Authored by Andrew Murray (Max)
Web: http://www.netyaroze-europe.com/~leon17
E-mail: sckoobs@yahoo.com

Contents

1.0. Version History

Version Number	Date of release	Comments
1.0	12/05/01	First release
1.2	08/07/01	Added BG Section Added Glossary Minor Amendments
1.3	12/03/02	Added BG Scrolling Added Sprite animation Major aesthetic changes
1.4	3/12/02	Added Advanced Pad Functionality Major content changes

2.0. Foreword

This tutorial is aimed at those members who have not yet done any 2D stuff on the Yaroze. I personally jumped straight into 3D on the Yaroze so I was not 100% sure about the workings of the second dimension. So basically I am writing this tutorial as I learn about 2D. It is as much for my benefit, so as to keep a record (for my reference), as it is for you. It is written as if it was designed to be a tutorial though. Good luck… and enjoy (if that is possible☺).

If you find any mistakes (and I'm sure you will), typos or you have constructive criticism then please e-mail me at sckoobs@yahoo.com and I would be happy to help.

Conventions used in this tutorial:

- The `boxed courier` font represents code.
- All code snippets that lie within normal text are like so: *Code Snippet.*
- Whenever I say 'sprite' or 'image' I mean just that. It is just that I throw that word around a bit, but you should know they both are the same thing, well in the context of this tutorial anyway. The PlayStation has two different structures *GsSPRITE* and *GsIMAGE* these are distinguished in the tutorial.

3.0. Reasoning

When I first got my Yaroze it was very hard for me to understand. I wondered what I had let myself in for and whether I should have gotten one at all due to its complexity. I hadn't done any game programming before and wasn't much good at programming anything in C, but writing this tutorial has really helped me. Both 2D and 3D are a hell of a lot clearer to me now. Therefore I did the tutorial because I wanted to explain every detail to anyone who cared to read it - I think too much is taken for granted when games programmers start talking about programming games so I wanted to take nothing for granted and instead lay the information out in a manner that is clear and easy to understand. I hope you benefit.

4.0. Aims & Objectives

Throughout this tutorial, I hope to go into the following aspects of 2D on the Yaroze:

Part No	Name	Date
1	Double Buffering and Screen Resolutions Displaying Debug Text	Added 12/5/01 Modified 3/12/02
2	Acquiring & Reacting to Pad Input Advanced Pad Input Handling	Added 30/6/03
3	Manipulating Sprites Creating a TIM File Displaying the TIM on the screen	Added 12/5/01 Modified 3/12/01
4	Sprite Animation TIM Layout and Theory of Animation Animation with pad input	Added 11/3/02 Modified 25/7/03
5	Background Creation PlayStation Backgrounds Handling Implementing a Background using the Yaroze libraries Implementing a Custom Background Handler	Added 3/11/01 Modified 30/6/03
6	Collision Detection Bounding Box Bounding Circle Pixel Based Collision Detection	TBA
7	Loading and using bitmap fonts	TBA
8	Game Music and Sound Sound Effects Music	TBA

There are many programs included in this tutorial each covering one of the above sections. Each part number specifies which directory it is in. There are two directories; one for GNU code and the other for Code Warrior code, although the Code Warrior code is not evident yet the idea is that inside each of these directories there will be the code relating to each section. There are, naturally, files that contain code from other sections, as the idea is to have a complete game at the end of the tutorial, adding functionality as we go. Sections 5 to 8 will be added in future revisions.

5.0. Assumptions & Recommendations

- I am writing this tutorial on the assumptions that you all know how to create a bitmap (.bmp) file. I will cover converting it to a TIM format file using TimTool3. I am using TimTool3, as it is better than TimUtil (which only deals with NTSC resolutions [yuck]).

- You will need a firm grasp of C to understand the code; get a good book if you do not have one. I personally recommend “Beginning C Programming” by Ivor Horton, ISBN 1-861001-14-2 Published by Wrox Press, but everyone has preferences.

- This is more of a strong recommendation than an assumption but you should read the “Net Yaroze Programming FAQ” it contains a lot of answers to questions you may be asking throughout this tutorial. Use it as a reference; that is the way it is written.

I should also let you know that the code that is used in this tutorial is PAL only. There has been NTSC code included in the NTSC subdirectory (included soon). There is also a section in the appendices on converting PAL games to NTSC and vice versa.

