Default

Paul Manias

COLLABORATORS			
	TITLE : Default		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Paul Manias	June 4, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Default

1.1 The Games Master System V0.5B

T H E G A M E S M A S T E R S Y S T E M

BY PAUL MANIAS

VERSION 0.5B

GENERAL DOCUMENTATION

Welcome to the latest version of GMS! It's been about 3 months since the last update, so you can expect lots of new features in this version. This includes multiple bobs, CPU assisted blitting, resource/memory tracking, external object files, updated docs, and lots of other bits and pieces. I removed GMSPrefs in this release, it will now appear on Aminet in a few weeks from now, hopefully in a 100% working state.

Remember, if you make any demos then I would like to see them, just mail them to me at my e-mail address so I can see what you are doing. There is also a questionnaire in this archive and I would like you to fill it out and return it to me. This is quite important as it guides the future of GMS in many areas.

Other than that, have fun with your programming and look out for V0.6B in 2 or 3 months!

1.2 Introduction to the Games Master System

This introductory text is taken directly from the web pages:

http://www.compkarori.co.nz/amiga/gameslib/

WHAT IS THE GAMES MASTER SYSTEM?

The Games Master System is a solution to one of the biggest problems the Amiga community has ever faced. What problem? Games support in the OS! Windows has it, the Apple has it, and the poor old Amiga has none. This lack of support has brought up some quite serious problems when it comes to hardware compatibility in games. Every Amiga owner encounters them at some time, if not often. There have been efforts made towards libraries specifically tailored to handling graphics cards, but these are missing the big picture.

THE BIG PICTURE

Hardware bashing is fast, sometimes many times as fast as an equivalent system legal routine. However hardware bashing is not very compatible with other hardware types, resulting in unpredictable results on other systems. No programmer can be expected to support dozens of obscure hardware configurations... just look at what it did to the PC, you can't install a game without going through and setting IRQs for sound cards. Let's face it, if the problem isn't dealt with directly, then the situation is just going to get worse as new hardware is released.

So how are we going to deal with this ever increasing problem? No-one has ever come up with an acceptable solution for our little Amiga. Maybe we are getting close to retargettable graphics, but what about retargettable sound? What about user support? What about networking? Where's our support for all those different joysticks? Where's the real support for writing games in an OS? What about all the other stuff people keep moaning about? Well after a lot of design and investigation into this, a solution is already on its way. It's a clever little system that everyone will come to know as...

THE GAMES MASTER SYSTEM

The "Games Master System" solves all the problems developers have faced in the past, and are still facing today. Project GMS encompasses not only the games.library but also all the GPI's, documentation, developer information, and most importantly the user prefences program (GMS Prefs). All these things have been designed to work together and will achieve the following:

* Erradication of the need to bash the hardware from within games.
* Make it easier to migrate from the current Amigas to the new Power (or whatever) Amigas.
* Make games programming, easier, faster, and more productive.

* Give users the ability to modify any game to suit their requirements.

HOW DOES IT WORK?

First lets look at the way games are written on the Amiga today. A lot of experienced coders would tell you that a lot of their source is automated, put into macros, drawing routines, sound routines, and so it goes on. Rather interestingly, this is usually the same stuff that is doing all the hardware hacking.

By taking these commonly used routines and putting them into a library we remove the hardware compatibility problem immediately. Unfortunately for us, although it works this method is just not fast enough for the speed that games require. Enter the use of GPI's!

A GPI (short for "Games Programming Interface") is a collection of library functions specifically designed to perform a task for just one hardware device. Each GPI can therefore be built to do whatever it wants without having to worry about compatibility problems. New versions of an existing GPI can be written for different hardware devices, so graphics cards, sound cards, 3D chips etc can be supported.

The benefit of this is that the user gets the best possible speed, while the programmer can simply utilise the built-in routines and know that not only will their game be fast, but also be compatible with Amigas everywhere.

