writer that can be connected to a PC, and games can then be transferred to the cartridge with the enclosed software. One of these Flash Ram cartridges, the EZ-Flash II, can be seen in Figure 20 below, together with the card reader/writer, the USB cable to connect the latter to a PC and the box it game in.

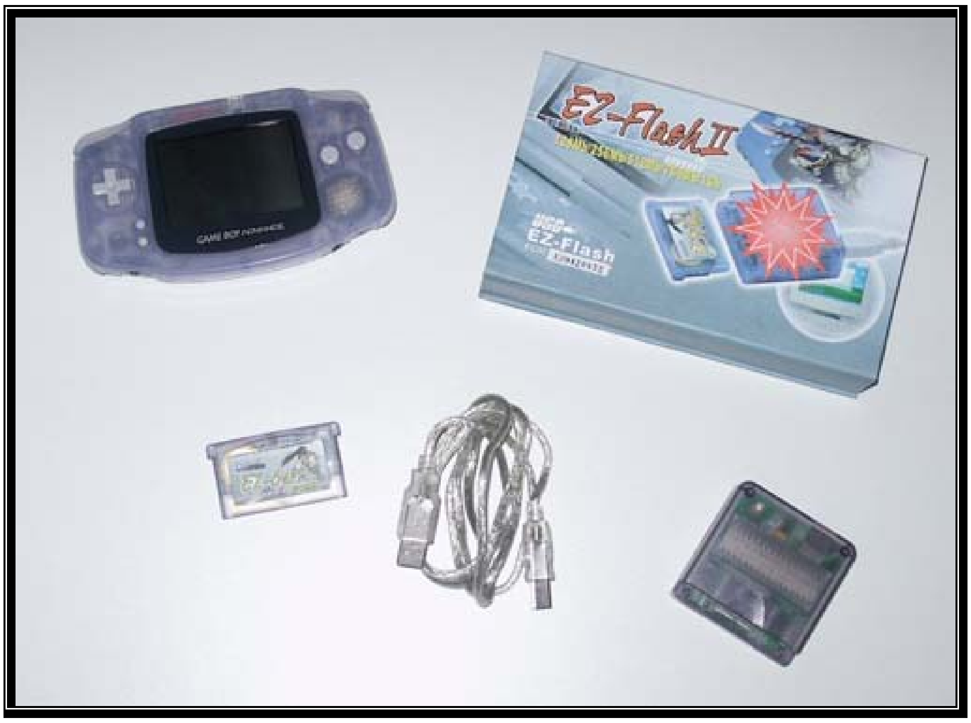


Figure 20: The EZ-Flash II Flash Cartridge and a Card reader/writer together with a GBA unit.

3 Game Engines

Most games include one or more of the same elements such as sound, sprites, a sprite handler, background images, a graphics library and handling of user input as well as means to ensure a consistent frame rate. This means that a lot of programming code is potentially identical from one game to another and thus, in stead of writing programming code that can take care of all the elements stated above every time a new game is developed, that code can be collected in a so called game engine.

However, different categories of games will emphasize different aspects of game playing and thus necessitate focus on different elements in a game engine. Also, different kinds of hardware will set the scene for different approaches when designing a game. For example, most games that are made for both the PC and the GBA, and which appear as 3D games on the PC, will only appear as 2D games on the GBA due to the lack of 3D accelerated capabilities on the GBA.

In this report, however, I will not emphasize one game category over another, but rather give an all round presentation of what a game engine does and why it is beneficial to have one when making games. On the other hand, as this report is about a game engine for the GBA, the primary focus will be on the elements that the GBA can handle.

In the remainder of this chapter I will try to give an account of what a general game engine consists of, and on occasion offer some considerations about choosing what to implement when making a game engine.

A game engine is responsible for maintaining a constant flow in the game. What I mean by this is that the game engine must make sure that certain things happen at every frame of the game including the handling of user input and the updating the screen. The game engine should make sure that the graphics move as smoothly as possible, which makes it important only to update the screen in the so-called Vertical Blank periods, which are the periods of time ranging from when one screen refresh has finished until the next screen refresh begins. Otherwise, if graphics are updated at the wrong time, the result could be that the graphics appear to be torn apart, e.g. that the upper part of a sprite appears to fall behind the lower part. This undesired effect of tearing is illustrated earlier in Figure 14 on page 17.

Conceptually, a game engine can be thought of as an operating system that is specialized towards the execution of games. As with ordinary operating systems, such as Microsoft® Windows® and Unix®, one of the more important aspects of a game engine is that it functions as a virtual layer between the hardware and the game program. This, however, is typically more important on multiple purpose computers such as a PC or a main frame than on game consoles such as a GBA. The reason for this is, that the hardware of multiple purpose computers usually differ from each other, in terms of different processors, graphics capabilities, available amount of RAM and so on, making it difficult to make programs that take advantage of the computer's hardware directly, whereas game consoles typically are very close to being identical to others of the same model. Furthermore, as game consoles are mostly used to play games, and as many games are time critical, adding an extra layer between the hardware and the game should be considered carefully. On the other hand, the advantages of including this extra layer on a game console are not to be disregarded. They include the following.

There are three main purposes for making a game engine:

- **Reusable code** – code that is identical in several games need only be made once, when it is part of the game engine.
- It **reduces the need for hardware specific knowledge**, such as how to directly handle interrupts, sprites and user input etc.
- It can help make the source code for the game more **clear and well-arranged**, as focus can be on game specific code rather than on hardware specific code.
- Clear and well-arranged code will make it simpler for different game programmers to read and understand each other's code, thus making it easier for more game programmers to participate in the same project.
- It can **save development time**, as a consequence of the points above.

As with ordinary operating systems, such as Microsoft® Windows® and Unix®, the game engine must take care of the most trivial tasks, of which other programs can depend, such as controlling user input and output to the screen. In the case of a game console like the GBA, this is especially true, as even simple tasks that are used in most games, e.g. displaying text, has to be programmed from scratch.

However, opposed to the ordinary operating systems, a game engine is typically distributed as an integrated part of a game. This makes it easier to use a different game engine for different types of games or even individual games, and thus also to specialize a game engine towards a specific game or game genre, which is important in itself, but more importantly, though, is the fact that this allows for much easier installation with regards to the end user, as games on game consoles typically only need to be inserted in order to be executed.

3.1 Content Management

Different kinds of data are part of many games; data such as sound data, images for sprites and backgrounds, data that make up rooms and/or levels and in the case of 3D games, data that make up characters and surroundings etc. etc. All that data can make a mess of programming a game, and thus, in this section is mentioned a structured way of handling data in a so-called Content Management System (CMS). In case of game programming, a CMS is a system that can handle all the mentioned game data in an organized way. One of the main benefits from this is that it allows for the programming of the game to be separated from designing characters, backgrounds, levels and rooms etc.

General data, such as sprite images, background images and sound data would be rather simple to make a content management system for, since the structure of that kind of data is the same regardless of the game. Data to make up rooms and levels of a game, on the other hand, would be more difficult to make a general system for, since the structure of these kinds of data much more depends on the actual game, and it is thus equally much harder to make a general content management system that can handle that.

In the next section I will go through the system Game Maker, which is an example of a game specific CMS which is furthermore combined with a game engine to make a complete game development system

3.1.1 A combined Game Engine and Content Management System

An example of how a content management system and a game engine can be combined to form a complete game development system is the subject of this section. Here I will briefly go through the game development system Game Maker, written by Mark Overmars. It can be found on the Game Maker web site [GAMEMAKER]. A screen shot of a game made with Game Maker is shown below in Figure 21.

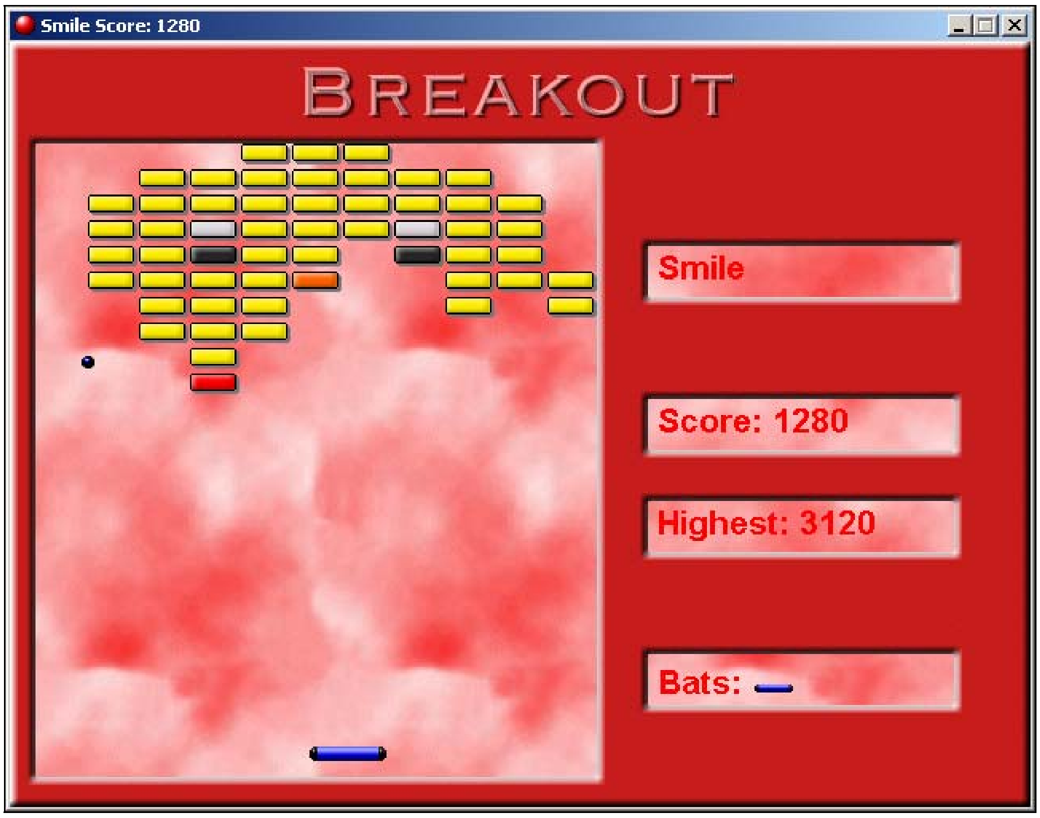


Figure 21: A scene from the game Breakout, which is included in the distribution of Game Maker.

A screen shot showing what Game maker looks like, when making a game, can be seen in Figure 22, below.

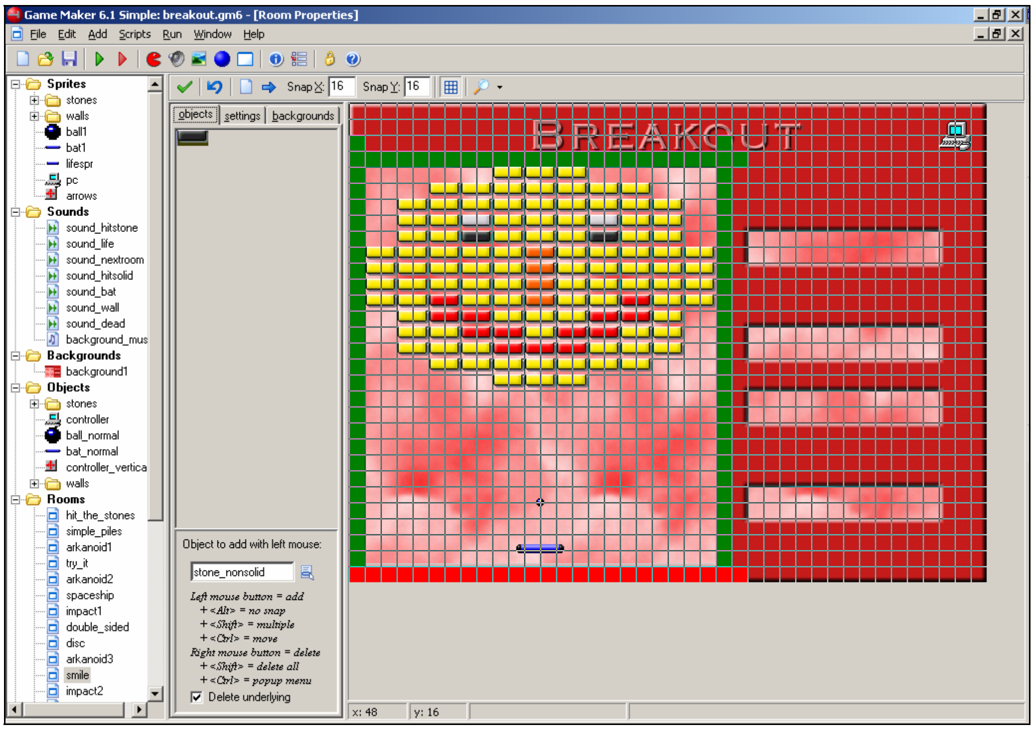


Figure 22: A screen shot from Game Maker, where a so-called room is shown.

At the left of the screen shot in Figure 22 is shown some of the types of data that make up Game Maker. These types of data are:

- **Sprites**. A sprite in Game Maker is merely an image or a sequence of images. These images can be loaded into game maker and edited.
- **Sounds**. A sound is just what it says, a sound.
- **Backgrounds**. A background is an image that can be used shown in the background of a game.
- **Objects**. An object is a sprite in the normal sense, meaning that it is an image with certain attributes. An object can be assigned certain events, and each of these events can then be assigned one or more actions. That way, when a certain event occurs, the proper actions are performed.
- **Rooms**. A room is where the scene is set. A room is a collection of objects that make up one of the scenes of the game. Many games include several rooms that define different places in a game world or different levels of the game.

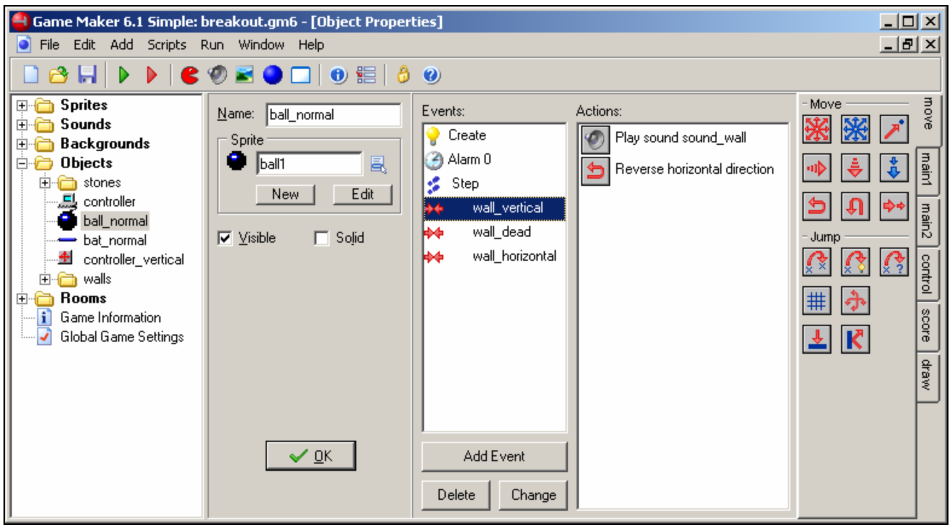
Game Maker has two different modes; a simple mode with the data types mentioned above and an advanced mode with additional data types. For the purpose of this section, the simple data types will suffice, however, for the sake of completeness, the advanced data types are mentioned below.

- **Paths**. A path is a collection of curves that can be defined. Objects can be made to follow a path.
- **Scripts**. A script is a fragment of code, which can be inserted in a game in order to perform a specific task.
- **Fonts**. A font is just what it sounds like. The default font in Game Maker is 12 points Arial. If other fonts are desired, they have to be defined.

- **Time Lines**. A time line is a group of actions that should be performed at specific moments in time.

Game Maker is event controlled which means that certain events will trigger one or more defined actions. Game Maker includes 12 events such as collision with other objects, mouse clicks and keyboard presses. It also includes numerous actions that can be carried out, when these events occur. Actions are very numerous and include movement of an object in a certain direction, towards a specific point or along a predefined path, as well as creation and destruction of objects and playing sounds etc. Furthermore some of the actions are if-then-else constructions and some define the start and end of a block of actions, thus allowing for a more advanced program flow.

Making games with Game Maker is done through a graphical user interface by moving icons around. Figure 23 shows the properties of an object from the example game Breakout, which is included in the distribution of Game Maker. These properties include several events as well as the actions that are performed when one of these events occur.