Please download the updated Library Reference .pdf file from the Yaroze Servers if you haven’t already got it because the manual that came with the Yaroze has quite a few mistakes in it and is not worded brilliantly either.

6.0. Double Buffering and Screen Resolutions

Right! First, we have to mull over the (really) basic stuff such as which library files to include, defining constants, declaring structures, variables and what procedures we need to implement an easy double buffering example.

6.1. Includes

I will not spend too much time on this because it is really a case of taking things at face value. You can look around inside the header files if you want but you only really need to know that the functionality is there.

```
#include <libps.h>
```

<*libps.h*> is the Play Station Specific Library we need to include to enable us to utilise the functions in the Reference Manual.

6.2. Defines

This part is easy when you know what you are looking for.

```
// screen height & width
#define SCRNW  320
#define SCRNH  256

// kbyte for setting the size of the work area
#define KBYTE  1024

// ordering table length
#define OT_LENGTH     9
```

First the screen width and height are defined so that they can be used throughout the code instead of constantly typing actual numbers. This is pretty much a convention as it explains in English what the values being passed to functions are. You must make sure that the numbers used are valid for the mode you are in i.e. PAL or NTSC, look at the reference manual page 100 for valid screen resolutions.

The *NO_OF_SPRITES* define basically is the number of sprites we want to display on the screen. This is used later for declaring our packet working area. A packet is the smallest amount of data the GPU can handle at any one time.

Finally we will have to declare the length of the ordering table. (An ordering table is a general game programming term, which just refers to an area on screen where sprites and other visual game data are put on for sorting and displayed according to the order they have been sorted in).

You see, ordering tables have several layers, the amount of which depends on how many you declare in your program. Then when you are drawing your sprites (and/or objects in 3D) they are prioritised meaning that each sprite and/or object is given a specific layer in the ordering table to be drawn to. This effectively means that if you give; say a sprite of a building a priority of 0 (which happens to be the highest priority) then it will be drawn first before anything else as 0 is always regarded as the highest priority and the higher the

priority the sooner it gets drawn. Then if you want to draw another sprite of a soldier and you want it to be drawn in front of the building you would give the soldier a lower priority of 1, this means that the soldier is drawn after the building because it has a lower priority. Anyway this is described more in the code for a later part of the tutorial – "Getting the TIM on the screen".

6.3. Global Variables

```
// current frame number
int frameNo;

// holds the ID of the open font stream
// used to pass to FntPrint so it can
// identify where on the screen to print
int g_fntID;

// ordering table headers
GsOT           g_WorldOT[2];
// ordering table elements
GsOT_TAG       g_WorldTags[2][1 << OT_LENGTH];

// gpu packet work area
static PACKET g_packetArea[2][24 * 100];
```

Here we have a variable to track the current frame number, in this demo we will wait till the program has ran for 200 frames then it will terminate, this will be dealt within the main function. The next variable, the *fntID* will hold the ID (a number) of the font drawing area that is opened to the screen. This is used later when actually making font calls to identify where on the screen the particular string should be displayed.

We need two Ordering Tables headers so we can implement double buffering (a table for an on-screen surface and a table for an off-screen surface). Therefore we make an array of *GsOT*. Next we use some bit-wise maths to declare the ordering table units (again a good C book will have this sort of thing):

1 << OT_LENGTH
OT_LENGTH = 9
1 << 9 = 512

We need a 2 dimensional array of *GsOT_TAG*, here is why: the '2' in the first dimension is because there are two buffers (hence double buffering) i.e. we need *GsOT_TAG*s for both the off-screen and the on-screen buffers. Next is the bit of maths you see above, this equates to 512. This number is for the amount of layers there are (think of the Z-axis in 3D except more like the layering system in paint programs). The more layers you have, the more priorities there are when drawing your objects to the screen as explained above. You may want to restrict this as it uses memory and you won't be using that many layers anyway (hopefully). This is explained in the Yaroze FAQ under the "What is an ordering table? What is an Ordering table Length?"

Lastly is the *packetArea* two-dimensional array. This is the area where all objects are z-sorted into layers to be put on the screen. The first dimension *[2]* is there because we have 2 ordering tables (this rarely changes from program to program; single buffering is messy and triple buffering is too complicated and limits performance as well as the

amount of space on VRAM). The second dimension only needs to be as big as the data you are putting on the screen. This is pretty much an arbitrary number but some sense can be obtained by using known values. The Yaroze FAQ states that a sprite command is approximately 24 bytes. Since this area is the place where graphics commands are placed, we can say that each call to a *GsSort** function will add more data to be processed, so generally you can make your packet area 24 * the number of *GsSort** calls made in the program. Right now we have no *GsSort** calls so we will just stick it at an arbitrary value of 100. Usually if you are having unexpected and unexplained behaviour then increasing this number in increments of about 10 might help.