USER SUPPORT

If you're a games player then there have probably been a number of times where you've thought "Why can't I use my Sega Joypad?", "I want to run in a DBLPAL screen!", "I want multi-tasking!", "I want to turn the music off!", "I want, I want, I want!". Unfortunately, that's what it sounds like to developers who don't have time to support all these things!

Luckily an important feature of the Games Master System is the level of support given to the user. The GMS Prefs program allows the user to select levels of mode promotion, type of joystick used, vector detail, networking, C2P routines, music re-direction, task priorities, and much much more. This solves a lot of the moans and gripes that users have had in the past, and since this is all transparent to the programmer, user support is easily achieved. Hopefully this news will make you all very happy!

ANYTHING ELSE?

Look around the rest of the site to get more detail on the things mentioned here, and other things that we haven't gone into yet. The GMS binaries and documents are available on Aminet, in dev/misc/gms_dev.lha. Remember to ask us if you have any questions about the project!

1.3 OverView

OVERVIEW OF THE GAMES MASTER SYSTEM

Project GMS started in the beginning of April 1996, in an effort to provide games support in the Amiga OS. The overall aim is to write the best games interface we possibly can, which should eventuate into a system that everyone can enjoy. The prime objectives of GMS are:

1. To erradicate the need to bash the hardware from within games.
2. To make it easier to migrate from the current Amigas to the new Power Amigas.
3. To make games programming easier, faster, and more productive.

GMS has been designed to be fully extendible in ways that will make future improvements very easy to implement. The system is split into a number of sub-sections: The master library, the GPI's, the debugger, and the preferences program. This is further enhanced by identifiable data structures, which allow us to write new structures in future, without overhauling the functions.

The library itself serves as a "skeleton" of function calls, rather than having all the code residing within the library itself. The library is "filled out" on opening by the use of Games Programming Interfaces (GPI's). A GPI consists of a group of function calls pertaining to one particular area of games programming. For example, the Screens.GPI contains routines for opening and closing a screen, altering the screen's palette, and placing sprites onto a screen. Other GPI's are purely optional, and must be initialised before use. This saves a lot of memory as you only have to load in what you need.

GPI's can be written and re-written for any hardware setup imaginable. It is possible to split them up into even smaller files if desirable. An example of this is the Screens.GPI's Chunky 2 Planar routines, which can give the user a variety of options for his system.

To see how everything links together, view the Communication.iff file to see a graphic representation of the links and connections now.

There will of course be a version of the games.library for the new Amiga's, so an AGA game written now could still work on these new machines! The great thing is that this library really closes the gap between differences in machine architecture. Even games intended for a PPC could work on an Amiga with CyberGfx card, granted that you compiled a 680x0 conversion.

In the future I hope to have available:

* Support for graphics cards.
* Support for sound cards.
* Vector and 3D functions.
* Support for all the new hardware capabilities in PPC Amigas.

This is looking quite some time into the future of course, but GMS is designed for these things so it's all quite possible.

1.4 Questions and Answers

QUESTIONS AND ANSWERS

I often get mail from people asking me questions about what you can and can't do in GMS. Here I will show you the answers to some of these questions, and hopefully this way everyone can benefit in learning more about how GMS works. If you have any questions you can mail them to me and they may appear here.

[Currently I've only put one question here, I'll have to look through my e-mail and get some more!]

BLITTING

Q. If I want to have 5 bobs with the same graphics, may I initialise the first with NULL in bob.maskdata and GENMASK set, then copy the pointer created to the other structures and init them with GENMASK cleared? Otherwise the masks would be created again several times and waste memory.

A. Yes, it is legal to copy masks generated in one BOB over to another BOB. Just remember when you free the "master" BOB, all BOB's containing pointers to its masks will become invalid. For this reason make sure that you free the master BOB last.

   Note: If your framelist contains graphics which have identical masks, this will be detected when your masks are generated. Any mask which is discovered to be identical to another mask in the framelist is automatically discarded and will be replaced with a pointer to the existing mask. This will save you memory without effort on your part.