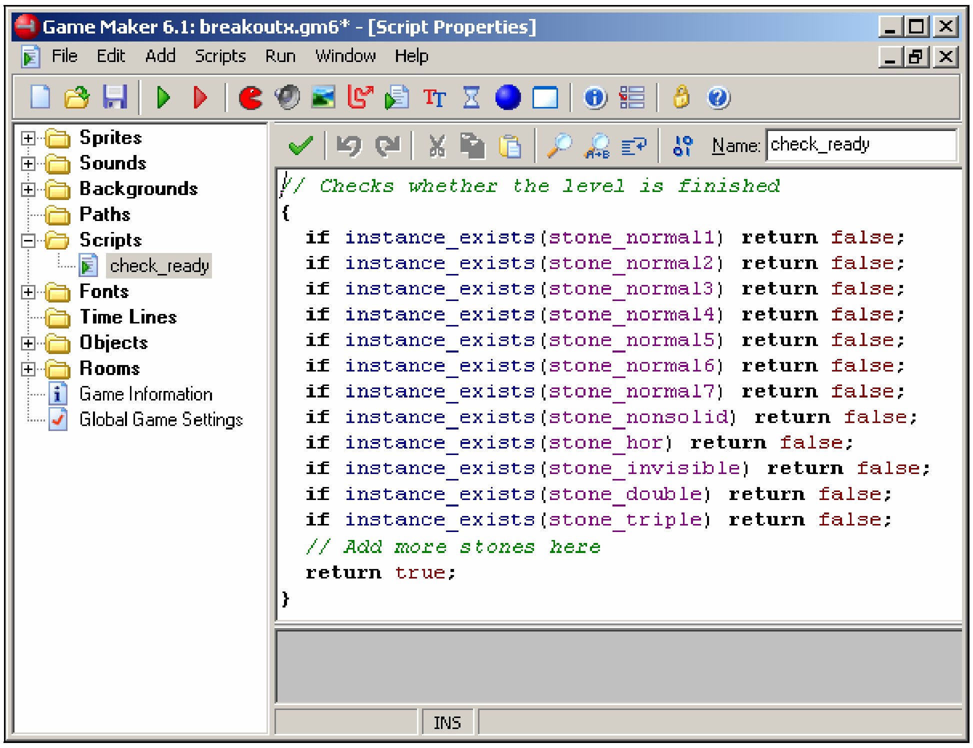
Name: ball_normal
Sprite: ball1
☑ Visible ☐ Solid

Events:
Create
Alarm 0
Step
wall_vertical
wall_dead
wall_horizontal

Actions:
Play sound sound_wall
Reverse horizontal direction

Figure 23: Screen shot from Game Maker, where an object can be seen along with its assigned events and actions.

If more specific actions are needed than the ones included in Game maker, the user can resort to the built-in scripting language. Here, small fragments of programs can be made, to perform the specific actions needed, and Game Maker keeps track of it all. The script editor, which is only available in the advanced mode of Game maker, can be seen in Figure 24, below.



```
// Checks whether the level is finished
{
  if instance_exists(stone_normal1) return false;
  if instance_exists(stone_normal2) return false;
  if instance_exists(stone_normal3) return false;
  if instance_exists(stone_normal4) return false;
  if instance_exists(stone_normal5) return false;
  if instance_exists(stone_normal6) return false;
  if instance_exists(stone_normal7) return false;
  if instance_exists(stone_nonsolid) return false;
  if instance_exists(stone_hor) return false;
  if instance_exists(stone_invisible) return false;
  if instance_exists(stone_double) return false;
  if instance_exists(stone_triple) return false;
  // Add more stones here
  return true;
}
```

Figure 24: A screen shot showing the script editor of Game Maker.

An illustration of the data structure of games made with Game Maker is shown in Figure 25, below.

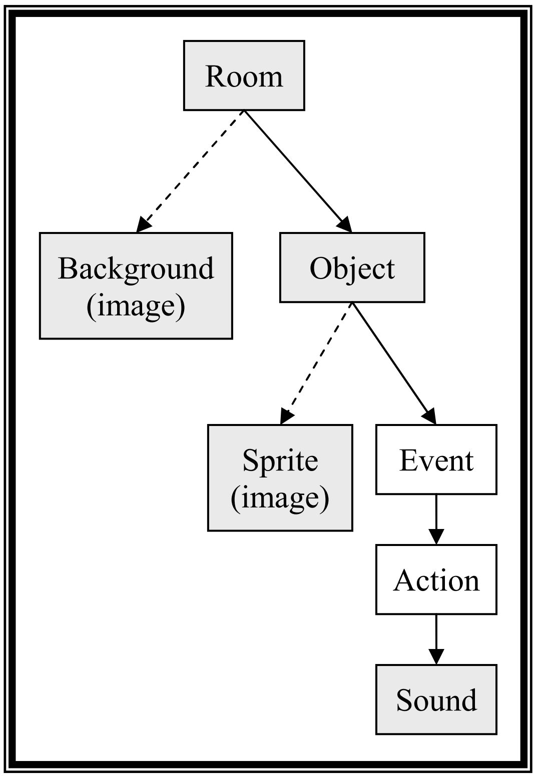


Figure 25: The data structure of a game in Game Maker.

The grey boxes indicate data types and the white boxes indicate features that may vary from one instance to another. For example, all instances of the same object have the same properties each time they are added to a room, whereas an event may have different properties each time it is added to an object, even though it is the same event. The dashed arrows indicate that each room and each object can only be assigned one image.

The figure should be read this way: A game consist of a number of rooms. Each room consists of any number of objects and possibly a background image. Each object consists of any number of instances of events and possibly a sprite image. Each instance of an event consists of any number of instances of actions. One of the possible actions is to play a sound.

3.2 The full game loop

A game engine must ensure that the game proceeds as planned. What this means is that the game engine must handle user input, graphics and sound. At the same time, the game engine must ensure a consistent frame rate, which means that it should ensure that the screen, including sprites and other graphics, is updated at regular intervals. Also, user input and sound must be handled regularly, though not necessarily at the frame rate. In fact, sound must typically be handled at a sound frequency.

3.3 Sound

Many games include some sort of background music along with various sound effects. The kind of sound available is mostly dependent on the available hardware. Some computers can play digital sound, i.e. sampled sound; whereas others generate a waveform. Nowadays, computers that are interesting from a gaming perspective, usually have several sound channels that can be manipulated individually. However, many games depend on playing numerous sounds at any one time, even more sounds than there are sound channels. In that case it is necessary to mix the sounds. An ideal game engine will do that for you.

3.4 Sprites

According to [Foley, p. 180], sprites are "multiple small, fixed-size pixmaps … on top of the frame buffer". What this means is simply that a sprite is a small image, which is drawn on top of other graphics. This definition, though, is not quite adequate, as a sprite can be more than just a small image. Usually sprites can move around on the screen, and can be manipulated in other ways, and consequently Lobão states in [LOBÃO, p. 80] that a typical sprite has certain attributes, such as a bitmap, a position on the screen, a direction and a scale factor. Furthermore, [MORRISON, p. 212] states that sprite properties include position, velocity, Z-order, a bounding rectangle, used for checking when sprites collide, as described in section 3.4.1.2.1 and 3.4.1.2.2 below, bounds action "which determines how a sprite acts when it encounters a boundary" and a flag to indicate whether the sprite is hidden or visible. Lobão also declares, in [LOBÃO, pp. 80-81], that sprites have methods like new, draw and undraw, that are used to make a new instance of a sprite, draw it on the screen and remove it from the screen, respectively.

One of the attributes, the position on the screen, deserves some extra attention. Obviously, as screen positions are measured in pixels, the position is made up by two integer numbers for the x and y coordinates. However, to easily achieve a situation where one sprite moves slightly faster than another, real numbers can be used instead. When the sprite is placed on the screen, these real numbers must then be truncated to integers. Furthermore, in order to avoid using real numbers, which can slow down performance on some computers, integer numbers can be used instead. In that case, the most significant bits of the x and y coordinates will be used to represent the location on the screen, and the least significant bits will be used to represent the value after the decimal point.

Many games use animated sprites. This means that several images are used for each sprite and in that case an ideal game engine will provide simple means of alternating between the different images in order to achieve the effect of animation.

In games sprites are typically used to represent the player character, enemy characters, bullets and every other thing that moves, and thus most games use several sprites at any one time. Instead of controlling every sprite individually, most literature I have come across on the subject of game programming describe a so-called Sprite Handler, that will do the hard work regarding the handling of the sprites, while providing some easier interface towards the rest of the game program.

Also described in most of the mentioned literature, is some sort of centrally controlled way of handling the situation that occurs when sprites collide with each other. Thus, Sprite Collision Detection and a Sprite Handler will be the topics of the next sections.

3.4.1 Sprite Collision Detection

As sprites represent moving objects, as mentioned above, it is extremely useful to have a standardized method of controlling what to do when they collide. Thus, the purpose of such a method is to make it trivial to check for sprite collision, thus allowing the game programmer to concentrate his or her efforts on programming game specific details rather than general game programming. Consequently, such a method must be designed to make it trivial to check, for instance, if a shot from an enemy character has hit the player character, if the player character has hit an enemy character or anything else that might collide.

The problems relevant to Sprite Collision Detection can be divided into two sub-problems:

1. Determine which pairs of sprites must be checked for collision, the so-called Broad Phase Collision Detection.
2. Check if the two sprites in each of the above mentioned pairs collide, the so-called Narrow Phase Collision Detection.

These sub-problems are discussed in the following.

3.4.1.1 Broad Phase Collision Detection

Games with a lot of sprites can potentially demand a lot of computer power just for testing for sprite collision. If n represents the number of sprites, and every sprite must be tested for collision with every other sprite, the complexity for sprite collision detection alone is $\mathrm{O}(n^2)$, as the total number of tests made will be $\frac{n*(n-1)}{2}$, to be carried out in each frame of the game (proof omitted). Thus, with e.g. 100 sprites, the number of collision tests will be 4950 per frame. This can easily be a bottleneck in overall performance, even with a much smaller number of sprites, depending of the complexity of the test. This is the subject of the next section, 3.4.1.2, whereas a method of reducing the number of tests is the subjects below.

Rather trivially, the game engine can depend upon the game programmer to define which pairs of sprites need to be tested for collision, thereby reducing the number of tests. In a simple shoot-'em-up game, for example, it is unnecessary to perform collision tests between the sprites representing the player character and the shots he fires, as the latter will always move away from the former, in most games, anyway.

As stated above, this method is dependant on the game programmer, and since the purpose of a game engine is to make life easier for him or her, an automated method would be desired. One possible method is to sort the sprites using a grid, so that only sprites within the same field of the grid are actually tested for collision. This method, hence forward called the Gridding Method, is illustrated in Figure 26, here.

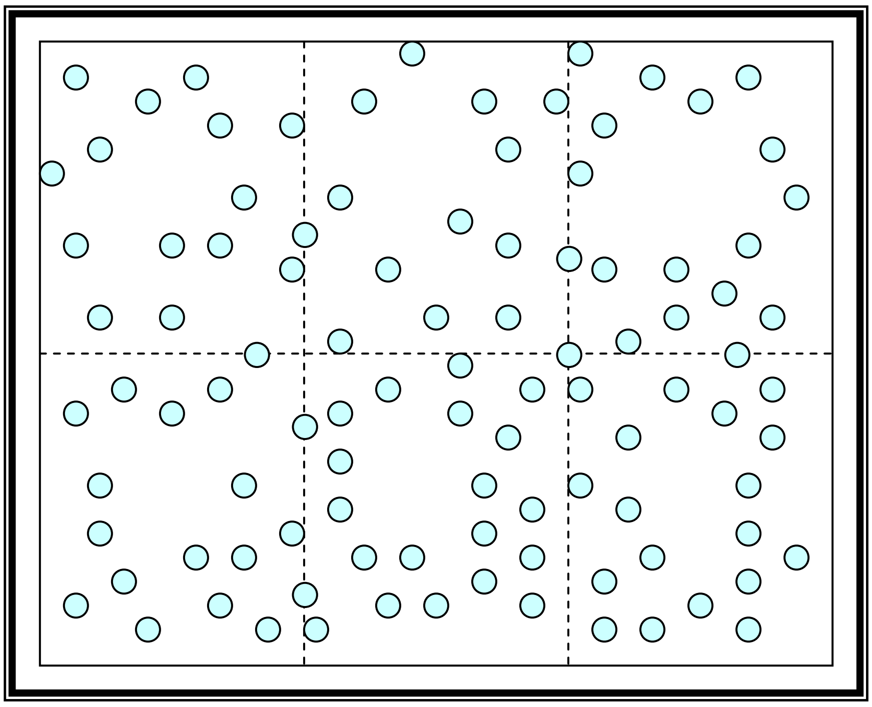


Figure 26: Illustration of the Gridding Method with a grid of six fields and 100 sprites.

The method takes advantage of the fact that often sprites are spread more or less equally across the screen. One of the reasons for this is that sprites are often used to simulate solid, physical objects, and one of the characteristics of these is that they cannot overlap. So, in a game, where sprites simulate solid, physical objects, there will be a maximum to the number of sprites that can be packed into one field of the grid.

If f signifies the number of fields in the grid, n signifies the number of sprites and n_m signifies the number of sprites in field m, instead of having to do

$$\frac{n(n-1)}{2}$$

collision tests at each frame,

$$\sum_{m=1}^{f} \frac{n_m(n_m-1)}{2}$$

would suffice.

Using the example showed in Figure 26 to illustrate what this means in terms of the number of collision tests to perform, first the number of sprites in each field of the grid must be counted. Since some sprites belong to more than one field, this amounts to the following:

16	15	18
18	22	20

This means that instead of having to perform

$$\frac{100 \cdot 99}{2} = 4950$$

collision tests,

$$\frac{16 \cdot 15 + 15 \cdot 14 + 18 \cdot 17 + 18 \cdot 17 + 22 \cdot 21 + 20 \cdot 19}{2} = 952$$

would suffice.

Apart from performing the collision tests, the sprites have to be sorted, in order to determine which sprites belong to which field(s) in the grid. The time used to do that also has to be taken into consideration when deciding whether or not to implement this broad phase collision detection algorithm. However, I will not go into details as to how this is done or the time complexity of that.

Ideally, each sprite will belong to exactly one field, and all fields will have the same number of sprites. In this case,

$$f * \frac{(n/f) * ((n/f) - 1)}{2}$$

pairs of sprites needs to be tested for collision.

With 100 sprites, this means that the number of collision tests needed would decrease from 4950 to 784 at each frame, under ideal conditions.

Conditions are rarely ideal, however, because

- sprites can be on the edge of up to four fields, thus having to be tested for collision with all other sprites in those fields
- sprites can be unevenly distributed, necessitating more collision tests in one field than another

Also, sorting the sprites into different fields will take some time, and even though this can be optimized, it will still make the worst case scenario even worse than if the Gridding Method were not implemented at all.

If the Gridding Method were to be implemented, the number of fields in the grid should be determined. Using too few fields means that there will be too many sprites per field, resulting in more collision tests to be performed. Using too many fields will mean that more sprites are on one or more edges of the fields, which means that they will be tested for collision with sprites from several fields. Too many fields might also mean that more time will have to be spent sorting the sprites.

3.4.1.2 Narrow Phase Collision Detection

When testing two sprites for collision with each other, it is necessary to define when a collision takes place. To do that, the shape of the sprites must be taken into account. The bitmap of the sprite is rectangular, but usually some of the pixels in the bitmap are transparent, thus making it possible to make sprites in non-rectangular shapes. In other words, the shape used to test for collisions with other sprites need not be the rectangular shape of the sprite image. Thus, in the next sub-sections I will discuss collision between different shapes of sprites, and present formulas for testing whether a collision is taking place or not. In the formulas I present, I will refer to x and y as a sprite's position on the screen, which is usually the top, left corner of the sprite. Also, h and w will refer to the sprite's height and width, respectively. When referring two one of two sprites, I will use the convention of naming a number representing the sprite, so that e.g. h_2 will refer to the height of sprite number 2. Also, I use a red, dotted line to indicate the shape of the sprite that is used for collision testing. An illustration of a sprite is shown in Figure 27, here.

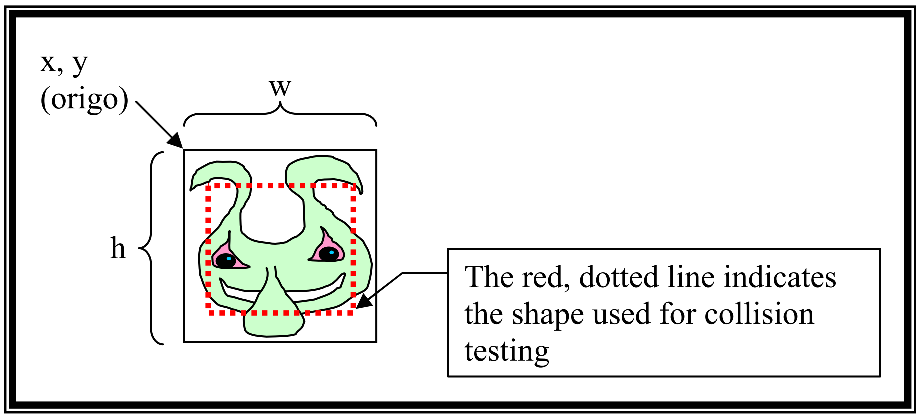

x, y
(origo)
w
h
The red, dotted line indicates the shape used for collision testing

Figure 27: Illustration of a sprite

3.4.1.2.1 Outer boundary sprite collision detection

As mentioned earlier, every sprite consists of a rectangular bitmap, even though some pixels may be transparent, and thus the boundary of that bitmap can be used to define the boundary of the whole sprite, so that a collision between two sprites is defined to occur, when the bitmaps overlap. This is illustrated below:

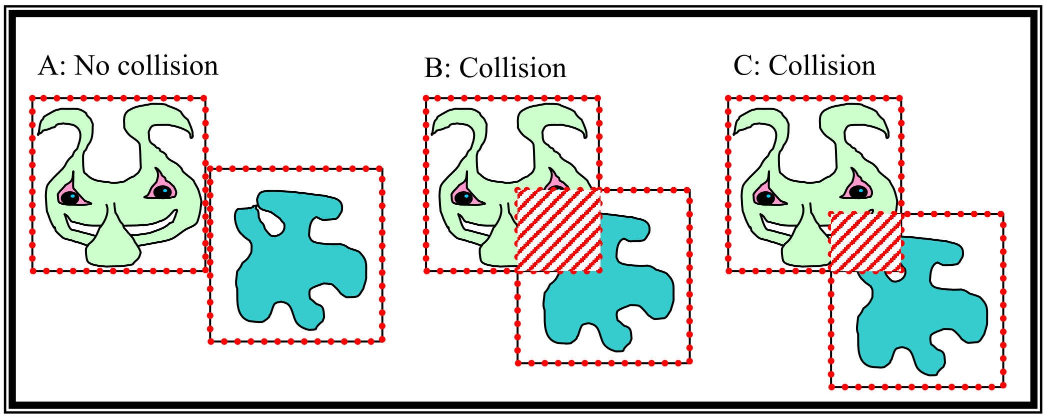


Figure 28: Outer boundary sprite collision detection.