6.4. Function Declarations

At this point we want to think of what we want to do with the program. I generally lay out my thoughts on paper taking into consideration what type of game/program you are making. In the case of this example all we want to do is initialise the graphics system for drawing. This includes setting up the screen resolution, the buffers and the ordering tables for each buffer.

```
void InitGfx(void)
{
	// this allows use of PAL resolutions
	SetVideoMode(MODE_PAL);

	// initialise the graphics engine with 320 x 240 resolution
	GsInitGraph(SCRNW, SCRNH, GsNONINTER|GsOFSGPU, 1, 0);

	// define the buffer positions on VRAM
	GsDefDispBuff(0, 0, 0, SCRNH);

	// set OT length and link the OT
	//elements (the GsOT_TAG structures)
	g_WorldOT[0].length = OT_LENGTH;
	g_WorldOT[0].org = g_WorldTags[0];
	g_WorldOT[1].length = OT_LENGTH;
	g_WorldOT[1].org = g_WorldTags[1];

	// initialise the OTs
	GsClearOt(0, 0, &g_WorldOT[0]);
	GsClearOt(0, 0, &g_WorldOT[1]);

	// set the initial display environment parameters
	// this is required as PAL resolutions require
	// extra initilisation code to display to the
	// full extent of the requested resolution
	GsDISPENV.screen.x = 5;
	GsDISPENV.screen.y = 18;

	GsDISPENV.screen.w = SCRNW;
	GsDISPENV.screen.h = SCRNH;

	// load the font into VRAM
	FntLoad(960, 256);

	// open the font to the screen
	g_fntID = FntOpen(10, 80, 256, 200, 0, 512);
}
```

The first call is to *SetVideoMode*; it tells the Yaroze that we will be using PAL screen resolutions. The next line looks a bit daunting! *GsInitGraph* does as it suggests; initialises the graphics engine, however it’s the parameters that need explained.

Referring to your manual (page 100) you will see that the first two parameters are the width and height i.e. the resolution of the screen. The second parameter, *GsNONINTER*|*GsOFSGPU*, is a flag to the GPU indicating that we want a non-interlaced display as we aren't using high screen resolutions. (Read the FAQ "How can I get more CPU/GPU time?"). The *GsOFSGPU* part means we want the GPU to handle the double buffer offset (refer to the *GsInitGraph* description in the Net Yaroze Reference manual).

Please note that the interlaced display mode is usually used for higher resolutions such as resolutions with the vertical resolution equal to 512 or in NTSC equal to 480, there are documents on-line that explain how to achieve higher resolutions such as James Chow's great explanation at http://www.netyaroze-europe.com/~jaycee under his link "Going Hi-Res". I recommend you stick to a lower resolution just now because VRAM is limited when in higher resolutions.

The call to *GsDefDispBuff* defines our double buffers for drawing. The first two parameters define the first buffer which is buffer #0 at a position of *0*, *0* on the VRAM (the top left corner). The second buffer is declared as *0*, *SCRNH* (*SCRNH* being 256). The second buffer is declared to be below the first buffer on the VRAM. To get a better idea of how this is set up look at the buffers sitting to the left of the VRAM in TimTool 3, this may help you understand how this is set up.

The ordering tables are set up next. These are the source of a lot of confusion for those just starting with the Yaroze. The *GsOT* structure acts as a header to the actual ordering tables, the ordering tables them selves (the elements within them) are the *GsOT_TAG* structures. The *GsOT_TAG* structures are set up in a linked list (have a look at any good C book). Each node (element) of this linked list contains the number of graphics primitives in the next GPU packet, a packet being the smallest denomination of drawing instructions. The *GsClearOT()* function calls after this are used to zero the memory within the newly set up ordering tables.

The next four lines set up the PlayStation drawing environment. First of all the position of drawing is set to values that I have set through trial and error; I ran the program and offset it as I saw fit, so the new screen position I have set is x: 5, y: 18. We will see later how we can change this at run time to give a screen positioning option. The next two lines set the drawing screen width and height, the reason this has to be done is that typically these values are set to the nearest NTSC resolution values, so PAL programmers have to set this manually to take advantage of the 16 or so extra lines at the bottom of the screen.