1.5 I want to write a really awesome-cool-funky-thingofa GPI.

WRITING NEW GPI'S

Anyone can write a GPI, but the first thing you must do is email me about your idea and describe what it will do. I will then consider wether or not a whole new GPI should be created, or if your idea should be added to an existing GPI. It may turn out that someone else has the same idea as you, in which case I will put you in contact with that person.

If in the event a whole new GPI needs to be created, I will first ask you to list all the possible functions you think will be needed. Once that is figured out we will design the structures that will be required - usually there will only be one of these, so it's important that we get it right.

You will then be sent a GPI development kit which details how to integrate your GPI with the games.library. GPI's are based on normal system libraries, but have the addition of some code supporting function remapping. Later when you have written the GPI you have to send it to me and I will suggest any changes to the functions etc. Once everything is

okay, then it's an official GPI.

The only other condition that you must adhere to is that your GPI is written in assembler, or very good C. Also, it helps to send me the souuce at some point so that if you lost interest later on, I could still update the GPI when necessary.

Remember there may be a lot of people using your GPI so I must ensure that it's 100% OK and can be upgraded for future Amigas.

1.6 Really cool features!

CURRENT FEATURES OF THE GAMES MASTER SYSTEM

This is just a summary of the major features that have so far been implemented. Not all new features and changes have been documented here. For the complete low-down on all features of GMS check the developer information files.

NEW!
* Support for Multiple BOB image structures (MBOB's) for blitting many images from one structure. Makes allowances for structure mutations (for storing extra program data with image entries).

NEW!
* CPU assisted drawing with blitter functions, drawing speed is no longer limited to blitter throughput. Added optional mask generation for BOBs on intitialisation.

NEW!
* Internal resource tracking on memory allocations. GMS will free resources that you have not deallocated when your program closes the library. Warnings are given to help you fix any problems.

NEW!
* Structure and object pre-processing, for compiling structures before run-time. This makes GMS the first system to support up to 100% user editing of game data.

* Fast blitter functions for drawing BOB's, copying for screen buffers, 3 different screen clears, auto background saving and clearing for BOB's. Also includes Pixel and Line drawing functions, and support for list's for very fast mass-drawing operations.

* Full sprite support, and that is: All available sprite dimensions, colour table offsets, 16 colour sprites, width-doubled sprites, full animation support, lo-res, hi-res, and superhi-res support,

* Sound support includes: Support for sound priorities, intelligent dynamic channel play-back, channel modulation for special effects, IFF support.

* Proportional colour fading, functions are: PaletteMorph, ColourMorph,

PaletteToColour and ColourToPalette. Support for setting speed and colour ranges.

* Full support for raster/copperlists, with effects such as: ColourLists, Mirror, Flood, Screen and Sprite Splitting, and Palette Changes.

* Allows you to support all different kinds of input devices (joysticks, joypads, mouse etc) through just one simple function call. This enables you to support devices that don't even exist yet.

* User preferences program to allow full configuration of a game's functionality. This includes configuration for: Game/Task Priorities, Choice of networking, Mode Promotion, Joystick Config, Music Redirection, and more.

* Stable memory allocation and a freemem routine that will not crash your machine if you have written over your memory boundaries.

* Smart Saving and Loading of files, with automatic packing and depacking. Packer support covers files crunched with XPK (external), PowerPacker (internal), and RNC methods 1&2 (internal).

* 320k of assembler, E and C sources, demonstrating all uses of the library.

* All games can multi-task with no significant drop in speed or performance.

1.7 What language to use with GMS?

GMS LANGUAGES

As GMS is no more than an extension to the OS, it can work with any language that you want it to. Currently supported languages in this archive are:

Assembler
C/C++
E

You could also use Blitz Basic, Pascal, Oberon and others if you know how, but I currently don't have any source demos or special include files to help you with those.

If you're new to programming then I would recommend starting out with C or E. In my opinion E is a little easier on the beginner, but C is more common place in other areas and you might find that useful. If you don't know which one to choose, try learning E first, and then C as they are quite similar languages.