In Figure 28 above the red dotted rectangles mark the outer boundaries of each sprite, as that is used to check for collision, and the red striped areas mark the part where two sprites overlap. In Figure 28.A the two rectangles do not overlap, and thus no collision has occurred. In Figure 28.B the rectangles do overlap, thus the sprites are colliding. The weakness of this method is illustrated in Figure 28.C, as the outer boundary rectangles do overlap, thus causing a collision to be registered, even though the figures in the sprites do not collide. When playing a game, this can be rather frustrating for the game player, as it may lead to events the game player did not expect, such as the loss of a life, even though a collision with e.g. a bullet was not visible.

The test for collision with this method can be performed by calculating the truth value of the expression below, where a collision occurs if and only if the expression evaluates to true.

$$x_1 < x_2 + w_2 \;\&\; x_2 < x_1 + w_1 \;\&\; y_1 < y_2 + h_2 \;\&\; y_2 < y_1 + h_1$$

3.4.1.2.2 Inner boundary sprite collision detection

To reduce the times when a player becomes frustrated on account of collisions between sprites, that occur even though no visible parts of them overlap, a better approach than the outer boundary can be made. Thus, a rectangle within the outer boundary can be defined, identifying the area of sprites that must overlap to cause a collision to be detected. This is called the inner boundary. Again, the red, dotted line indicates the area of each sprite to be tested for overlap:

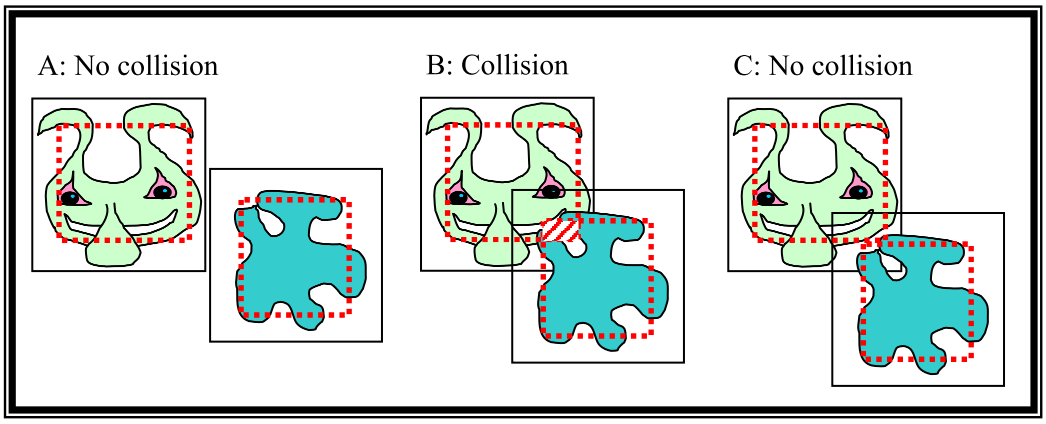


Figure 29: Inner boundary sprite collision detection

Let t, b, l and r represent distances from the top, bottom, left and right sides respectively, as shown in Figure 30.

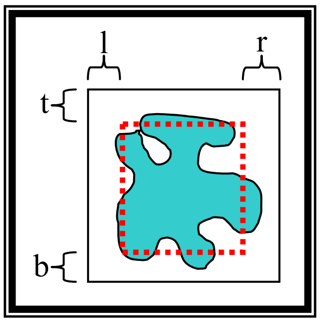


Figure 30: Definition of t, b, l and r.

Then the test for collision with this method can be performed by calculating the truth value of this formula.

$$(x_1 + l_1 < x_2 + w_2 - r_2) \& (x_2 + l_1 < x_1 + w_1 - r_1) \&$$
$$(y_1 + t_1 < y_2 + h_2 - b_2) \& (y_2 + t_2 < y_1 + h_1 - b_1)$$

3.4.1.2.3 Circular boundary collision detection

Another method of testing for collision between two sprites is to define a sprite as circular, which is illustrated in Figure 31, below.

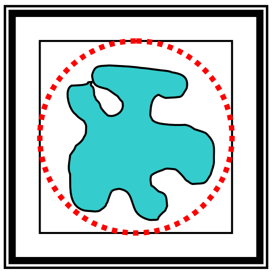


Figure 31: A sprite with a circular boundary

This makes sense, if the sprite images closely correspond to the circular shape, and is particularly useful if the sprites must be rotated at any angle, as this does not have any effect on the formula for testing for sprite collision. The latter is illustrated in Figure 32, Figure 33 and Figure 34, below:

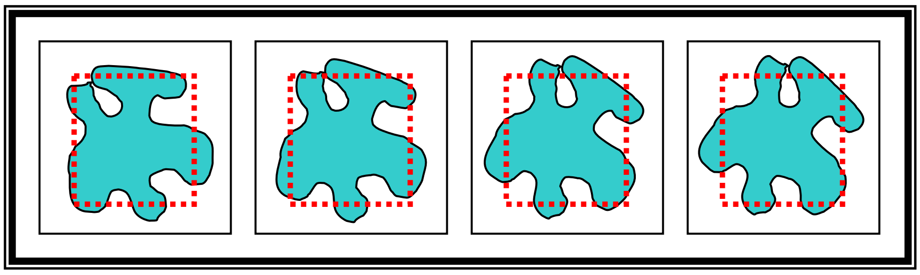


Figure 32: A sprite is shown rotated. The inner boundary is not rotated.

In Figure 32 the inner boundary is not rotated, which makes the boundary out of synchronisation with the sprite image. This could cause the game player some confusion as to when a collision occurs and when it does not.

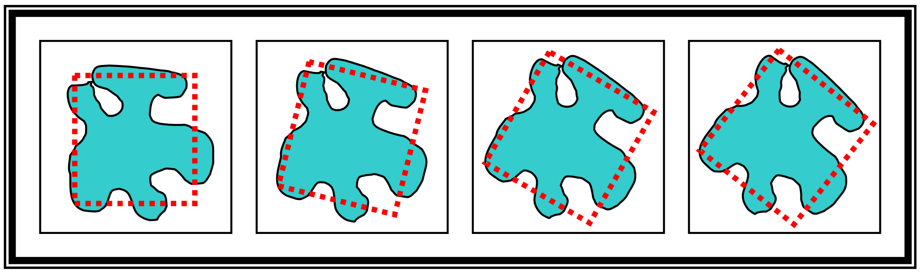


Figure 33: A sprite is shown rotated. The inner boundary is also rotated .

In Figure 33 the inner boundary is shown rotated together with the sprite image. The formula to be used when testing for collision between two sprites rotated at different angles with this approach might be somewhat complex, both in terms of constructing and performance when executed.

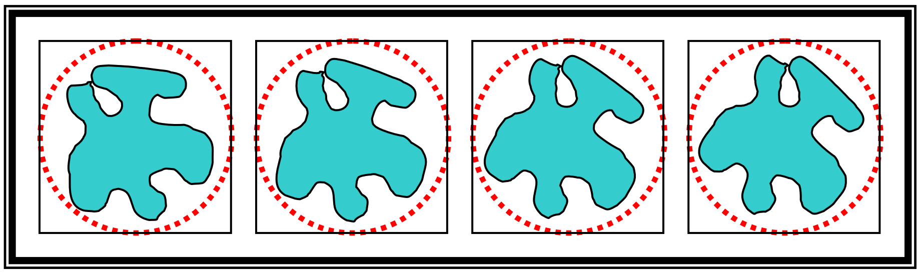


Figure 34: A sprite with a circular boundary is shown rotated.

In Figure 34 the rotated sprite has a circular boundary. The formula used to test for sprite collision will remain the same whether or not the boundary is rotated, as the rotated boundary will be identical to the non-rotated.

The formula for testing for collision between two sprites with circular boundaries is made by the use of Pythagoras' paradigm. Each circle has a radius, indicated by r_1 and r_2 respectively, which is illustrated in Figure 35, below.

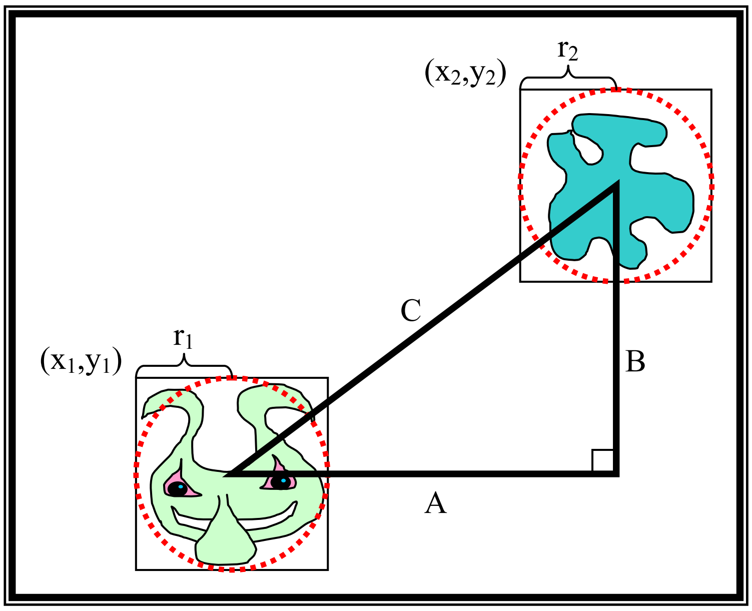


Figure 35: Illustration of the Circular Boundary Test.

A collision occurs, when the distance between the centres of the two sprites, C, is smaller than the sum of the sprites' radiuses, i.e. r_1 and r_2. Using the theorem of Pythagoras, the distance between the two centres of the sprites, can be calculated as $C = \sqrt{A^2 + B^2}$.

The length of A can be calculated as $A = |(x_2 + r_2) - (x_1 + r_1)|$ and the length of B as $B = |(y_2 + r_2) - (y_1 + r_1)|$. Thus, the length of C can be calculated as

$C = \sqrt{((x_2 + r_2) - (x_1 + r_1))^2 + ((y_2 + r_2) - (y_1 + r_1))^2}$ and so testing for collision between two sprites with a circular boundary can be done by calculating the truth value of this expression:

$$\sqrt{((x_2 + r_2) - (x_1 + r_1))^2 + ((y_2 + r_2) - (y_1 + r_1))^2} < r_1 + r_2$$

On some computers, especially older ones, calculations with real numbers use more CPU power than calculations with integer numbers. As the GBA has no hardware that is optimized for real numbers, I assume this is also the case for that system. It then follows that an integer square function is probably more effective than a square root function, in which case this formula can be used instead.

$$((x_2 + r_2) - (x_1 + r_1))^2 + ((y_2 + r_2) - (y_1 + r_1))^2 < (r_1 + r_2)^2$$

If the sprites are known to have the same size, and thus the same radius, the formula can be simplified to

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} < 2r$$

or, on integer computers

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 < (2r)^2$$

For sprites of a circular shape, that also need to be rotated, the circular boundary could be a good choice.

3.4.1.2.4 Pixel based sprite collision detection

One way to define when a collision between two sprites is taking place is when at least one non-transparent pixel from one sprite overlaps a non-transparent pixel from the other sprite. This way, a test has to be performed on every pixel in the area, where the rectangles of bitmaps the two sprites overlap as illustrated in Figure 36, thus making the method potentially rather expensive. However, if the rectangles of the bitmaps do not overlap, the method is no more expensive than the other methods.

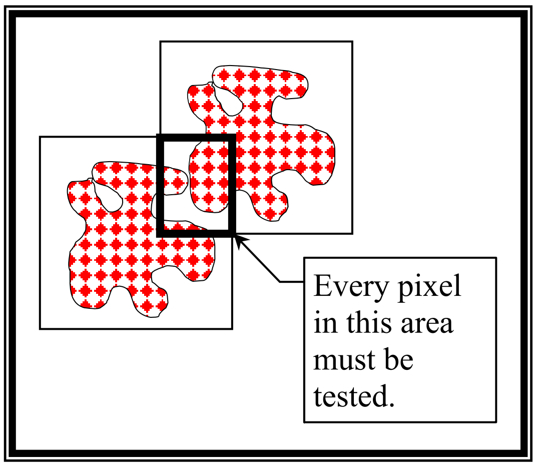


Figure 36: Pixel based sprite collision detection

3.4.1.2.5 Preliminary test

Some of the narrow phase collision detection algorithms are likely to be more expensive than others when it comes to performance. For argument's sake, let us assume that e.g. the circular boundary method is more expensive than the outer boundary method. This assumption seems quite reasonable, since the circular boundary method involves several calls to a square function, which is relatively expensive when it comes to performance on some computers. In view of the fact that a sprite never exceeds the outer boundary, as defined in section 3.4.1.2.1 above, using a more expensive method such as the circular boundary method can be avoided in many cases, thus improving performance. This can be done by first performing an initial outer boundary collision test. If that test turns out positively, i.e. indicating a collision, we can proceed with performing the circular test to check if a collision really is taking place. If the initial test turns out negatively the test can stop, as no collision is then possible. Thus, the more expensive test is only performed when the sprites are relatively close to each other.

Below follow some thoughts on deciding when to implement a preliminary test.

If C indicates the amount of time used for checking for collision with the circular sprite collision method, and O indicates the amount of time used for checking for collision with the outer boundary collision detection method and P is the probability for the latter test to be negative, then a combination of the two tests should be used if the following expression is true:

$$C > O + (1 - P) * C$$

through normal mathematical rules, this can be converted to:

$$C > O + (1 - P) * C$$
$$\Updownarrow$$
$$C > O + C - P * C$$
$$\Updownarrow$$
$$0 > O - P * C$$
$$\Updownarrow$$
$$P * C > O$$
$$\Updownarrow$$
$$\underline{\underline{P > O/C}}$$

I will illustrate this with an example, using the arbitrarily chosen values shown below.

- ***P***: The probability for the outer boundary test to be negative is estimated at 0.95.
- ***O***: Performing the outer boundary test takes 10 units of time.
- ***C***: Performing the circular boundary test takes 50 units of time.

$$P > O/C$$
$$\Updownarrow$$
$$0.95 > 10/50$$
$$\Updownarrow$$
$$0.95 > 0.2$$

This statement is true. Thus, an outer boundary test should be performed before the circular test.

Another example, with a different set of arbitrarily chosen values:

- ***P***: The probability for the outer boundary test to be negative is estimated at 0.7.
- ***O***: Performing the outer boundary test takes 45 units of time.
- ***C***: Performing the circular boundary test takes 50 units of time.

$$P > O/C$$
$$\Updownarrow$$
$$0.7 > 45/50$$
$$\Updownarrow$$
$$0.7 > 0.9$$

This statement is false. Thus, only the circular test should be performed.

If it is possible to measure, in some unit, the outer boundary test and the circular test, along with the probability of the outcome of the former, this formula can be helpful in deciding which method to use.

3.4.1.2.6 Additional thoughts on testing two sprites for collision

The shape that defines a sprite in terms of collision detection can of course be more complex than merely involving a simple circle or rectangle. For instance, it can be a polygon, or a shape defined by several connected Bezier-curves. Or it can be made up from several shapes, as illustrated in Figure 37 below, where a sprite showing an airplane is parted by three rectangles. As with the circular method, it might be beneficial, performance wise, to make an outer boundary test before proceeding with a test that involves several rectangles, circles and/or other shapes.

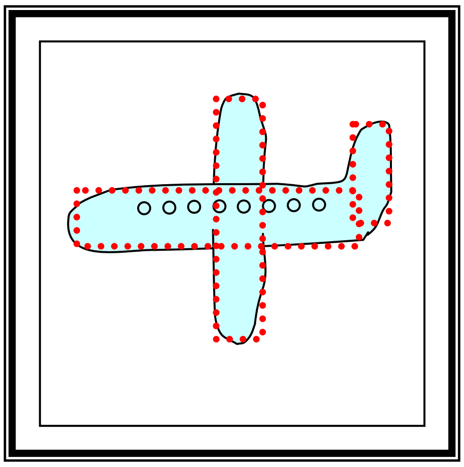


Figure 37: A sprite showing an airplane, divided into three rectangles.

Another thing is that sprite collision testing can be made more subtle, if a collision is only registered, when the collision is greater than a given threshold. For example, in the case of pixel based sprite collision detection, the collision would only be registered, if more than some determined number of pixels from one sprite overlapped with pixels from the other sprite. Likewise, in case of the outer boundary method, the collision could be made to only be registered if the overlapping area was larger than some given size. In the case of the inner and circular boundary methods, the inner boundary or the circle could simply be made smaller, and that way allow for some more overlap between two sprites before a collision was registered.

The advantage of using a threshold as described above is, that it makes it possible to determine how strict a sprite collision detection algorithm should be. This can be useful in games, where collision detection algorithm that is too strict may cause the game player some frustration when sprites are unexpectedly registered as colliding.