The next step loads the Yaroze debug font into the VRAM. Remember when placing images on VRAM in TimTool 3; don't place any graphics in the lower right t-page (the tall divisions of which there are 2 vertically) as this is where this font is stored. After this an area on the screen is set up as a region into which all font calls will be drawn (refer to manual page 62 and 63).

6.5. The Main Function

```
void main()
```

```
{
	// buffer index
	int buffer = 0;

	// set up the graphics
	InitGfx();

	// grab the active buffer on which to commence drawing
	buffer = GsGetActiveBuff();

	// loop through 200 times and then terminate
	while(frameNo < 200)
	{
		// increment frame count
		frameNo++;

		// set the packet work area to the packet area for the current buffer
		GsSetWorkBase((PACKET*)g_packetArea[buffer]);

		// initialise the current ordering table for drawing
		GsClearOt(0, 0, &g_WorldOT[buffer]);

		// print the frame number to the screen
		FntPrint(g_fntID, "frame: %d\n", frameNo);

		// print the font call
		FntFlush(g_fntID);

		// execute all non-blocking GPU drawing commands
		DrawSync(0);

		// block until vsync occurs - makes sure everything
		// for current frame is drawn before going onto next
		VSync(0);

		// swap the double buffers
		GsSwapDispBuff();

		// clear the buffer to black
		GsSortClear(0, 0, 0, &g_WorldOT[buffer]);

		// execute ordering table drawing commands for
		// appropriate graphics buffer
		GsDrawOt(&g_WorldOT[buffer]);

		// change side to 1 or 0
		buffer = buffer^1;
	}

	ResetGraph(0);
}
```

The *buffer* variable tells us which one of the buffers are active, this being 1 or 0 (*g_WorldOT[1]* or *g_WorldOT[0]*). The *InitGfx()* call will set up all the graphics features that we are going to use as you saw earlier. The next line will get the current active buffer (1 or 0). Then we enter the main while loop, which is set to repeat 200 times.

Firstly, the frame counter is incremented. The *GsSetWorkBase()* call sets the working area that we use for drawing the screen and everything in it. This uses the *g_packetArea* variable we declared earlier and casts it from type *static PACKET* to *PACKET* * as the function requires a reference to the work area memory. The array reference part refers to which of the buffers is being drawn.

GsClearOt() will clear the buffer ready for drawing (it steps through the *GsOT_TAG* linked list and zeros them). After this the frame number is drawn. When making font calls remember to use the ID of the print stream that was opened. The ID is passed to the Print call and to the flush call. The print call essentially registers the text on the screen and the flush call will draw all text with the same font ID.

The next call, to *DrawSync()* will ensure that all drawing commands are finished before the buffers are swapped. The *VSync()* call will ensure that a complete vertical screen trace has executed i.e. the whole buffer has been displayed to the screen.

Here comes the perplexing bit. What you draw in the current frame will be drawn two frames later. What happens is that the GsSort* functions you call are entered into the ordering table in the current frame while the current buffer is being displayed. The next frame the commands entered in the ordering table are drawn onto the back buffer. The next frame, when the buffers are swapped, the commands that you entered two frames ago are being displayed on the screen. Now, with that out of the way…

GsSwapDispBuff() will swap the back and front buffers, so that the drawing commands that were registered last frame are now being drawn onto the back buffer. *GsSortClear()* will register a command to clear the specified ordering table to the specified colour which are the first three parameters (red, green, blue). The *GsDrawOt()* will register the command to actually draw on the back buffer next frame. The buffer is then swapped.

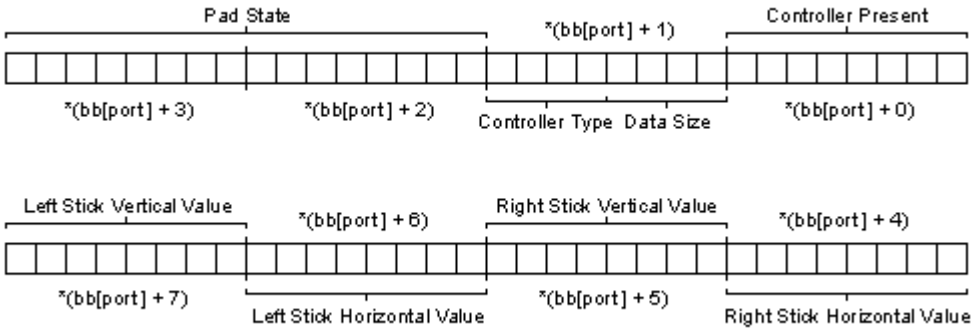
When the loop is exited from (when the frame number counter reaches 200), the graphics system is completely reset with the call *ResetGraph()*.