If you want to write really fast games, you will have to learn assembler. With GMS learning assembler is quite easy, as you don't have to think about programming the hardware registers. Look at the demo sources and make up your own mind if you want to learn it or not. Using GMS you could become a fairly adequate assembly programmer in as little as 2 months if you have

come from something like C.

WHAT COMPILER?

If you know what language you want to use, you will have to think about what compilers you should get. You can't program without a compiler! Here is my opinion on the most common and best compilers available:

Assemblers

The best assemblers are AsmOne, DevPac and PhxAss. I have all three of these and use each one of them for different situations. You don't need that many, but two of these are free, so it won't cost you anything.

AsmOne has an excellent source-level debugger and I recommend it to beginners, as you can observe how the 680x0 instructions work. I don't use it that much today, but it is useful and has some features that make it very easy to use. It also has the fastest compiler speeds that you could imagine. I got the latest AsmOne from the WWW, go to one of the Amiga Web searches and look for "AsmOne" to find it.

DevPac is a good, robust compiler with many options, but it's a little slow and hasn't been updated in a while. I recently moved to using PhxAss for these reasons. You can get DevPac from HiSoft and other software dealers, it's a commercial product so you will have to pay for it.

I just got PhxAss a few days ago and have so far found it to be a very impressive asm compiler. It is compatible with DevPac sources and has very good compiling times. The package is regularly upgraded and it's freeware. Good work Frank Wille! You can get PhxAss from Aminet, just download it as dev/asm/Phx*.lha.

You will also need a text editor if you're using DevPac or PhxAss, I recommend CygnusEd as it's small and you can alter the TAB stops. This feature is important as it keeps your sources easier to write and manage. You will notice that all my assembler sources look strange unless you view them with CED or AsmOne. CygnusEd hasn't been updated for a while, but until I find another good text editor with this feature, I'm sticking with it.

C Compilers

SAS C/C++ is what I use most often, it's very reliable and I've never had a problem with it. The documentation is very extensive, so you'll be able to get help for all your problems. This product is commercial but I don't think it is receiving updates at the moment. You may be able to get it from various software dealers.

DICE C is a nice package but it doesn't support C++. You have to register the compiler to get all the features, but you get to play with it first which is good for beginners. You can get this one from Aminet, in dev/c/ I believe.

Again, you will need a good text editor for efficient programming. CygnusEd is ideal here, and I believe GoldEd is a popular choice as well.

E Compilers

There is only one E compiler available (EC) which you can get as part of

the E package. You can get this from Aminet, along with everything else that you will need. You will probably have to register, although this program was put on a coverdisk some time ago.

1.8 About Structures and Lists

STRUCTURES AND LISTS

One of the things you will notice about the GMS structure format is the version header in the first longword (eg GSV1 for GameScreens). In the past, header fields have only ever been used for easily identifying files. If we didn't have them, we'd never know what sort of data we were dealing with.

Now the idea of these headers has now been taken and is being used in GMS structures. Why? Well for debugging, support for the growth of structures, and identifying exactly what any structure is used for. Also, it allows the functions to do different things according to what sort of structures you give them.

An example of this identification is for lists. What's a list? Well in the case of GMS a list is intended for processing 2 or more structures inside a function. This is the fastest way that you can process a whole lot of structures without having to make heaps of function calls. Lets say you wanted to load in 10 sounds from your hard-drive using InitSound(). Normally InitSound() takes a Sound Structure, but it can also identify a List by checking the header ID.

To illustrate, a typical list for initialising/loading sounds looks like this:

```
SoundList:
  dc.l  "LIST"              ;List identification header.
  dc.l  SND_Boom            ;Pointers to each sound to load and
  dc.l  SND_Crash           ; initialise.
  dc.l  SND_Bang
  dc.l  SND_Ping
  dc.l  SND_Zoom
  dc.l  SND_Zig
  dc.l  SND_Zag
  dc.l  SND_Wang
  dc.l  SND_Whump
  dc.l  SND_Bong
  dc.l  LISTEND             ;Indicate an end to the list.
```

When you want to load all your sounds in, just use this piece of code:

```
  move.l  GMS_Base(pc),a6
  lea SoundList(pc),a0    ;a0 = Pointer to the soundlist.
  CALL  InitSound
```

```
  tst.l d0
  bne.s .error
```

Pretty easy right? Of course, there are lots of other functions that support lists. The not-so obvious ones are:

```
  InitBOB()
  InitSprite()
  InitSound()
  FreeSound()
```

Some functions are specially written to be given lists only, eg DrawBOBList(). This is mainly for speed reasons, as we don't want to waste time checking if a structure is a list or not in time critical situations.