3.4.1.2.7 Summing up on Narrow Phase Collision tests

In this section I will sum up the major advantages and disadvantages of the various methods described above, in an easy-to-read fashion. This is done in the table below, in Figure 38.

	Pro	Con
Outer boundary	Fast	Very inaccurate
Inner boundary	Fast, but a bit slower than the Outer Boundary	A bit more accurate than the Outer Boundary, as it can be adjusted to the sprite image.
Circular boundary	Good for sprites that should be rotated.	A bit slow.
Pixel based	Exact by definition.	Very slow.
Several shapes	Relatively fast and more accurate than the others (except the Pixel based method)	More complicated to implement than the others.

Figure 38: Pros and cons of various narrow phase collision testing methods.

3.4.1.3 What to do, when sprites collide

In order to detect whether or not two sprites have collided, obviously this must be tested and it must be determined how to perform the wanted actions that should be performed when that test turns out positively.
There are two ways of doing this:

- It can be left entirely to the game programmer, i.e. the game engine should include a subroutine that can check two sprites for collision, but should not automatically call this subroutine. Instead, calling that subroutine should be left to the game programmer, who must then decide in which parts of his program, the subroutine should be executed and depending on the result of the call, perform the relevant actions. In pseudo-code it would look something like this
    - if collision(SpriteA, SpriteB) then DoSomething;

  The game programmer must then make sure to perform this test regularly.
- It can be event based, and so it will be the responsibility of the game engine to make sure that the sprite collision tests are carried out automatically at a regular interval, the most obvious interval being once every frame. It must then be the game programmer's responsibility to define which pairs of sprites should be tested for collision and also what to do, when a collision occurs. The latter can be achieved by the game programmer supplying a subroutine to be executed when two sprites collide. In pseudo-code it would look something like this
    - OnEvent collision(SpriteA, SpriteB) DoSomething;

The advantage of the first alternative is that it is easy to implement in the game engine, as everything is more or less left to the game programmer. Also, it provides the game programmer with maximum freedom to do things his or her way, since the game engine does not put any restraints on the game programmer. On the other hand, it might be difficult for the game programmer to make a game that should constantly perform sprite collision tests. That is to say, that the freedom provided by this method might very well be too much.

Furthermore, since some of the purposes of a game engine are to support the game flow and the game programmer, it does not make much sense to make a game engine at all, if everything is left as the responsibility of the game programmer. This means that unless unusual arguments speak against it in a concrete situation, alternative two seems to be the better one, and thus most sensible game engines would probably choose the event based method.

3.4.2 Sprite Handler

As sprites are an important aspect of most 2D games, the sprite handler becomes equally important, since the purpose of a sprite handler is to provide a systematized way of dealing with sprites, primarily in terms of initializing, moving and destroying them. So, although a sprite handler in not strictly necessary, particularly if only dealing with a few sprites, it is one of the main tasks for a game engine.

As with sprite collisions, discussed in the previous section, movement of sprites can be left as the responsibility of the game programmer or the responsibility of the game engine, and so two alternatives exist.

- Movement of sprites is the **responsibility of the game programmer**, so that the game programmer must move the sprites when needed. Possibly directly followed by a manual sprite collision detection test.
- Movement of sprites is the **responsibility of the game engine**. Each sprite must move automatically, and as each sprite must be able to move independently of the others, the game programmer must provide a sub-routine for movement that can be assigned to each sprite.

When it comes to destroying a sprite, it must be decided which sprites can be destroyed. The following possibilities exist:

- Any sprite can be destroyed.
- Only the last created sprite can be destroyed. When that is destroyed, the second lastly created sprite can be destroyed. Etc.
- All sprites must be destroyed at the same time.

The choice can very well depend on hardware capabilities. On a PC, for example, where the hardware has no support for sprites, the hardware also does not set any limitations to the use f sprites, and so there is no reason not to enable the game programmer the possibility of destroying any random sprite. On a GBA, on the other hand, where the hardware demands that sprite data is stored at particular memory addresses, there is a need for the system to keep track of which sprites are used and which are not. Such a system is simpler if sprites are at one end of the sprite memory and space for new sprites are at the other end, than a system where any location in the sprite memory could be in use or not.

Since the game engine is supposed to assist the game programmer as much as possible, the alternative where a game engine that takes the responsibility of moving sprites automatically is preferable to the alternative where that responsibility is left to the game programmer.

3.5 Background graphics

In order to make a game more appealing to the game player, an image can be shown in the background and so a game engine should support showing background images. Also, an ideal game engine should support some sort of interaction between the background image and the sprites. The background image can be a simple bitmap, i.e. a two-dimensional array of pixels, or be made up by tiles. Many games use background scrolling and some use rotating and scalable backgrounds, so a game engine that can handle this is preferable to one that cannot.

Many games use tiled backgrounds, and these games could thus benefit from a game engine that could handle tiled backgrounds and especially sprite/tile collision.

The advantage of using sprite/tile collision detection compared to sprite/sprite collision detection is that since tiles do not move on the background, the sprite/tile collision tests has a time complexity of O(n) where n is the number of sprites, as opposed to the sprite/sprite collision tests that has a time complexity of $O(n^2)$ – at least when no broad phase collision detection algorithm is implemented.

The reason is that a sprite/background tile collision test can be performed in constant time, since the location of the sprite can be used to look up the relevant tiles in constant time, which means that one test per sprite is sufficient, whereas with sprite/sprite collision, each sprite has to be tested for collision with every other sprite.

At least the possibility of showing still background images should be part of any reasonable game engine, whereas the possibility of using tiled backgrounds must be part of any superior game engine.

4 Game engine on Game Boy® Advance

In the previous two chapters, chapters 2 and 3, I have described the GBA and a general game engine, respectively. In this chapter I will combine those two areas and choose between the possibilities that best suit a game engine customized for the GBA. Obviously, I cannot always choose the best solution, as time prevents me form implementing a too complicated game engine. Therefore, my choices will be a weighing out of what I estimate to make out the best game engine for the GBA within the given time limit. The source code for the final game engine can be found on the enclosed CD.

This chapter will largely consist of discussions between different solutions, which will end up in choices. The strategy for making these choices is based on simplicity, user friendliness towards the game programmer and taking advantage of specific capabilities of the GBA, meaning that I will primarily focus on the areas where the GBA is hardware enhanced.

4.1 Content management

Even though any game engine could almost certainly benefit greatly from a content management system, I choose not to make one as part of the game engine for the GBA, since that would be a big project in itself. In stead, I will depend on separate programs that can convert images and sounds to arrays in the form of C/C++ source code and make the game engine capable of handling these arrays in a reasonable manner, so that this way, images and sounds can relatively easily be incorporated in the games.

4.2 Loop

In order for most games to function properly, some things must be carried out repeatedly. These tasks, which are the primary components of the game, include the following.

- Sprite handling, inclusive sprite movement and sprite collision detection.
- User input.
- Sound generating.

As these tasks must be carried out repeatedly, they will be included in the loop that is responsible for executing the game.

In order to secure a consistent frame rate, it is necessary to use the vertical blank interrupt. As mentioned in section 2.2.5.2 about Interrupts, it is automatically generated every time the screen has been refreshed. The game engine must necessarily set up the GBA so, that every time the vertical blank interrupt is generated, the work that should be done in every frame is done, i.e. the movement of sprites, handling of sprite collisions, handling of user input and updating the screen.

As mentioned in section 2.2.5.2 about interrupts on the GBA, when handling an interrupt, further interrupts are usually disabled. What this means is that if the game engine spends too much time in a frame, further interrupts will be ignored and thus one or more frames will be skipped. This will automatically result in a lower frame rate, which is preferable to allowing an interrupt to interrupt another, and thus risking a system crash.

4.2.1 Sprites

In order avoid using too much time on the game engine I will not implement a number of the sprite effects. What I will implement includes different sizes of quadratic sprites, as well as movement of sprites and sprite collision detection. In order to keep things simple, all sprites must use the same palette of 256 colours. What is excluded, then, are non-quadratic shapes as well as effects such as mosaic, flipping, rotation, scaling, alpha blending along with a priority to indicate which sprites should be shown in front of other sprites. Also, sprite animation will not be a part of the game engine at this time. However, I might very well include many of these features in the game engine at a later time, after this project has finished.

Movement of the sprites will be handled by a sprite handler, which is the subject of the next section.

4.2.2 Sprite handler

As described in section 3.4.2 about sprite handlers in general, a sprite handler is responsible for movement of sprites as well as performing proper actions when sprites collide. How to do this is the topic in the next two subsections:

4.2.2.1 Sprite movement

In various games a lot of algorithms exist, that control the movement of sprites, because sprites are typically used to represent moving objects within a game. These algorithms range from very simple predetermined paths, such as moving left and right within specified boundaries, to very complex algorithms using artificial intelligence.

How sprites move is a substantial part of most sprites based games, and is thus best left to the game programmer, as even a large selection of predetermined movement algorithms would probably limit the game programmer more that it would help.

So, in order not to limit the game programmer to a predetermined selection of movement algorithms, I find it better to supply him with the means of making his own.

What the sprite handler can do, then, is provide simple means of achieving movement. Thus, I will make it so that the game programmer must assign a subroutine that handles movement to each sprite. Obviously, in order to keep down the number of subroutines, each of these movement subroutines must be able to be used by several sprites. Thus, each of these subroutines must have access to the sprite it is moving at any given time.

To make things simple for the game programmer, I will also provide every sprite with certain attributes (variables) for the game programmer to manipulate, such as an x and a y value that, when changed, will influence the sprite's position on the screen. Furthermore, in order to make it easy for the game programmer to make sprites that move at other speeds than an integer number of pixels per frame, e.g. a sprite that moves 1.5 pixels each frame on average, I will use what I call sub-pixel movement. What it means in this case is, that the x and y values will be divided by 128, and the resulting values will determine the sprite's location on the screen. Thus, if for instance, a sprite's x variable is increased by 192 at each frame, it means that the sprite will alternately move one and two pixels to the right. In addition to the x and y variables, I will add some other variables for the game programmer's convenience.

These variables will not have any automatic influence on the sprites, but are merely included for the convenience of the game programmer, who has access to them from within the movement sub-routines and so can use them any way he or she wants, e.g. to control the direction or speed of a sprite or to determine when to change direction.

4.2.2.2 Sprite Collision Detection

In the following sub-sections, I will give an account of the choices I have made with regards to sprite collision detection. This includes deciding whether or not to implement a broad phase sprite collision detection algorithm, as well as choosing which narrow phase sprite collision algorithm(s) to implement. It also includes choosing between an event based method and a method that leaves the sprite collision detection pretty much up to the game programmer.

4.2.2.2.1 Broad Phase Collision Detection

As the time for this project is rather limited, I have to prioritize which of the elements a game engine can include I will focus on, when implementing the game engine for the GBA. That and the fact that a broad phase sprite collision detection algorithm is not strictly necessary in order to produce a functional game engine, although it can certainly help improve its speed, are the reasons why I choose not to implement any broad phase sprite collision detection algorithms at all. Still, it has to be decided, which pairs of sprites should be tested for collision and how to initiate the proper actions when collisions occur. As mentioned in section 3.4.1.3, there are two possibilities:

- Leave it to the game programmer.
- Make an event based system.

In order to support the game flow and make game programming an easier task for the game programmer, I will make the sprite collision detection part of the game engine event based. Thus, the game engine must provide means for the game programmer to define pairs of sprites along with sub-routines that will be executed when the sprites collide. And, naturally, the game engine must then check each pair of sprites for collision at each frame, and when a collision occurs execute the corresponding sub-routine.

4.2.2.2.2 Narrow Phase Collision Detection

In section 3.4.1.2 I have described several methods to perform Narrow Phase Sprite Collision Detection. One or more of these methods have to be implemented, or a way to enable the game programmer to easily implement his or her own method has to be provided. At this point, I see no particular reason why the game programmer should want to include his or her own narrow phase collision detection algorithm, so I will include one rather than bother him or her with that task. So it remains to choose which algorithm to use. As the pixel based method is expected to be too time consuming I will choose one of the other methods. There are no weighty reasons for choosing one method over the other at this point, but since the GBA has hardware support for sprite rotation at any angle, as described in section 2.2.2 about Sprites, the circular boundary method appear appealing. An advantage of the circular boundary method is that the algorithm of that method is the same whether or not the spites have been rotated, since a circle does not change its shape when rotated. However, as this is a simple game engine and time for the project is limited, I will not implement sprite rotation. Still, I will implement the circular method so that the game engine is prepared for a time when sprite rotation is implemented. As I expect that most games will have a lot more narrow phase sprite collision detections with a negative result, i.e. the two sprites tested do not collide, than with a positive one, i.e. they do collide, I will precede the circular boundary test with an outer boundary test in order to quickly reject most collision tests.

4.2.2.3 Sprite Images

As a GBA has no access to a file system, all data must be incorporated in the source program, which includes sprite images. Fortunately programs for converting images to C or C++ exist and they are the topic of chapter 7, but here it is in its place to note what the game engine must do in order to be able to work with these images. The above mentioned programs generate C or C++ fragments that define one or more arrays holding the image data. In order to use that data as sprite images, it needs to be placed in video memory on the GBA. Thus, the game engine needs to include a subroutine, which can copy an array of data from a location in memory to a location in the video memory. This subroutine should return some sort of reference to the address to which the data has been copied, so as to make it possible to later connect the sprite image with a sprite.

4.2.3 Sound

As mentioned in section 3.3, an ideal game engine should be able to do sound mixing in order to play several sounds simultaneous, such as background music along with sound effects. However, this being a simple game engine, I have to choose not to implement a sound mixer. There are several reasons for this:

- A sound mixer should be as effective as possible, so as not take more CPU power from the rest of the game than necessary. This means that in order to make a satisfactory sound mixer, it must be written in assembler, with which I am not too familiar on the GBA.
- Several sound mixers for the GBA already exist and are publicly available, although not all of them free of charge. The most well-known are probably the Apex Audio System [APEX] and Krawall Advance [KRAWALL].
- Since several sound mixers already exist, game programmers might already be comfortable with one or more of them, and have their preferences as to which one to use. By not making a sound mixer part of the game engine, the game programmers can make the choice of which one to include, if any at all.

Instead, I choose the simplest acceptable solution, and doing so, choices have to me made with regards to these items:

- Using one or both sound channels
- Controlling the sound volume
- Controlling sound volume for each speaker individually (for stereo effect)
- Controlling the playback frequency
- What should happen when starting a new sound before the old one has finished? A choice should be made between these possibilities:
    - The new sound interrupts the old one, with the result that end of the old sound is never played
    - The new sound is queued, so that is played when the old sound has finished.
    - The new sound is rejected, i.e. not played.

As mentioned, I will choose the simplest solution to each of these items. Thus,

- Only one sound channel will be used. However, the output from that channel will be directed to both the left and right speaker, in order to improve the sound experience when using a headset. Without a headset it will make no difference, as the GBA only has one built-in speaker.
- No means will be provided for the game programmer to control the sound volume. The game engine will set the volume at the highest level, and leave it to the game player to turn it down manually, if so desired.
- The playback frequency could be set at a pre-determined value. However, this would not be a satisfactory solution, as the game programmer might have sounds that are sampled at different frequencies. So, the game programmer must supply a frequency when initiating the playback of a sound.
- As to what should happen if a new sound is initiated while another one is still playing, I find that it would probably cause some confusion, if sounds were queued, as it could have the effect that sounds were being played long after the event that initiated the sound had passed. For example, it would be strange to hear an explosion seconds after it had taken place. When choosing between interrupting an old sound and rejecting a new one, I find that the most recent event must take precedence over older events, meaning that a new sound should interrupt an old one, rather than being rejected altogether.

4.2.4 Background graphics

As mentioned in section 2.2.1 the GBA has two kinds of backgrounds, three bitmapped and three tile based. The tile based backgrounds are more difficult to deal with, since they involve a palette, a tile map and a lot of tiles, opposed to the bitmap modes that primarily involve handling a two-dimensional array of pixels. The benefits of the tiled backgrounds are the many effects that can be performed on them, namely scrolling, rotation and scaling. Another benefit is that a sprite/background collision detection algorithm would probably be more efficient when dealing with a tile based background, as that would usually just involve a check of a few background tiles, whereas on a bitmapped background it would involve the checks of a good many pixels. On the other hand, the tiled background modes have a maximum of 256 colours, whereas the bitmapped mode 3 takes advantage of the full colour spectrum of 32,768 colours.

As time prevents me from implementing more than one background mode into the game engine, I choose background mode 3, partly because it is the simplest background mode to implement, partly because it offers the highest resolution with the most available colours. This means that the game engine will have no support for the other background modes, and so must do without scrolling, rotating and scalable background graphics. It does mean, though, that images will be easily included as backgrounds for games.

4.2.5 User input

As user input on a GBA typically means which buttons are pressed and which are not, and as the status of all buttons is handled through one 16-bit word in memory as mentioned in section 2.2.3, it is hardly necessary to make a subroutine in the game engine to handle user input. However, the game programmer's need to access memory directly should be as limited as possible for simplicity reasons. I will, therefore, provide the game programmer with a simple C++ macro, which extracts the relevant information from the relevant address of memory. I expect that this macro will mainly be used in functions that are used to handle the movement of the sprite that represents the player character, but how and when it is used is entirely up to the game programmer.

5 Performance and scalability

The present game engine will scale badly with regards to the Sprite Collision Detection algorithm, because no Broad Phase Sprite Collision Detection algorithm has been implemented.