7.0. Acquiring & Reacting to Pad Input

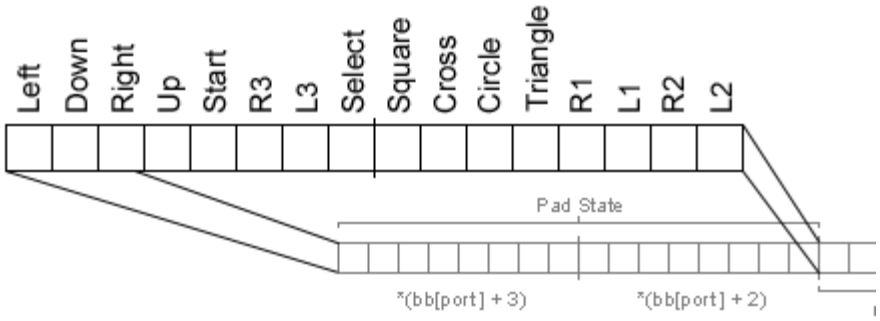
I suggest that you read chapter 12 of the Yaroze user's manual, pages 104 – 108.

7.1. The Pad Buffer

Obtaining pad input is actually quite easy. Many Yaroze programs already have source and header files you can link into your code to get some functionality. All these source and header files do is access a specific part of memory, which will hold the state of the pad. Fundamentally, the pad state is all of the buttons that are pressed at the time of input capture, however; we will expand the program to include other functionality such as reacting to button presses and releases. The pad buffer area is depicted in the diagram below.



The main area of memory that we need to concern ourselves with are the first 64 bits of the pad buffers. There is other data but this is for specialist pads, such as the Mouse and NegCon. Each of the small boxes represents one bit, eight of them equal a byte and bytes are sectioned off in the diagram by the lines after every eighth box. When a button is not pressed its bit is on i.e. it is 1 and it is 0 when the button is pressed. The code in the pad source file reverses this (for the context of our program) so that 1 represents a button being pressed. The diagram below represents the region of the pad buffer that contains the buttons that are pressed or released.



Before I get into much depth I will explain one thing that may be bothering people new to C\C++ and programming in general. The pad buffers are an area of memory on the

PlayStation, which is specially set-aside for the sole purpose of storing the pad's state. The following line of code, found in some shape or form in most *pad.c* files;

```
volatile u_char *bb[2];
```

is an array of two pointers to this special memory, an array element for each of the two pads. The volatile keyword means that the memory is liable to change outside of the program's control, which of course we expect because the hardware constantly updates the buffers corresponding to the player's input. So whenever you see **bb[port]* or similar, we are requesting data from the pad buffer (the star is a de-referencing operator used to get the actual values in the memory that the pointer references, instead of the address of the memory).

Before using a pad it is necessary to see if it is actually plugged in. The value in the first byte (**(bb[port] + 0)*) is *AND*ed with the value 0xff, this is testing that all the bits in the first byte are set to 1, as this means the pad is connected. If this is not the case then you should take steps in your code to inform the player that there is no pad connected (for example, pause the game if there is one in progress). You may also want to test what type of controller is connected. This is stored in the second 4 bits of the second byte (**(bb[port] + 1)*). To obtain the value; bit-shift **(bb[port] + 1)* right by 4 as shown in the function below; *PadType()*, this moves the data size value out of the way.

```
int PadType(int port)
{
	return (*(bb[port]+1))>>4;
}
```

Typically you want to test for a standard pad, an analogue pad or a dual shock pad. The values of each are defined in the pad header file; a simple *AND* of the return value from the *PadType()* function with the appropriate pad type define will determine what type of pad is connected.