That's basically the summary on lists. You may be interested to know that the GMS package is the only programmers aid that supports structures in this way. You will learn more about lists and how ID fields will help you in other areas of this doc.

1.9 Hints and Tips

GAMES MASTER SYSTEM

HINTS AND TIPS

This section is written to offer some friendly advice and tips on how to get full use from the games.library, and what tricks you can use to make sure your game runs at the highest speed possible. I'm still writing this section, but if you have a trick of your own that should be here, please write to me at sandman@welly.gen.nz.

1.1 GENERAL CODING TIPS

Less... equals More!

Never call the same routine twice in your main loop unless absolutely necessary. For example, look at this routine that calls ReadKey() twice:

```
----
Loop: lea KeyStruct(pc),a1
  CALL  ReadKey
  cmp.b #K_ESC,d0
  beq Game_Over

  ...
  Rest of main loop
  ...

  lea KeyStruct(pc),a1
  CALL  ReadKey
  cmp.b #" ",d0
  beq .Exit
```

```
  lea GameScreen(pc),a0
  CALL  WaitSVBL
  bra.s Loop

KeyStruct:
  ds.b  KP_SIZEOF

----
Do this instead...

Loop: lea KeyStruct(pc),a1
  CALL  ReadKey
  cmp.b #K_ESC,d0
  beq Game_Over

  ...
  Rest of main loop
  ...

  lea KeyStruct(pc),a1
  cmp.b #" ",KP_Key1(a1)
  beq .Exit
  lea GameScreen(pc),a0
  CALL  WaitSVBL
  bra.s Loop

KeyStruct:
  ds.b  KP_SIZEOF
```

As you can see the second version is faster because it doesn't make an extra call to ReadKey(). Simple really, and this common sense applies to many situations.

1.2 MULTI-TASKING UNDER THE GAMES.LIBRARY

It is up to you wether you want to multi-task or not, I can only encourage you to do so. The games.library offers some special features to allow your game to multi-task effectively without disabling the OS entirely. Four functions are specifically related to multi-tasking, and they are:

```
  SetUserPrefs()
  Switch()
  AutoSwitch()
  WaitSVBL()
```

Firstly, SetUserPrefs() is a compulsory function that you must call immediately after opening the games.library. Apart from the fact that it can load in the user settings for you program, it will also set your task priority to the preferred user level. This is usually set to a default of 4, enough to give your program all CPU time. The highest setting is 8 which locks out just about everything except for standard system procedures (input device etc), while a setting of 0 is on equal footing with all other tasks. There is absolutely no need to completely lock out the system with Forbid()/Permit()!

Switch() is a very important function to use for multi-tasking. Lets say you've opened the screen and now the view is taken over. But, what if the user wants to return to the OS momentarily without quitting the game? By calling Switch() the screen will leave the display and either a) drop out to a window on workbench or b) drop out to a standard OS screen. What will happen exactly is up to the user. The OS will now be available to the user until he signals the games.library to return to your task.

The Switch() function is further supported by the AutoSwitch() function. This is very similar to Switch(), but checks the Amiga-M key combination for you and if found, switches the screen automatically. It's a lot easier to incorporate this in your main routine but some may prefer the Switch() function for full task control.

Finally is WaitSVBL(), which is probably the function you will use most often. This will perform an Amiga-M check before the vertical blank occurs, which can be very convenient in all circumstances. If you use this function whenever you have a VBL wait, your game should be 100% supportive of screen-switching without you needing to think much about it.