Performance measurement is performed by allowing the game programmer to specify two functions: One that will be called at the beginning of each frame and one that will be called at the end of each frame. In my example game, the shoulder buttons, L and R, are used to set up a timer, and register how much time passes between the start of the frame and the end of the frame, and vice versa.

Several areas of a game engine could be worth testing for performance. The purposes of testing performance are two; Firstly, it can help determine how well the game engine scales when it comes to the number of sprites and maximum number of sprite collision tests per frame as well as help determining how costly the routines made by the game programmer can be and secondly it can help determining which areas to work with, if better performance is needed, e.g. if more collision tests per frame are needed.

Several variables influence the performance of a general game engine, such as the number of simultaneous sprites, the number of sprite collision tests per frame, the frame rate as well as the complexity of the routines for moving a sprite, checking for sprite collision and handling sprite collision. As the frame rate automatically decreases if and when the game engine is trying to do too much work in every frame, as mentioned in section 4.2, it makes sense to find out how many sprites and sprite collision tests can be handled, and what numbers will cause the frame rate to decrease. Obviously, I haven't got the time to test the frame rate for all combinations of number of sprites and number of sprite collision tests that are performed in every frame, and I will therefore use the maximum number of sprite collision tests in relation to the number of sprites, i.e. a test where every sprite is tested for collision with every other sprite. As mentioned in section 3.4.1, the number of sprite collision tests, when every sprite is tested for collision with every other sprite is $\frac{n*(n-1)}{2}$, where n is the number of sprites.
Thus, the purpose of the test I will perform is to find the maximum number of sprites at any frame rate, under the condition, that every sprite is tested for collision with every other sprite at each frame.

I expect to present the result in a graph that looks something like the one I have shown in Figure 39, below.

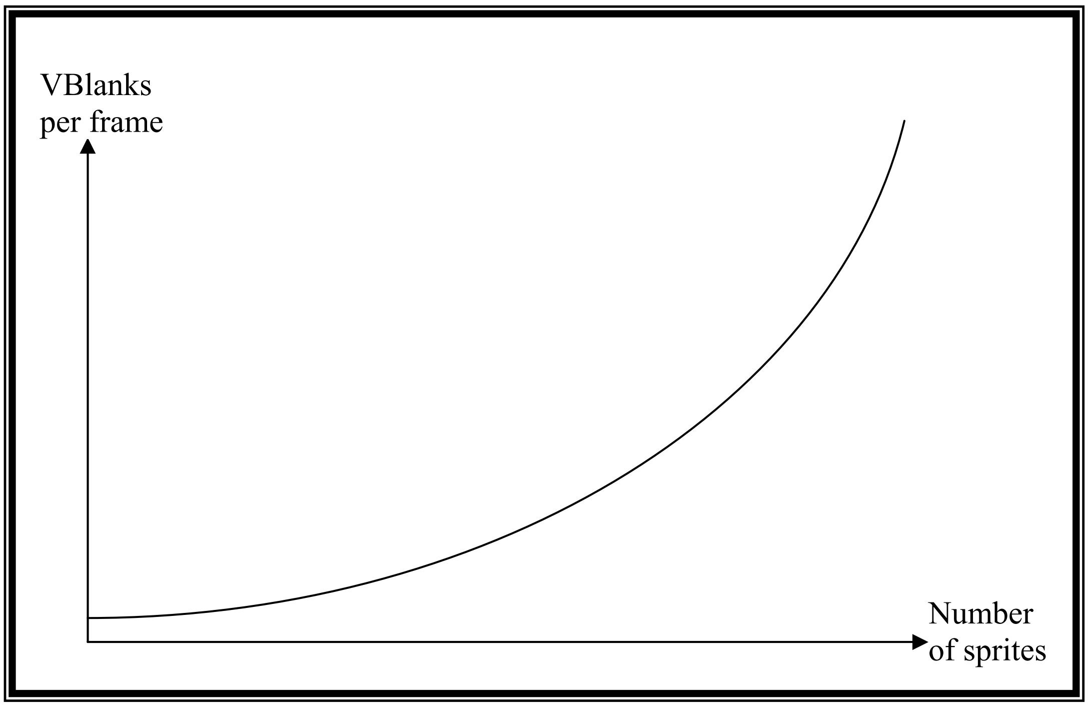


Figure 39: Prototype of graph, showing the relationship between frame rate and the number of sprites as well as sprite collision tests.

I will, therefore, test the performance of the game engine with the purpose of finding the maximal number of simultaneous sprites that are all tested for collision with each other in each frame. In doing so, I will create a small test program, that uses the built in timers of the GBA to register the time that passes in each frame from just before sprite handling starts to just after sprite handling has finished.

5.1 The Performance and Scale Testing Program

I plan to make a simple program, with a number of sprites, placed randomly on the screen, each moving in some direction. When two sprites collide, they will switch direction with each other, and when a sprite is moving out of the screen, it must bounce off the side of the screen. A draft of this test program is shown below in Figure 40, below:

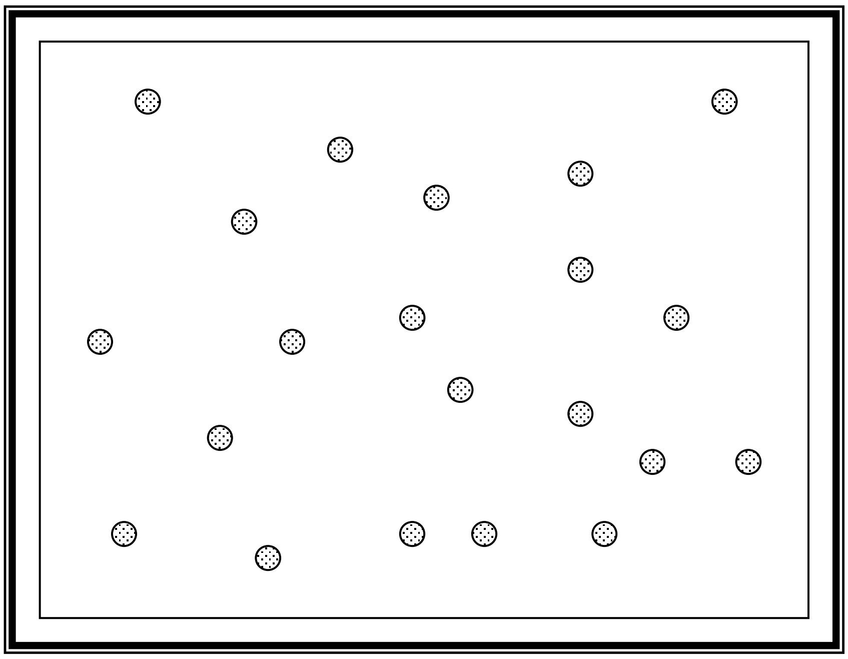
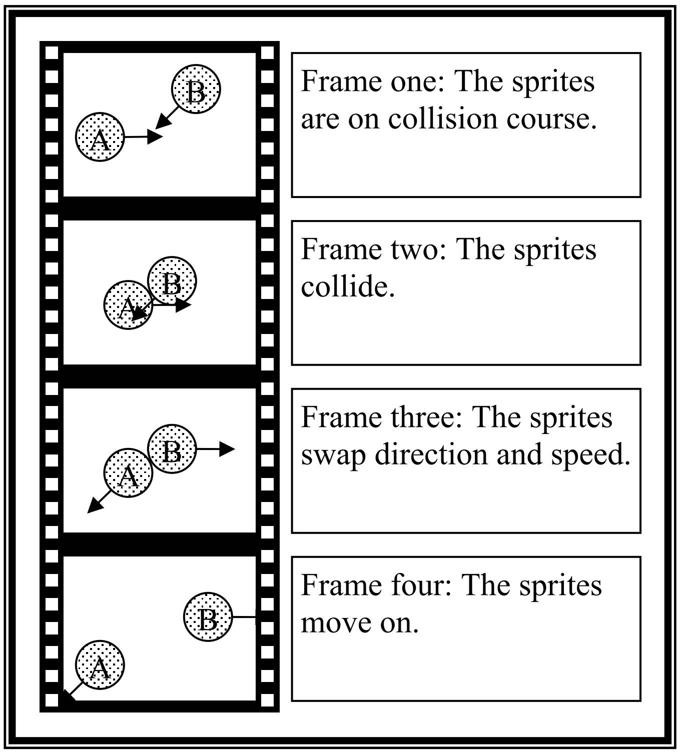


Figure 40: The performance and scaling test program.

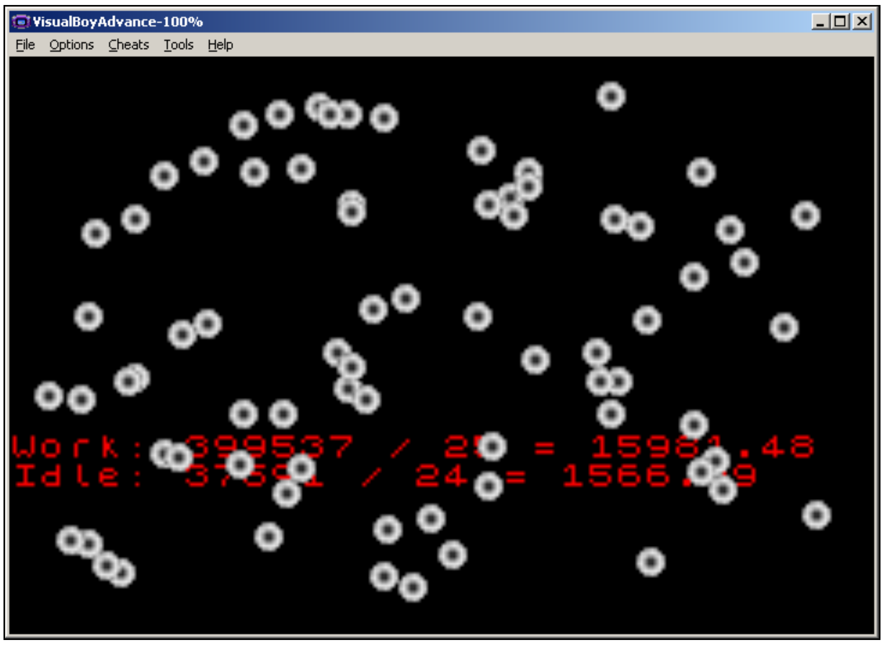
Below in Figure 41, is an extract of the screen in four frames, which shows what happens when two sprites collide.


B
A
Frame one: The sprites are on collision course.
B
A
Frame two: The sprites collide.
B
A
Frame three: The sprites swap direction and speed.
B
A
Frame four: The sprites move on.
Figure 41: Four frames showing the workings of the performance and scaling test program.

The source code for the performance and scale testing program can be found on the enclosed CD.

5.2 The Performance and Scale Test

I have executed the performance and scale test with various numbers of sprites. A screen shot of the test program running is shown in Figure 42.



```
Work: 399537 / 25 = 15981.48
Idle:  37591 / 24 = 1566.29
```

Figure 42: The Performance and Scale Test program running.

Since the purposes of this test is to find the highest possible number of sprites at any given frame rate under certain conditions that every sprite is tested for collision with every other sprite at every frame and a very simple sprite movement algorithm is used, I have performed a test for every ten sprites, and when the frame rate changed, I tried with a slightly different number of sprites until it was clear at what number of sprites the frame rate changed. A diagram showing all the results of all the performed tests is shown in Figure 43 below, which also shows a second degree polynomial, that closely approximates the results.

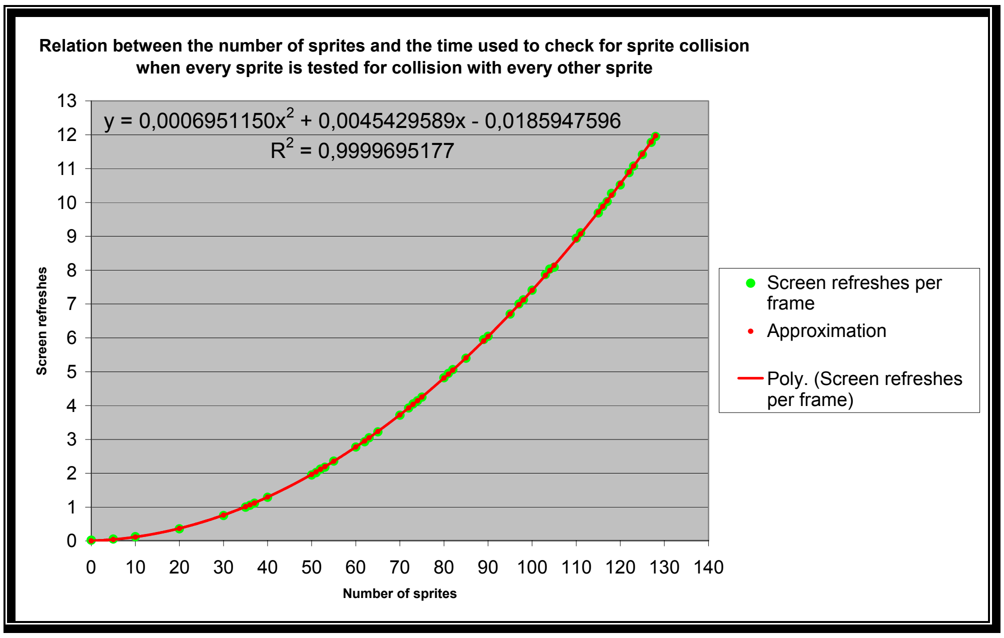


Figure 43: Relationship between the number of sprites and the frame rate. The green dots represent the actual values found in the test. The red line and the red dots represent the approximated polynomial.

The polynomial is found through quadratic regression, of which Microsoft® Excel® is capable, and roughly states that

$$y = 0{,}0007x^2 + 0{,}0045x - 0{,}0186$$

where x is the number of sprites and y is the number of screen refreshes per frame. Since it is known that the screen is refreshed 59.73 times per second, it is possible to make a summary of the maximum number of sprites at any given frame rate, still with every sprite being tested for collision with every other sprite at every frame. This summary is shown in Figure 44 below.

Screen refreshes per frame	Frames per second	Maximum number of sprites	Number of sprite collision tests
1	59,73	35	595
2	29,86	50	1225
3	19,91	62	1891
4	14,93	72	2556
5	11,95	81	3240
6	9,95	89	3916
7	8,53	97	4656
8	7,47	103	5253
9	6,64	110	5995
10	5,97	116	6670
11	5,43	122	7381
12	4,98	128	8128

Figure 44: Summary of the maximum number of sprites at different frame rates, under certain circumstances.

The relationship between the number of sprites and the frame rate has to be taken into consideration when designing and programming a game, and Figure 44 can be used to help make the decision when choosing the desired number of sprites and the desired frame rate. However, as the figure shows, it is not possible to get a high frame rate combined with a high number of sprites. In some cases compromises need to be made. These can consist of the following.

- A broad phase collision detection algorithm must be implemented.
- A lower frame rate must be accepted.
- The conditions must be changed, if e.g. there is no need for every sprite to be tested for collision with every other sprite.
- The game engine must be optimized, e.g. by writing time critical sub-routines in ARM7 assembler in stead of in C++.

5.3 Conclusion of the performance and scale test

As expected, the test has shown that the relationship between the frame rate and the number of sprites has to be taken into consideration when designing a game. It has shown that there is still a long way up to handling the 128 sprites the GBA can handle simultaneously, with the condition that every sprite is tested for collision with every other sprite, at full speed, i.e. a frame rate of 59.73 frames per second. And so the test has shown that it would be tremendously beneficial for the game engine if it were to be equipped with a broad phase collision detection algorithm, which might allow all 128 sprites to be used simultaneously under the same condition. Furthermore, the test has provided the game programmer with a tool on which to base a compromise between the desired frame rate and the desired number of sprites.

6 Test Game

I have developed a simple test game, in order to show how simple it is to develop games using the game engine. It is a Space Invaders clone with the following features

- A splash screen
- A high score
- Three shelters that can be shot to pieces. Each shelter consists of six sprites that each take three shots to vaporize.
- Five lives.
- 30 aliens divided among three different alien images.
- Five different background images.
- Custom made graphics. This might not be an asset, as my artistic skills are somewhat limited.
- Graphical explosions when aliens are hit.
- Explosion sound when aliens are hit.
- Increased speed and difficulty, during game play.

The game and source code can be found on the enclosed CD. Some screen shots of the game can be seen in Figure 45 through Figure 50.