7.2. Obtaining Pad Data

There will typically be a function used to obtain the pad data as an integer or signed/unsigned long type (each is a 32bit type), the included header file includes a function called PadState() which returns an integer.

```
int PadState(int port)
{
	return ~(*(bb[port]+3) | *(bb[port]+2)<<8);
}
```

Essentially this function stores the pad state in the lower 16 bits of the integer that is returned. The function *OR*s the contents of the 3rd and 4th byte of the buffer, the 3rd byte (**bb[port]+2*) is left shifted by 8 bits so that it is put into the left side of the lower 16 bits, if the bits were simply *OR*ed, the 3rd byte would mess up the data in the 4th byte. The NOT operator, ~, at the start of the statement inverts the bits so that a 1 represents a button being pressed instead of a 0, as discussed earlier.

The buttons can be tested for by *AND*ing the integer pad state, returned from the *PadState()* function with the appropriate define in the pad header file. For example to test for the circle button being pressed the code would be along the lines of;

```
// aquire pad status from the buffers
int padState = PadState();

// test for circle being pressed
if(padState & PADCircle)
{
	// do something
}
```

7.3. Obtaining Advanced Pad Data

Requesting the pad state in the aforementioned form does not give very much information about player input, for example, we don't know in which frame a button has been pressed or released. If you take the pause function of a game as an example, merely testing for the pad state will result in the game being paused every second frame for as long as the button is held down, when what we actually want is the game to be paused from the first button press until the second button press.

When the word 'press' is mentioned, what is meant is the actual action of pushing the button down. The word 'release' refers to the action of the button moving back up and the button state is the in-between part, i.e. the state is 1 (button down) until the button is released.

We can encapsulate the new functionality in the structure below.

```
typedef struct
{
	u_short		old,
			state,
			toggle,
			release,
			press;
}PadInfo;
```

Each member is an unsigned short, 16 bits, which will hold the 3rd and 4th byte of the pad buffer. What follows is an explanation of the members.

old	Holds the state of the pad for the previous frame or last time the pad state was acquired and is used to set the toggle member.
state	Holds the current state of the pad, i.e. which buttons are pressed.
toggle	Holds the change in pad state between frames which is used to determine the press and release members.
release	Holds a snapshot of the buttons that were released since the last time the pad state was acquired.
press	Holds a snapshot of the buttons that were pressed since the last time the pas state was acquired.

The only members that will be used explicitly by our code are the *state*, *release* and *press* members, the other members are merely used to calculate the *release* and *press* members. The following function *GetPadData()* returns an integer type containing the pad state in the same way that the aforementioned *PadState()* function did, but *GetPadData()* also fills the above structure with data that we can use to gain further insight into the player's input.

```
int GetPadData(int port, PadInfo *pPadData)
{
	// set the keys that were pressed last frame
	pPadData->old		= pPadData->state;

	// set the pad's current state
	pPadData->state		= ~(*(bb[port]+3) | *(bb[port]+2)<<8);

	// the keys that have changed state between frames
	pPadData->toggle	= pPadData->old ^ pPadData->state;

	// the keys that have just been pressed this frame
	pPadData->press		= pPadData->toggle & pPadData->state;

	// the keys that have just been released this frame
	pPadData->release	= pPadData->toggle & pPadData->old;

	// return the pad state
	return(pPadData->state);
}
```

An integer value indicating which port is to be read (0 or 1) is passed in as well as a pointer to the instance of the *PadInfo* structure.

The fist line simply copies the last state of the pad over to the member *old*, this obviously has to be done before the new pad state is acquired on the second line, which is the same line of code as in the *PadState()* function, above.

On the next line the toggle member is set by doing an exclusive OR of the previous pad state with the new pad state, from your C text book you should garner that an exclusive OR gives a 1 when there is only a single 1 involved in the operation. We could therefore have a pad buffer with the bit set for the circle button and cross buttons, i.e. they are pressed, and the old pad state had the R1 and cross buttons pressed, so the buffers would look like this:

Old	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
New	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

So the result, which toggle would be set to is:

Result	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0

In the following lines of code you can clearly see the effect that this will have on the results for the press and release members. A simple AND of the bits with the new state gives the keys that have been pressed this frame and the same operation with the old state gives the keys that have been released this frame.

7.4. Pad Demo

Now I will present a demonstration of the functionality that we have spoken of in this section. The demo allows the user to exit the application now at any point, rather than waiting until 200 frames have passed. This is done by pressing the start and select buttons at the same time. The pad demo (part 2) contains the same *InitGfx()* function as the previous demo for Double Buffering and Screen Resolutions (part 1) did so I will leave this function out.