As a final note, whenever the OS is returned your game will be paused until the user enables you again. This is quite convenient, since we don't want the player to die when he can't see the action.

1.3 CONTINUATION OF TASK PROCESSING WHILE PAUSED

There are times when pausing of your main task (through Switch()) will be inconvenient if it is necessary to continually process information. For example lets say you are writing a game that can connects via the serial port for 2 player communications. If one machine was to stop its processing, the serial buffer will continue to receive information and could go into overflow, potentially causing you some problems when your task is reactivated. The easy solution to this is to activate a secondary task that will continue to process when the main task is switched. This is a simple procedure and only requires that you put all your communication handling into this separate task. Another method is to use an interrupt, which can be ideal.

An option that may be more convenient for the user in a TCP environment, would be to send out a message saying "This machine is temporarily paused" so that all other machines know that they must not send information to you. This will give any other TCP tasks running on the paused machine more time to send/receive data, eg for FTP.

1.10 games.library/Sound Overview

SOUND.GPI OVERVIEW

The GPI for sound support is one of the best Amiga interfaces for the support of Amiga sound effects. It features full sound prioritisation, intelligent channel selection on playback, and will eventually support special sound formats such as the common PC WAVE.

To illustrate the power of the Sound.GPI, here is an example. Lets say you develop your game on your A1200 to make dynamic use for all 4 sound channels. This is simply done by specifying CHANNEL_ALL in the SAM_Channel field. What that does is play the sound through whatever channel is currently available. Used in conjunction with the PlaySoundPri() function, you can make maximum use of all 4 channels, rather than just one channel with no prioritisation.

Clever right?

Sure, but if your game was to be run on an Amiga with say... 12 channels... then all those 12 channels will suddenly be supported by your game! This is because the Games Master System has been designed to allow your game to improve as the hardware gets better. Our aim is to get games written in 1997 to still have up-to-date sound support in 2007. (Yes, really).

As a further example, if you were to use IFF sounds loaded in from disk, you could support 16 bit sound, if the user was to update the sound files. (The hardware would have to support 16 bit sound of course). Packed sounds are also supported by the GPI so there are no problems in that department.

If you have any ideas for further improvements to the Sound.GPI, send them on in...

1.11 Screens Overview

SCREENS.GPI OVERVIEW

The Screens.GPI was the first GPI to be designed, and speed was a major factor while it was being programmed. To keep it fast, the GameScreen structure has been kept highly simplified while not giving away any powerful features. One example of this power is that you can move the screen around by changing its coordinates, and even dynamically alter its width and height without any adverse affect on the picture display (see the Redimension demo).

There exists a wide range of functions, including special effects such as proportional fading, which allows you to add some very smooth and impressive touches to your programs. Sprite support can give you some extra speed increases where you need it. Features from AGA sprites are also fully supported if you need extra wide sprites or access to colours in different areas of the palette. Finally is the rasterlist support, which provides an easy to use gateway to the copper chip. Using the available commands you can acheive affects like mirrors and smooth gradients of colour.

The Screens.GPI is further supported in GMSPrefs, allowing the user to select his preferred screen modes. A powerful feature is being able to select the screen type, so you can change the display type from ILBM to Chunky for example. This can give you a great speed up if your hardware allows you to use such modes and if the game would benefit from such a change (eg 3D vectors). It is even possible to do things such as upgrading a game to hi-res interlaced, or running in different video modes such as

DBLPAL.

1.12 Blitter Overview

BLITTER.GPI OVERVIEW

The blitter support is designed so that it can perform the task of drawing images to screen as quickly and efficiently as possible. The best way of doing this is to provide you with a large amount of drawing options, so that you can specify exactly how you want an image to be drawn. For experienced programmers this level of functionality is extremely useful in providing fast and optimised drawing speeds. If you're a beginner it may take some time before you learn what methods to use in each situation, but with a little practice you will know how to use the available options to your advantage. Currently, options include blitting in lists, clipping on/off, restore and clear modes, masking on/off, mask generation, multiple BOBs, setting clip areas, and others.