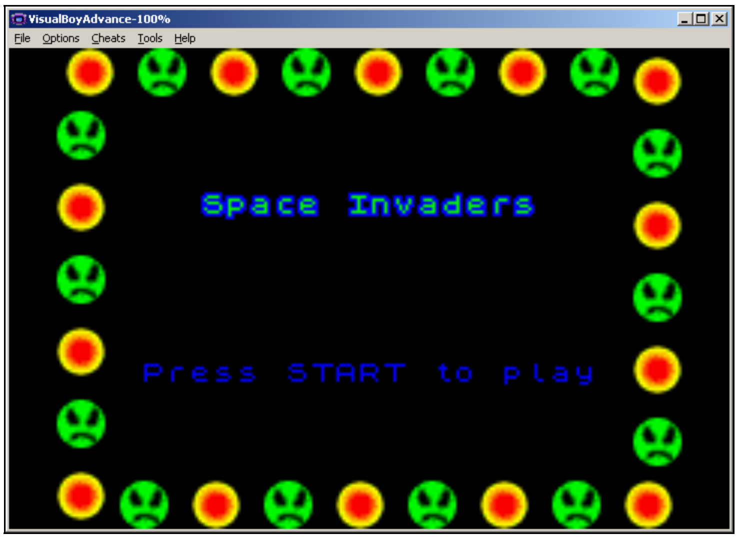


Figure 45: The splash screen

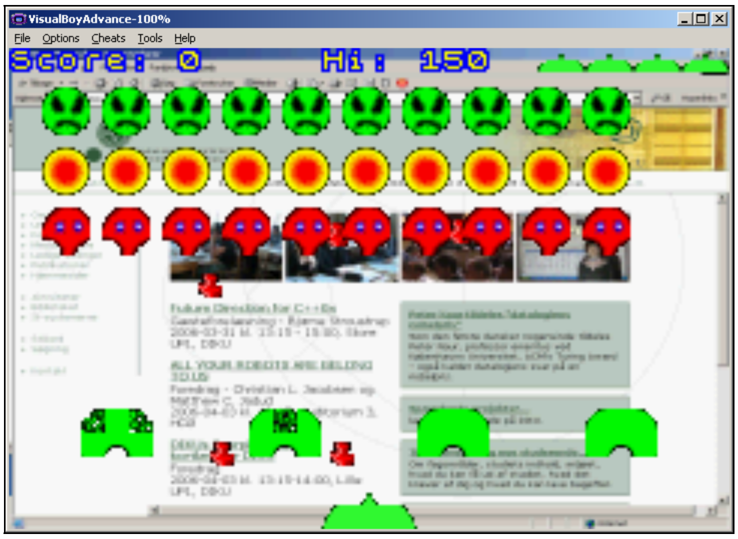


Figure 46: The aliens visit the DIKU web page and decide to invade Earth.

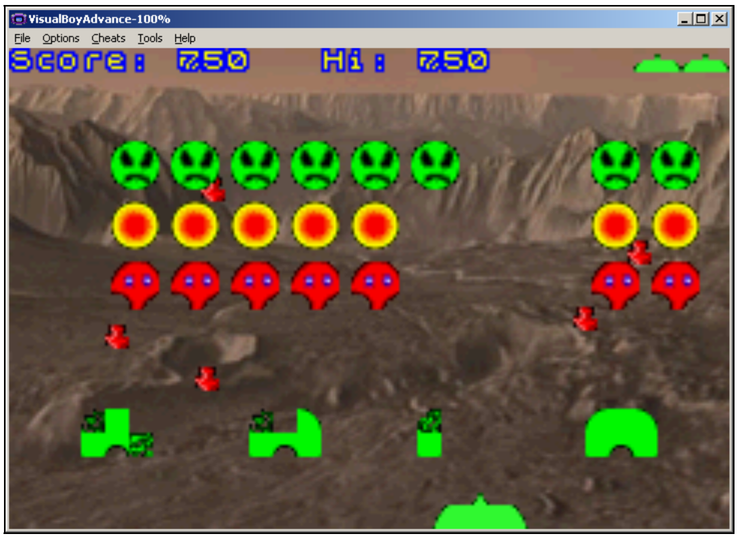


Figure 47: So they leave their home planet, Mars.

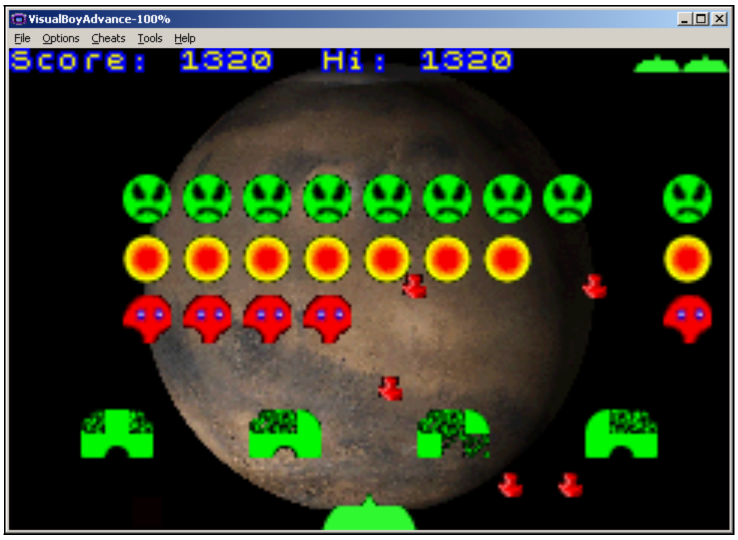


Figure 48: With a last glimpse of home, the aliens leave Mars and are heading for earth.

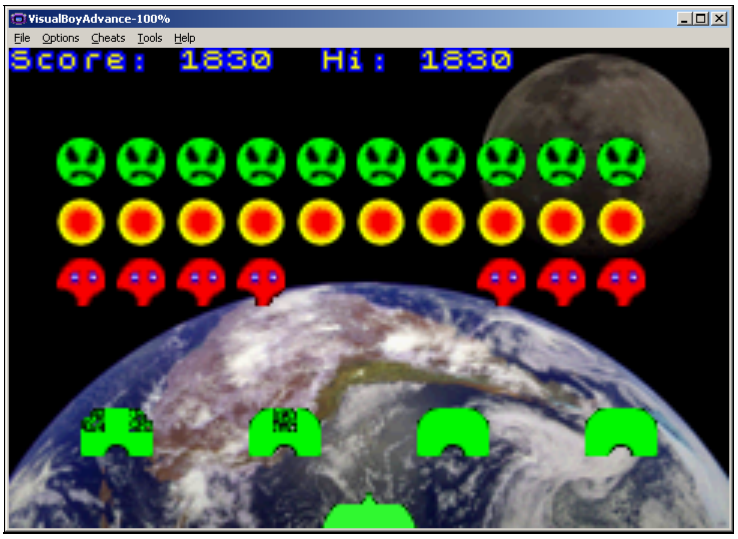


Figure 49: The aliens are past the moon, still heading for Earth.

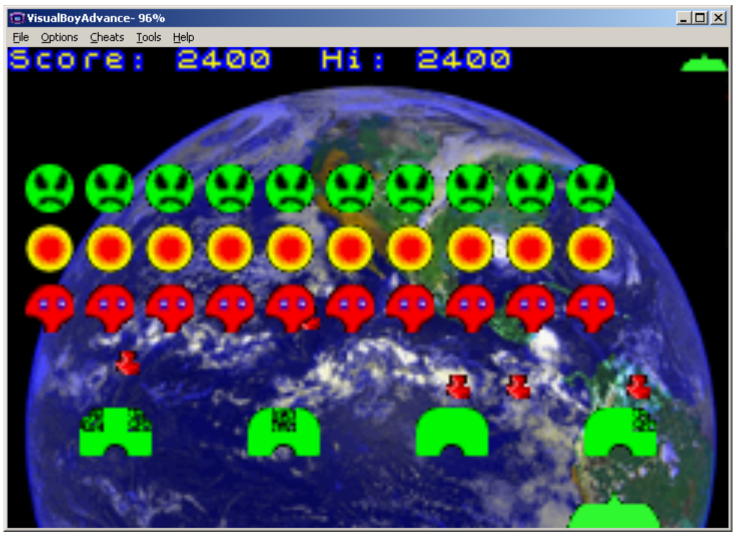


Figure 50: The aliens are getting ready to invade Earth.

The game is quite playable, although some of the graphics could be nicer and the game probably gets too difficult too fast.

7 External tools

An external tool in this context is a tool that runs on another computer than the GBA, and that can make the production of games on the GBA easier. Examples of external tools are programs that convert audio or graphics to a useable format, or it might even be a content management system.

I have found several external tools that can convert sound and graphics to useable formats, of which I have frequently used two; namely wav2gba and pcx2gba, which convert 8 bit wav sound files and images in the PCX file format to arrays in the form of C/C++ source code, respectively.

However, I have found no tool that can convert images to more than 256 colours, and for that reason, I have made two such tools.

- The first converts any image in a well-known format, such as jpg, png and bmp to a C/C++ array containing data for a 240 * 160 pixels image with 32768 colours. This program is shown in Figure 51.
- The second converts a 720 * 160 greyscale BMP-image to a C/C++ array containing data for a 240 * 160 pixels image with 32768 colours using sub-pixel conversion. The program can also save converted images in BMP-format. The principle of sub-pixel conversion is described later in this section. The program is shown in Figure 52.

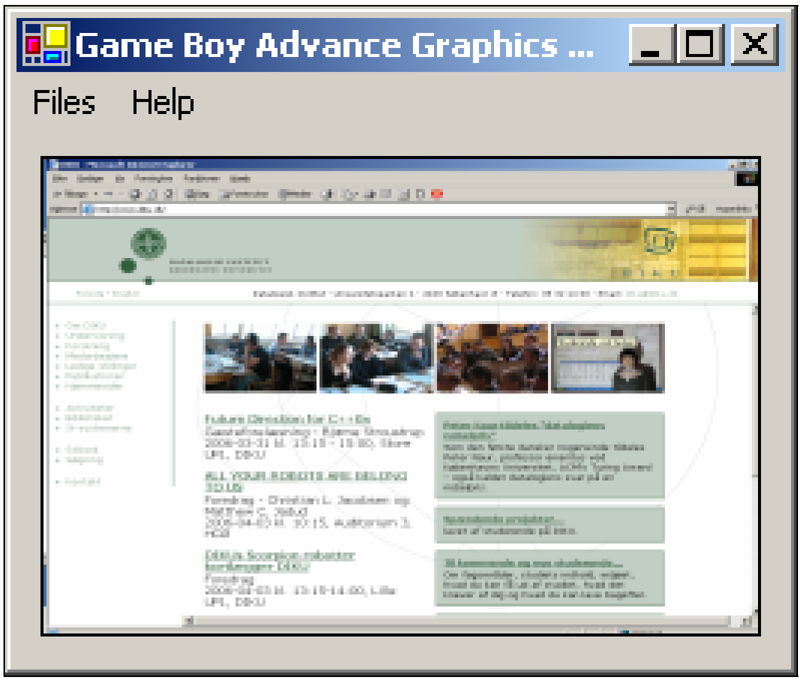


Figure 51: Screen shot of the tool that converts images to GBA-format. The screen shot shows an image of the DIKU web page being converted.

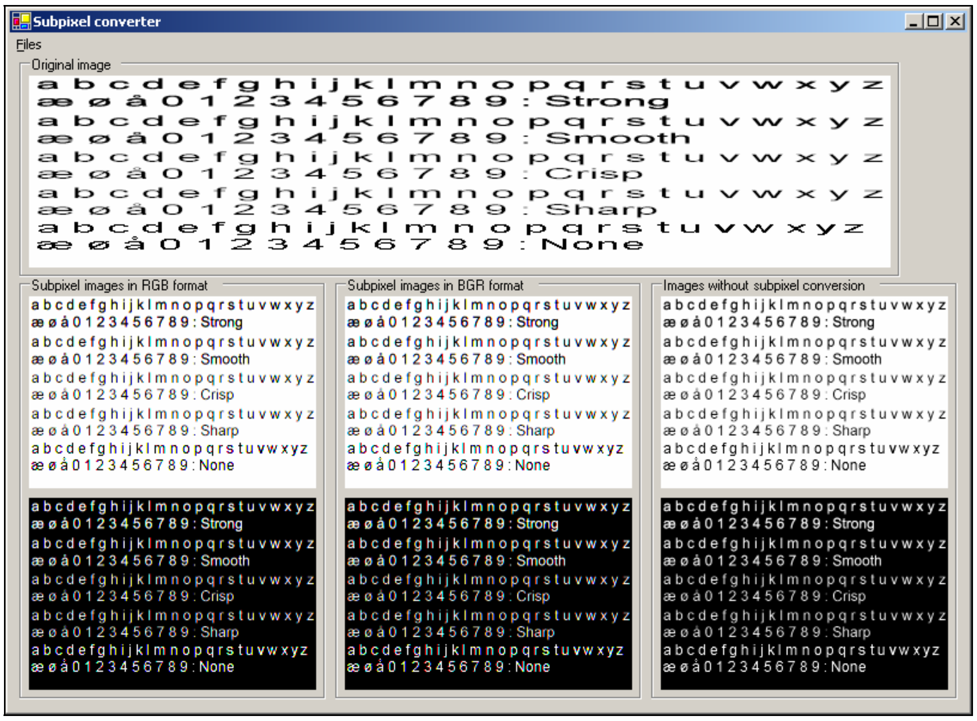

Subpixel converter
Files
Original image
Subpixel images in RGB format
Subpixel images in BGR format
Images without subpixel conversion
a b c d e f g h i j k l m n o p q r s t u v w x y z
æ ø å 0 1 2 3 4 5 6 7 8 9 : Strong
æ ø å 0 1 2 3 4 5 6 7 8 9 : Smooth
æ ø å 0 1 2 3 4 5 6 7 8 9 : Crisp
æ ø å 0 1 2 3 4 5 6 7 8 9 : Sharp
æ ø å 0 1 2 3 4 5 6 7 8 9 : None

Figure 52: A screen shot of the program that converts images to GBA-format, using sub-pixel conversion. The image at the top is converted to the six images at the bottom. At the left it is converted for the sub-pixel order red/green/blue, which is the order used by most LCD-monitors. In the middle the image is converted for the sub-pixel order blue/green/red which is the sub-pixel order used on the GBA. At the right, the image is converted without regards to sub-pixels. The converted images are also shown inverted; in order to see if that produces a better result.

Both tools are written in Microsoft® Visual Basic .NET 2003 and thus, they need Microsoft® .NET Framework to be installed on the computer. Both programs convert images to a format that is tailor-made to suit the graphics mode 3 on the GBA, which is the bitmapped graphics mode with highest resolution and colour depth.

The sub-pixel conversion tool takes advantage of the fact that on an LCD screen a pixel is made up from three separate sub-pixels (blue, red and green). The principle is illustrated in Figure 53, where two ways of drawing the same geometric figure is shown magnified. On the left, each pixel is treated as a unity, and thus the result is somewhat coarse, whereas on the right, each pixel is treated as three sub-pixels, which are handled individually, thus resulting in a much smoother transition. The method ignores the colour of the particular sub-pixel, and will thus work best with greyscale images.

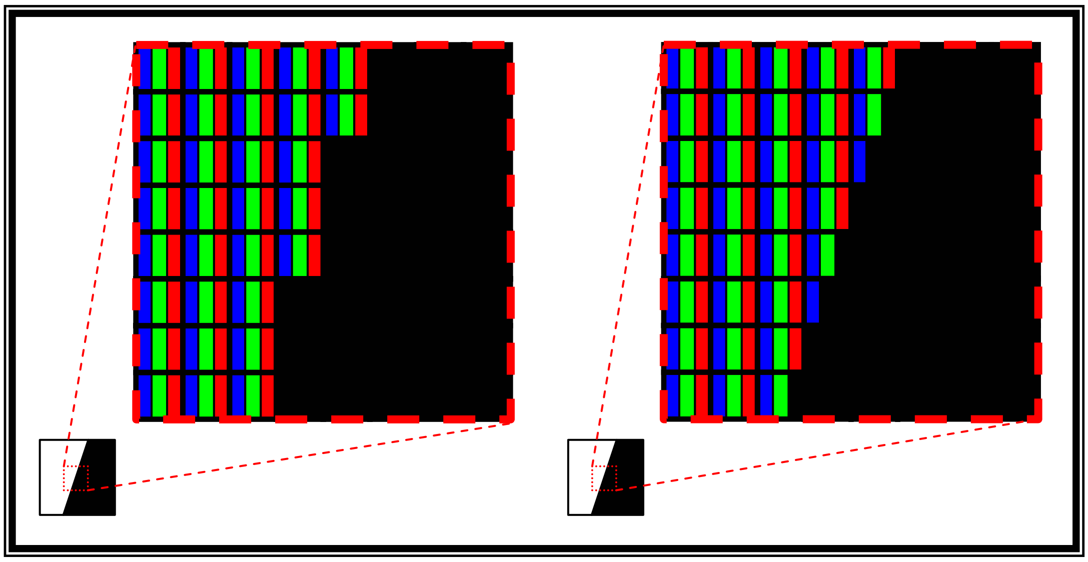


Figure 53: Illustration of sub-pixel images

Used with greyscale images, this technique produces much smoother images, compared to traditional techniques. The downside is that it only works well with grey-scale images. However, the technique works very well with images that show text, as text has a … Thus, games that display text can benefit from this technique.

Unfortunately, due to the nature of the technique, an LCD-screen is needed to see just how well it works. Naturally, this is possible, as all the programs mentioned in this chapter can be found on the enclosed CD. However, to make even simpler see the effect, I have also included some images that were generated by the sub-pixel conversion tool. They can be found on the enclosed CD.

I have not found external tools to handle content management for the GBA, nor have I made an attempt at making a content management system myself, as that would be for too big a task for this paper.

8 Data flow and brief program description

In this chapter I will briefly describe how the game engine I have made for the GBA works. It is written in C++ and consists of several classes. How the classes depend on each other is shown below in Figure 54.