7.4.1. Obtaining and Checking Pad Input

The new function *GetInput()* obtains the pad input by calling *GetPadData()* and then tests the data to determine if certain buttons are pressed or not.

```
void GetInput()
{
	// instance of the pad data structure
	// although 2 are defined (for both pads)
	// only one is used in this function
	static PadInfo padStatus[2];

	// request the input from the pad in port 1
	GetPadData(0, &padStatus[0]);

	if(PadConnected(0) != PadOut)
	{
		// determine what type of pad is connected
		switch(PadType(0))
		{
			// standard playstation digital pad
			case PadStandard:
				FntPrint(g_fntID, "STANDARD PAD DETECTED\n");
				break;
			// analogue playstation pad
			case PadAnalogue:
				FntPrint(g_fntID, "ANALOGUE PAD DETECTED\n");
				break;
			// dual shock playstation pad
			case PadDualShock:
				FntPrint(g_fntID, "DUALSHOCK PAD DETECTED\n");
				break;
		}

		// start pressed
		if(padStatus[0].press & PadStart)
		{
			g_paused = !g_paused;
		}

		// up pressed
		if(padStatus[0].state & PadUp)
			FntPrint(g_fntID, "UP PRESSED\n");

		// down pressed
		if(padStatus[0].state & PadDown)
			FntPrint(g_fntID, "DOWN PRESSED\n");

		if((padStatus[0].state & PadStart) &&
			(padStatus[0].state & PadSelect))
			g_finished = true;
	}
	else
	{
		FntPrint(g_fntID, "PORT 1: PAD NOT CONNECTED\n");
	}
```

```
}
```

The first line here creates an array of two new *PadInfo* structures, one for each port. The pad data is then fetched by calling *GetPadData()*, its arguments are the port that we want the data for and a pointer to the *PadInfo* structure that will be filled with the data. If the next line is false i.e. the pad is not connected, the *PadInfo* structure will contain nothing.

The if statement checks to see if the pad is connected by calling the *PadConnected()* function with the port number passed as an argument. The value returned from this is tested against the defined value (from pad.h) called pad out, which is the value 0xff, which is 255 the maximum value a byte can hold, the byte which is tested inside *PadConnected()* is **bb[port] + 0*.

If a pad is connected then the pad type is determined, this is good practice as certain features you may want to include in your game may only be available if an analogue or dual shock pad is present for example. Page 105 of the Yaroze User Guide should clear up any questions you may have, except that the dual shock type is not mentioned. The *PadType()* function simply returns the top 4 bits of the 2nd byte of the pad buffer. This is then checked against the defined types and the appropriate message is printed to the debug front stream.

The if statements that follow simply check for pressed buttons. You can add as many if statements here as you like, I have only checked for start being pressed which will pause the demo and the d-pad up and down buttons. The terminating condition is if the start and select buttons have been pressed together.

7.4.2. The Main Function

There are relatively few changes in the main function.

```
void main()
{
	// buffer index
	int buffer = 0;

	// set up the graphics
	InitGfx();

	// grab the active buffer on which to commence drawing
	buffer = GsGetActiveBuff();

	// terminate only when told to
	while(!g_finished)
	{
		// set the packet work area to the packet area for the current buffer
		GsSetWorkBase((PACKET*)g_packetArea[buffer]);

		// initialise the current ordering table for drawing
		GsClearOt(0, 0, &g_WorldOT[buffer]);

		if(!g_paused)
		{
			// increment frame count
			frameNo++;
		}
```

```
                // get the user input
                GetInput();

                // print the frame number to the screen
                FntPrint(g_fntID, "frame: %d\n", frameNo);

                // print the font call
                FntFlush(g_fntID);

                // execute all non-blocking GPU drawing commands
                DrawSync(0);

                // block until vsync occurs - makes sure everything
                // for current frame is drawn before going onto next
                VSync(0);

                // swap the double buffers
                GsSwapDispBuff();

                // clear the buffer to black
                GsSortClear(0, 0, 0, &g_WorldOT[buffer]);

                // execute ordering table drawing commands for
                // appropriate graphics buffer
                GsDrawOt(&g_WorldOT[buffer]);

                // change side to 1 or 0
                buffer = buffer^1;
        }

        ResetGraph(0);
}
```

There is the inclusion of the *g_finished* variable which stays as true until the start and select buttons are pressed at the same time. The if statement checks the pause state which is changed when the user presses start, the frame counter is only incremented if the demo is not paused. The last new line is the *GetInput()* function call, which will read the user's input.