To make the Blitter.GPI as effective as possible, special rules are in place that are ideal for games applications only. If you have come from OS programming then the ideas may sound a little unusual, but for games programming they make sense. The nature of any game is not to multi-task with other games, as it is impossible for a player to play two different games on the same screen at once. If two different games try to share resources, the result can be a catastrophe. Since GMS only allows one game to be using the display at any given time (ie no windows or screen title bar moving), it made sense that the only task allowed to use the drawing functions is the one at the front. This means that at any time when your task is active, you know that you have 100% available blitter time. Your drawing cycles will not be stolen by hidden tasks!

It was suggested to me in the past that I could use QBlit() or similar interrupt driven blitting. The advantage of this is that it is easier for the processor to do things while the blitter is active, and there is no blitter waiting involved. This sounded like a good idea at the time, but after trying it I found the results to be disappointing, so I dropped it. Why? Because this method did not recognise the fact that the blitter is so SLOW! Using the blitter only, you would be lucky to get 7 32x32x32 BOBs on a 50fps screen with clear modes on. It doesn't matter how fast your routines are, the blitter will not move data any faster. This a sorry speed for any arcade game to be using.

Instead I am now in the process of implementing high-speed CPU assisted blitter drawing routines. These work extremely well in mass drawing operations with about 20-30% speed up on my '020 A1200+Fast in comparison to blitter-only drawing. On a '030 I would expect at least 40% faster operations, while on '040 and '060 we are probably looking at the CPU drawing 2 bobs while the blitter draws 1. I think Amiga owners with fast CPU's will appreciate this feature, while '000 users will not suffer because the blitter will take most of the load for slow CPU's.

Enjoy the fast drawing, and if you have any good ideas for advancements then send them all in.

1.13 Compatibility Problems

COMPATIBILITY PROBLEMS

One of the most important decisions I made in the design of GMS, was to get the absolute most out of what the Amiga hardware is capable of. The fact is, if I wrote GMS with respect to other gfx cards, there would be no:

Sprites
Hardware Scrolling
Overscan
Double Playfields
Split Screens
and RasterLists

Strangely enough, isn't this what makes the Amiga unique? Also if the new Amiga's came out with quadruple 256 colour playfields and 512 colour sprites, should I support that if other gfx boards don't? Why should I not support it?

Well one of the goals of GMS, is to always be as up to date as the hardware that is available at the time. It's vital if the Amiga platform is going to beat the competition, and there is a lot of it.

Now, this is at the cost of total compabitility. How compatible you want your game to be on other systems is entirely your own choice. Generally, the more hardware-specific features you use, the more you risk your games failure on different hardware. If you use less, your game has an excellent chance of successfully working on all systems. Whatever happens, you will have to make your own decisions on the compatibility issue.

What I will do in the future of this section is try and help you face these problems, and hopefully overcome them. With some intelligent programming, you can still use features like sprites, rasterlists and hardware scrolling, and still keep your game running on other systems. It takes a little work but the least it will do is make a lot of people very happy. Good luck!

1.14 The Authors

THE AUTHORS

The Games Master System is written in assembler by Paul Manias (that's me!). Paul has 5 years 68000 and games programming experience, and another 1 year in other languages like C and Pascal. Paul's favourite past-times are sitting down, staring at the ceiling, and eating soft toys. So far he has written two games of his own and contributed graphics to two other commercial ones. None of those games have beeen released (yet?), for all sorts of various reasons. Luckily this is not the case with GMS.

GMSPrefs is written in E, by Richard Clark. Richard's favourite past-times are standing, sending morse code via blinking, and talking to suspicious

items of furniture.

Thanks to Jyrki Saarinen and Fabio Bizzetti for their donations to the project. Some of the C demos were converted by Adam Dawes, good job Adam! To the people that send in the useful bug reports, thank you. Also to the many people that sent in ideas when the project first started (but we still need more!).

The web page exists thanks to Graeme Chiu, who owns a computer shop at CompKarori. To see the pages, visit:

http://www.compkarori.co.nz/amiga/gamelib/