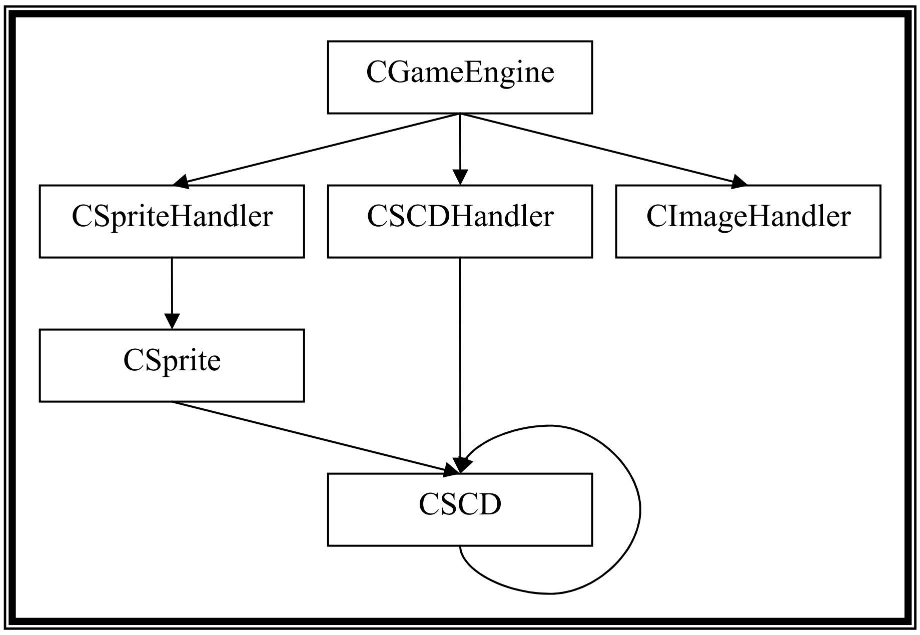


Figure 54: The relationship between the classes of the game engine

The responsibilities of each class are described below.

- **CGameEngine**. This class controls the game engine. It initializes the interrupts of the GBA, so that at each frame sprites will be moved and tests for sprite collision will be performed.
- **CSprite**. This is the Sprite class. It contains information about one sprite, such as its position, image information and a reference to a user defined function that is responsible for moving the sprite. When instructed to, usually by the CSpriteHandler class below, that function will be called.
- **CSpriteHandler**. This is the sprite handler. When instructed to, usually by the game engine, it will move all the sprites.
- **CSCD**. This is the Sprite Collision Detection class. It handles a pair of sprites and contains a reference to a user defined function that is to be called when the two sprites collide. When instructed to, usually by the CSCDHandler class below, it will test if the two sprites are colliding and if so, call the user defined function.
- **CSCDHandler**. This is the Sprite Collision Detection Handler. It handles a number of instances of the CSCD class, in order to handle several pairs of sprites that must be checked for collision.
- **CImageHandler**. This is the Image Handler class. It is responsible for preparing images, in order for them to be used with sprites.

Apart from this, the game engine depends on user defined functions, which must be provided by the game programmer. These functions handle the following:

- Initialization of the game engine at the beginning of a level, room etc. depending of the game.
- What should happen at the beginning of every frame.
- Sprite movement.
- What should happen when sprites collide.
- What should happen at the end of every frame.

Thus, a typical game will have the structure shown in Figure 55 below, where the grey boxes indicate the responsibilities of the game programmer and the white boxes make up the game engine in itself.

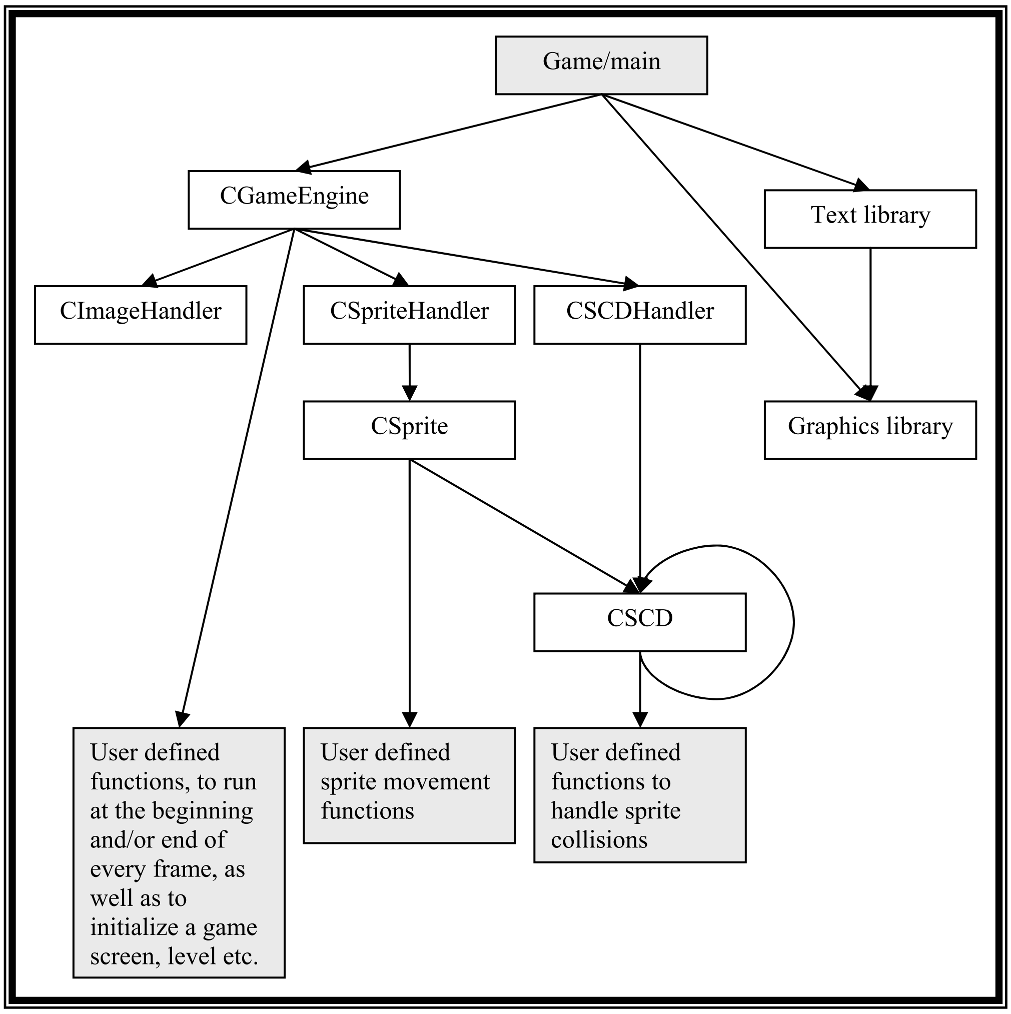

Game/main
CGameEngine
Text library
CImageHandler
CSpriteHandler
CSCDHandler
CSprite
Graphics library
CSCD
User defined functions, to run at the beginning and/or end of every frame, as well as to initialize a game screen, level etc.
User defined sprite movement functions
User defined functions to handle sprite collisions

Figure 55: The structure of a typical game made with the game engine.

For many purposes, however, the game engine can be thought of as a single unit, thus a clearer version of the diagram can be seen in Figure 56 below.

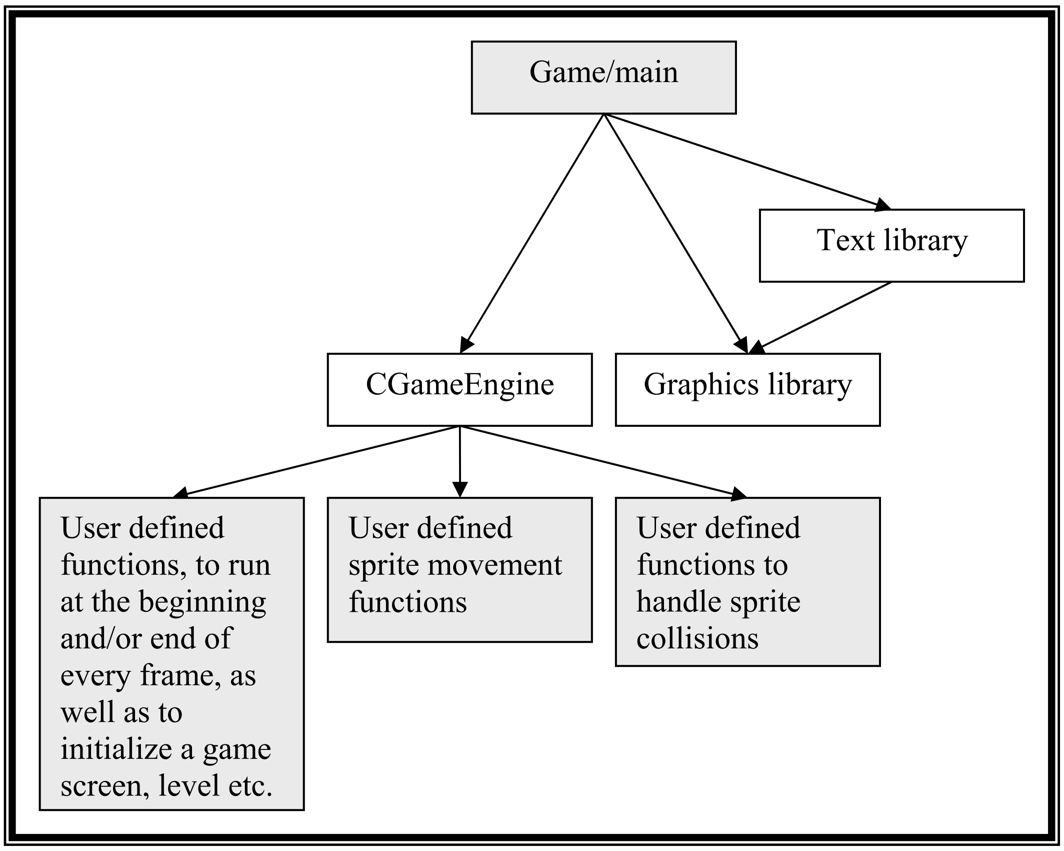

Game/main
Text library
CGameEngine
Graphics library
User defined functions, to run at the beginning and/or end of every frame, as well as to initialize a game screen, level etc.
User defined sprite movement functions
User defined functions to handle sprite collisions

Figure 56: Abbreviated version of the structure of a typical game made with the game engine.

At every frame of a game, this is what takes place:

- If the game programmer has assigned a certain pointer, RunBeforeFrame, to point to a function, that function is executed.
- For every sprite, the user defined movement function is executed.
- For all pairs of sprites that must be tested for collision, this test is performed. For all collisions of two sprites, the corresponding user defined function is executed.
- If the game programmer has assigned a certain pointer, InitScreen, to point to a function, that function is executed, after which InitScreen is reset.
- If the game programmer has assigned a certain pointer, RunAfterFrame, to point to a function, that function is executed.

In the next chapter, chapter 9, the classes and various functions of the game engine will be described, and here it will appear how to assign user defined functions to the game engine.

9 Game Programmer Instructions

In order to use the game engine to program a game, some knowledge of how the game engine works will probably be beneficial. Especially, as the game engine depends on the game programmer to supply certain functions that are responsible for some of the operations that are critical for most games, such as

- Sprite movement.
- Reactions to sprite collisions.
- Initialization of the game engine when a new level or room is entered. This function typically resets the game engine, causing all sprites to be terminated and then initialize the sprites that make up the new level or the new room.

Apart from these, user defined functions are used to determine what should take place at the beginning and end of every frame.

How to assign user defined functions to the game engine is closely related to the structure of the game engine, and thus, section 9.1 below briefly describes the various functions and data structure that make up the game engine. It is primarily intended for game programmers, who wish to use the game engine to make their own game on the GBA. As an example of how to use the game engine, please take a look at the source code for the example game, Space Invaders 2, which can be found on the enclosed CD.

9.1 The classes and libraries of the game engine

In order to use the game engine to make games, an understanding of the various classes and functions of the game engine will be helpful, particularly the parts about how to assign user defined functions to the game engine. A thorough description of the various classes and functions can be found in Appendix A, whereas this section merely offers a brief description.

- The Game Engine Class initializes the sprite handler, the sprite collision detection handler and the image handler. It is also responsible for maintaining a consistent frame rate and thus initiates sprite movement and sprite collision detection at each frame.
- The **Image Handler Class** is responsible for preparing images that are to be used with sprites.
- The **Sprite Handler Class** is responsible for a collection of sprites. It is used to add sprites to a game as well as move the various sprites.
- The **Sprite Class** is responsible for one sprite, but most functions are called through the sprite handler. It includes a method to kill a sprite.
- The **Sprite Collision Detection Handler Class** is responsible for the keeping track of a number of pairs of sprites that must regularly be tested for collision, as well as keeping track of which functions must be called when two sprites collide.
- The **Sprite Collision Detection Class** is responsible for one pair of sprites that must be tested for collision, as well as keep track of which function is to be executed when the sprites collide.
- The **Graphics Library** so far has very few functions. It can be used to draw a single pixel, clear the screen and show an image on the background screen,
- The **Text Library** includes several functions that can display text on the screen.

10 Conclusion

In order to study the concept of game engines for the Game Boy® Advance (GBA), I have got around general 2D game engines and also dealt with the capabilities of the GBA. The combination of the knowledge presented about these two areas has given me a good idea about what to include in a game engine that is specifically designed for the GBA, which has resulted in the fact that I have implemented such a game engine from scratch.

Furthermore, even though a lot of more features could have been implemented, I have shown the usability of the game engine by making an example game, Space Invaders 2. Although I am not an artistic painter, which is shown by the rather unsightly graphics in the game, it is still quite playable.

As it is one of the purposes of making a game engine in the first place, the game was made without accessing the GBA hardware directly, leaving that to be handled through the game engine. Thus, by using the game engine, it is possible to make games for the GBA without possessing GBA specific hardware knowledge.

Another purpose of making a game engine is to be able to make games faster. By making a game engine that takes care of sprite handling, sound playing and user input this has also been achieved.

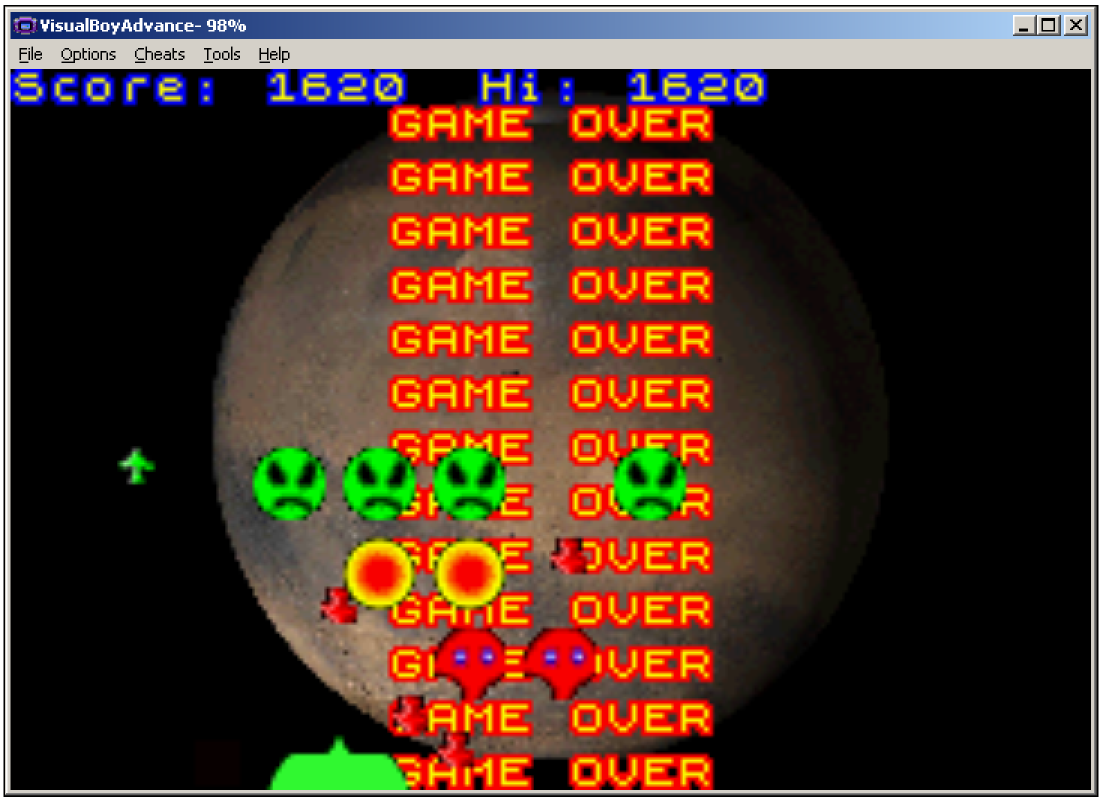
The greatest weakness of the game engine is the lack of a broad phase collision detection algorithm. This means that the game engine scales poorly when the number of sprites is increased. For this reason, I have conducted a test, where the result is to show the maximum number of available sprites at any given frame rate, under the condition that each sprite is tested for collision with every other sprite at each frame of the game. However, even without broad phase collision detection, the game engine can still be used to produce games with a high number of sprites. In the report I present some of the considerations that can made in order to secure a reasonable compromise between speed and the number of concurrent sprites or even achieve both.

In order to make background graphics for games, I have also made a tool that converts an image to a format that makes it easy to include in games that are made with the game engine.

On the more theoretical side, I have made some considerations regarding narrow phase sprite collision, i.e. a test where it is checked whether or not a pair of sprites collide. If that test is too slow, it can be preceded by a faster test, which can make the slower test unnecessary. Whether or not this is a good idea in an actual case is not always clear, but I have provided means to help decide this.

All in all, I will say that I have made a reasonably thorough going through of the aspects of game engines on the GBA, and I am particularly happy with the resulting implementation of a game engine.

At the end of this report, let me wind things up with an image from the example game, Space Invaders 2:


VisualBoyAdvance- 98%
File Options Cheats Tools Help
Score: 1620 Hi: 1620
GAME OVER

Appendix A: Classes and functions

This appendix contains a description of the various classes and functions of the game engine. Each class or function is described in a separate section, which, where appropriate, includes the following sub-sections:

- **Header file**. This is the name of the header file that needs to be included in order to use the class or function.
- **Initialization**. This will describe how the class or function is initialized.
- **Functions**. This will describe the various functions or methods,
- **User defined functions**. Some parts of the game engine rely on the game programmer to provide certain functions. What they are and how they are incorporated in the game engine will be described in these sections.
- **Data**. Variables that are important enough to be mentioned will be so in these sections, along with a description that states what they can be used for.

A.1 The Game Engine Class

The Game Engine Class takes care of the timing of the game, i.e. it makes sure that sprites are only moved between screen refreshes. Here the game engine class will briefly be described.

A.1.1 Header file

- GameEngine.hpp

A.1.2 Initialization

To initialize the game engine, use:

- CGameEngine* GameEngine;
  GameEngine = new CGameEngine();

This will automatically initialize the Sprite Handler Class, Sprite Collision Detection Handler Class and the Image Handler Class. They can be reached directly through

- CGameEngine->SpriteHandler
- CGameEngine->SCDHandler
- CGameEngine->ImageHandler

A reset of the game engine will result in the destruction of the SpriteHandler, the SCDHandler and the ImageHandler instances, and thus the reset function should only be used in a InitScreen function as described in section A.1.3 below, as the behaviour of the game engine will otherwise be undefined. In particular, use of the reset function in the user defined sprite movement functions and the user defined functions to handle sprite collisions should be avoided. The game engine can be reset through the following statement.

- GameEngine->Reset();

The game engine can be set to halt, either until started again or for a given number of frames. These functions do the jobs:

- GameEngine->Pause()
- GameEngine->Pause(Length)
- GameEngine->Play()

where Length is the length of the pause measured in frames.

A sampled sound can be played. To do so, use

- GameEngine->PlaySound(*sound, length, delay);

where

- sound is a pointer to the sampled sound.
- length is the length of the sound sample measured in frames.
- delay is a delay value that must be calculated by this formulae:

$$65536 - \frac{16777216}{\textit{sample frequency}}.$$

A.1.3 User defined functions

A function can be provided by the game programmer to re-initialize the game during game play, e.g. when entering a new room or level. This is done by assigning the address of a function to the pointer InitScreen. At different stages of the game, the InitScreen pointer can be set to point to different functions, after the wishes of the game programmer. When InitScreen is set to point to a function, that function will be executed at the end of the frame. When this function call has finished, InitScreen will be reset, making sure that the function is called only once. Notice that the use of **GameEngine->Reset()** is safe from within this user defined function. To assign a function to the InitScreen pointer, use

- GameEngine->InitScreen = somefunction;

where somefunction is the function provided by the game programmer.

Two special user defined functions are available, namely functions pointed to by the pointers RunBeforeFrame and RunAfterFrame. They can be set to point to functions provided by the game programmer and those functions will then be called at the beginning and end of every frame, respectively. To assign a function, use

- GameEngine->BeforeFrame = somefunction;
- GameEngine->RunAfterFrame = somefunction;

where somefunction is a function provided by the game programmer.

A.1.4 Data

The current frame number can be accessed through this unsigned integer number

- GameEngine->FrameNumber

A.2 The Image Handler Class

This class prepares images for use with sprites. In order to convert sprite images to program code that is compatible with the Image Handler Class, use a program such as PCX2GBA.EXE. This program can be found on the enclosed CD.

A.2.1 Header file

- ImageHandler.hpp

A.2.2 Initialization

This class is automatically initialized when initializing the Game Engine Class.

A.2.3 Functions

To load a sprite image for later use with a sprite, use

- ImageIndex = GameEngine->ImageHandler->LoadImage(ImageData, Size);

where

- **ImageIndex** is an integer, which must be used later, when assigning the image to the sprite.
- **ImageData** is an array representing the image, which is typically made by an appropriate program such as PCX2GBA.EXE.
- **Size** is a value indicating the proportions of the sprite image. The possibilities are SIZE_64, SIZE_32, SIZE_16 and SIZE_8, which are constant values representing quadratic sprites of the obvious sizes, measured in pixels.

In order to get the colours of the sprite image right, the colour palette must be loaded. Notice that the palette must be the same for all concurrently used images, which means that the palette need only be initialized once independently of the number of sprite images used. This is done through

- GameEngine->ImageHandler->LoadPalette(ImagePalette);

where **ImagePalette** is an array representing the sprite image's colour palette, which is typically generated together with the sprite image.

When all sprite images are to be removed, e.g. to make room for new ones, this function must be used:

- GameEngine->ImageHandler->Reset(void);

A.2.4 Data

Sprite images are stored in tiles, of which there are up to 1024 available. The number of currently used tiles can be checked using this integer

- GameEngine->ImageHandler->TileIndex;

In some screen modes, e.g. mode 3, not all tiles are available. In that case, TileIndex must be manipulated directly before sprite images are loaded, by setting

- GameEngine->ImageHandler->TileIndex = 512;

A.3 The Sprite Handler Class

This class handles a collection of all the sprites in the game engine.

A.3.1 Header file

- SpriteHandler.hpp

A.3.2 Initialization

This class is automatically initialized when initializing the Game Engine Class. When the Sprite Handler Class is initialized, 128 sprites of the Sprite Class are automatically initialized. See section A.4 for further information on the Sprite Class.

A.3.3 Functions

To add a sprite, use the following

- GameEngine->SpriteHandler->AddSprite(x, y, Min, Max, Speed, TileIndex, SpriteSize, MoveFunc);

where

- **x** and **y** are the initial location of the sprite. Notice that these values must be 128 times greater than the pixels they represent on the screen. The reason for this is to make it possible to work with movement steps that are smaller than one pixel.
- **Min**, **Max** and **Speed** provided as extra variables, the game programmer can use in order to manipulate the sprite.
- **TileIndex** is the ImageIndex number retrieved when initializing the sprite image through the LoadImage described in section A.2.3.
- **SpriteSize** must be the Size number that was used when loading the sprite image as described in section A.2.3. Otherwise, if SpriteSize is smaller than the original Size, the sprite will use only the top/left part of the sprite image. If SpriteSize is greater than the original Size, the sprite's image will be made up from several images, ordered in a way that will probably make it appear as garbage.
- **MoveFunc** is a function that must be provided by the game programmer. It is described below, in section A.4.4.

The Sprite Handler can be reset, thus resulting in all sprites being removed. To achieve this, use

- GameEngine->SpriteHandler->Reset();

A.3.4 Data

All sprites can be accessed manually through an array of instances of the Sprite Class, called

- GameEngine->SpriteHandler->sprites[Index];

where **Index** must be a value between 0 and 127, both included.

The number of the next available sprite can be attained through this integer variable

- GameEngine->SpriteHandler->NextSprite

A.4 The Sprite Class

The Sprite Class is responsible for handling of a single sprite.

A.4.1 Header file

- Sprite.hpp

A.4.2 Initialization

128 instances of this class are automatically initialized when initializing the Sprite Handler Class.

A.4.3 Functions

Obviously, the Sprite Class includes several functions. However, except one they are all executed automatically and there is no need to access them directly. The one remaining function is used to kill a sprite, meaning that it will disappear from the screen and not be moved anymore. A sprite is killed by the call of

- Sprite->Kill();

where **Sprite** is an instance of the Sprite Class.

A.4.4 User defined functions

Movement of a sprite is handled through a user defined function that must be provided by the game programmer when the sprite is added through the Sprite Handler. The function can, however, be changed manually through

- Sprite->Move = MoveFunc

where

- **Sprite** is an instance of the Sprite Class
- **MoveFunc** is the new function that should henceforth be responsible for the movement of that sprite.

The prototype of the MoveFunc function must be like this:

- void MoveFunction(CSprite* Sprite);

The MoveFunc function for each sprite is called automatically at each frame and in this function the location of the sprite can be changed by manipulating these variables

- Sprite->x
- Sprite->y

Notice that the value of x and y must be 128 times greater than the desired position on the screen when measured in pixels, in order to work with movement smaller than one pixel.

A.4.5 Data

To check whether or not a sprite has been killed, access this Boolean variable

- Sprite->Alive

A.5 The Sprite Collision Detection Handler Class

This class is responsible for testing which sprites collide with which other sprites and take the proper actions when sprites collide. In order to achieve this, game programmer must define which pairs of sprites should be tested for collision, and also provide a function that should be called when the two sprites collide. At each frame the Sprite Collision Detection Class will check all defined pairs of sprites and call the relevant functions.

A.5.1 Header file

SCDHandler.hpp

A.5.2 Initialization

This class is automatically initialized when initializing the Game Engine Class.

A.5.3 Functions

To define a pair of sprites that should be tested for collision, use

- GameEngine->SCDHandler->AddSCD(Sprite1, Sprite2, CollisionFunction);

where

- **Sprite1** and **Sprite2** are instances of the Sprite Class
- **CollisionFunction** is the name of the function that will be called if and when Sprite1 and Sprite2 collide. The function has this prototype:
    - void MoveFunction(CSprite* Sprite1, CSprite* Sprite2);

All definitions of pairs of sprites can be erased by calling

- GameEngine->SCDHandler->Reset();

A.6 The Sprite Collision Detection Class

This class maintains the handling of one pair of sprites that is to be checked for collision. It includes the function that is responsible for testing whether or not two sprites are colliding. However, neither any of these functions nor the data in the class are intended for manual manipulation, thus making further explanation unnecessary.

A.7 The Graphics Library

The game engine comes with a very simple graphics library. It only operates in graphics mode 3, which is the bitmap graphics mode with the highest resolution, i.e. 240*160 pixels, and colours of the highest bit depth, namely 15, but with no back buffer capabilities.

A.7.1 Header file

- Graphics.hpp

A.7.2 Functions

The Graphic Library includes three functions that can set a pixel on the screen, clear the screen and load an image respectively. To set a single pixel, use

- SetPixel(x, y, color)

where

- **x** and **y** point out the pixel to be set.
- **color** is an integer value representing the colour to be set.

To clear the screen, use

- ClearScreen(color)

where **color** is an integer value representing the colour to be set.

To show an image on the background of the screen, use

- LoadImage(Image)

where **Image** is an array of integers, e.g. made by the "Game Boy Advance Graphics Mode 3 Background Conversion Tool" as described in section 7.

A.8 The Text Library

Several text functions have been implemented, including functions that can display a single character, a 0-terminated string and signed integers. These functions each come in two variants, i.e. with either one or two colours. When two colours are used, the text being written will be of one colour and the background will be of the other. When only one colour is used, the background will remain unchanged.

A.8.1 Header file

- Text.hpp

A.8.2 Functions

To display a single character, a 0-terminated string or a signed integer in decimal format, use one of these

- printChar(x, y, char, color)
- printChar(x, y, char, inkColor, paperColor)

- printString(x, y, text, color)
- printString(x, y, text, inkColor, paperColor)
- printDec(x, y, num, color)
- printDec(x, y, num, inkColor, paperColor)

where

- **x** and **y** are the location on the screen where the text should be displayed.
- **char** is the character to be displayed.
- **text** is the text to be displayed.
- **color** and **inkColor** are the colour of the text that should be displayed.
- **paperColor** is the colour that the background of the text is changed to.
- **num** is the signed integer to be displayed.

Appendix B: Vocabulary

This appendix contains words and phrases that might need some explanatory remarks.

- **Screen refresh**. The screen refresh is the process of drawing everything on the physical screen. On the GBA the Screen refresh is handled automatically by the hardware and occurs approximately 60 times a second.
- **Screen update**. This is the process of changing the graphics on the GBA screen, such as moving the sprites and changing the background. Notice that these changes will not be visible until the next **Screen refresh**. Updating the screen during the Screen refresh might result in tearing and should be avoided.
- **Tearing**. This is an undesired effect that is the possible result of performing a **Screen update** during a **Screen refresh**. It happens when the graphics are changed at a time when only part it been refreshed, thus resulting in the effect that the graphics appear to be torn apart. This is illustrated in Figure 14, on page 15.
- **Game Programmer**. The user of the game engine, i.e. the person using the game engine to writing a game.

Appendix C: Bibliography

[FOLEY] Foley, James D., et al., *Computer Graphics: Principles and Practice - Second Edition in C*, Addison Wesley, Reading, 1996.

[HARBOUR1] Jonathan S. Harbour, *Programming The Nintendo® Game Boy® Advance: The Unofficial Guide*. Never published, but was supposed to be so by Premier Press, 2003.

[LOBÃO] Alexandre Santos Lobão et al., *.NET Game Programming with DirectX 9.0*, Apress, The Author's Press™, United States of America, 2003.

[MORRISON] Morrison, Michael, *Beginning Game Programming*, Sams Publishing, Indianapolis, 2005.

Appendix D: Web sources

[APEX] http://www.apex-designs.net/tools_aas.html. Home of the Apex Audio System, which is a sound mixer for the GBA.
[EMUBASE] http://nocash.emubase.de/gba.htm. Home of the No$GBA emulator.
[EB] http://ekstrabladet.dk/erhverv/digitalt/article210393.ece. The web site of the Danish newspaper, Ekstra Bladet, mentioning the game about Zinedine Zidane.
[GAMEMAKER] http://www.gamemaker.nl/. Home of Game Maker by Mark Overmars.
[GAMESECTION] http://www.gamesection.dk/nyheder.asp?nid=1686. A web site concerned with computer games. This web page mentions the game about Zinedine Zidane.
[GBADEV] http://www.gbadev.org/. A site dedicated to sharing developing information about the Game Boy® Advance.
[GBADEV1] http://www.gbadev.org/tools.php?showinfo=66. Link to "GBA Project appwizard" that makes it possible to make Game Boy® Advance projects in Visual Studio.
[GBADEVRS] http://www.devrs.com/gba/. A web site for Game Boy® Advance developers.
[GBATEK] http://nocash.emubase.de/gbatek.htm. A thorough specification of the GBA hardware.
[HARHOUR2] http://www.jharbour.com. Personal web-page of Jonathan S. Harbour, author of *Programming The Nintendo® Game Boy® Advance: The Unofficial Guide*.
[HYG] http://www.hyggestedet.dk/index.asp?game=zidane. A web-page that hosts, among other things, some small games; including one based on Zinedine Zidane butting an opponent in the WM final of soccer, 2006.
[KRAWALL] http://synk.at/krawall/. Home of the Krawall Advance, which is a sound mixer for the GBA.
[LIK-SANG] http://www.lik-sang.com. A site selling, among other things, the MBV2 cable for connecting a GBA to a PC, and thus be able to transfer small programs (max. 256 KB) to the GBA.
[NGEMU] http://vba.ngemu.com/. Home of the Visual Boy Advance emulator.
[NGINE] http://www.ngine.de/. Home of the development kit, HAM, for the Game Boy® Advance.
[NINTENDO1] http://www.nintendo-europe.com/NOE/en/GB/system/gba_topic2.jsp. Nintendo®'s official site for Europe.
[NINTENDO2] http://www.nintendo.com/gamelist?sf=Game+Boy+Advance. Nintendo® of America's official web site. This particular web page shows a list of games for the GBA.
[NOCTURNAL] http://www.nocturnal-central.com/catapult.php. Homepage for the commercial system, "Catapult Development System", for making games on the GBA. The system comes with its own language.
[SF1] http://devkitadv.sourceforge.net/. DevKit Advance's web-page at SourceForge.
[WIDELEC] http://widelec.org/zidane.html. One of the web-places that hosts the Zidane game.
[ZOPHAR1] http://www.zophar.net/gba.html. This sub-page contains Game Boy® Advance emulators for various systems.

Appendix E: The Contents of the Enclosed CD

This report comes with an enclosed CD, in order to supply the reader with source code and programs used throughout the report. The folder structure of the CD is reflected in the structure below.

E.1 MyOwn

This folder contains the material that I have made as part of this project.

E.1.1 External Tools

This folder contains the two programs that I have made to convert images to C/C++ arrays that can easily be incorporated in a GBA project.

E.1.1.1 Executables

This folder contains the executable version of the two programs.

E.1.1.2 Visual Studio .NET 2003 Project Files

This folder contains the Visual Studio project files for the two programs.

E.1.2 Text

This folder contains this report in PDF file format.

E.1.2.1 Easy2Read Source Code

This folder contains all the source code files in a format prepared for reading, as well as an index that should make it easy to find and view the wanted source code files. Most of the source code files in this folder are coloured in order to make them easier to read. The index can be seen by opening the file `\MyOwn\Easy2Read Source Code\index.html` in an ordinary browser.

E.1.3 GBA programs

This folder contains two GBA programs that can be executed either by using an emulator or by transferring them to a GBA unit. The two programs are the example game Space Invaders 2 and the Performance and Scale test program described in chapter 5.

E.1.4 GBA Source Code

This folder contains source code for the game engine, the Space Invaders 2 game and the Performance and Scale testing program. In addition to this, the computer generated source code files containing sounds and graphics are also included.

E.1.5 Miscellaneous Images

This folder contains the various graphical images used throughout the project., as well as some sup-pixel images as described in chapter 7.

E.2 3rd party material

This folder contains material made by others and which has been used during this project.

E.2.1 Development Environments

This folder contains installation files for Game Maker 6.1 and HAM 2.80 including Visual HAM.

E.2.2 External Tools

This folder contains the two programs, wav2gba and pcx2gba, that are used to convert sounds and graphics to be used on the GBA, respectively.

E.2.3 GBA Emulators

This folder contains installation files for the two GBA emulators described in section 2.4.1.