8.0. Manipulating Sprites

8.1. Creating a TIM File

Now that we want to delve into the Yaroze sprite handling capabilities we need a sprite to do it with.

Creating the sprite is the easy bit. Make a sprite you wish to display on the Yaroze in your chosen paint package. If you have a paint program that can convert images to different bit indexes then I suggest you use these functions as they make your image look better when converted to lower bit indexes. You can use the 8 bit index most of the time, but there may be some cases where a 4 bit index can be used without much quality being lost from the image, basically the lower the better as long as it doesn't look too ugly.

Now you should export your 8 or 4 bit image as a .bmp. Keep the filename within eight characters so that when converted through TimTool3 they will not have an absurdly non-understandable file name. You should consider naming conventions if you are making a complete game.

Open TimTool3 and under the "Frame Buffer Options->Set Graphics Mode" set the resolution to PAL 320x256. This must be set to be the same resolution as your game uses or else you will have a corrupted picture when displaying it. Now click "File->Import Image File" or Ctrl + I, then point the browser to your 8 or 4 bit BMP, check that the colour depth is the colour depth that you set the image to in your paint package, then click the "OK" button in the bottom right corner. This will put your image and it's CLUT (colour look-up table) on the video RAM, which is that large area in the window depicted by many oblong rectangles. Click and drag the image so that it is lying on the turquoise coloured area, also grab the CLUT (this may take a while to find but is usually located at 0, 0 on the frame buffer when imported) and place it somewhere on the turquoise area as well. When making a full game it is best to have a designated area in V-RAM for your CLUTS and another area for your bitmaps/textures. Or you can keep your CLUTS and textures together; it's up to you really.

The areas where you have placed the image and CLUT are the areas that you will reference in code so that the sprite can be set up ready for display. You can find the actual image position co-ordinates on the V-RAM at the bottom right corner of the main window under the TIM Information panel. From here you can position the image and the CLUT exactly rather than dragging them around with the mouse. Also check that the image Colour Depth panel is set correctly to either 8 or 4 bit depending on what you set it to in your paint program.

Finally click "TIM Options->Save Modified TIMs" or Ctrl + S to save your TIM to the directory you opened your BMP from. Now you have a TIM file to use. I have included a small sprite of a character we will call Barney for now. You can replace it with a sprite of your own. Anyways, it's onto the next stage…

8.2. Modularity & Readability

Now this is the fun bit. If you get this done and you understand it very well, you are well on your way to making a complete 2D game. I will re-type the code sections that appear from part one as this helps you see the structure of the program and where everything fits in.

The first thing you will notice about the project is that there are a lot of separate source and header files. Up until now it wasn't really necessary for the use of lots of files simply because we had very little code, now however we need functionality for sprites which entails the use of two large functions called *LoadSprite()* and *InitSprtie()*. Notice also that the functionality previously associated with graphics handling (the ordering table and GPU work area stuff) has been included in graphics.c and graphics.h, I will explain these functions first, before we start into the sprite functions and the main program.

First open up graphics.h. You will see some defines at the top of this file, these define the screen width and height (as with previous programs), the size of a kilobyte (used for assigning memory mainly) and the depth of our ordering table, so nothing really new here. Next is the graphics structure which is really handy:

```
// This stucture holds all the data used for double
// buffering, the font id(s) and the work area
typedef struct
{
	GsOT ot[2];
	GsOT *pOT;
	GsOT_TAG ottags[2][1<<OTLEN];

	PACKET gpuPktArea[2][64*KBYTE];

	int buf;
	int fntID[1];

	// these hold the GsDISPENV position on the screen
	// this is required to properly initialise PAL resoloutions
	int screenPosX;
	int screenPosY;
}Graphics;
```

This structure basically just contains all of the stuff that has previously been found at the top of the main source file: the ordering table headers, a pointer to an ordering table header and the tags (ordering table elements). We have a pointer to an OT header because most of the drawing functions require a pointer to an ordering table header, so it's really just for convenience. Next is the packet area (working area for the GPU), then the number of the buffer currently being drawn on (0 or 1), a font id variable used for debug text output, finally we have two members *screenPosX* and *screenPosY*, these are used to offset the drawing environment which allows us to include options in our game such as screen centring.

The rest of the header file just contains the function prototypes for the function definitions in graphics.c. I haven't gone into much depth because all of this stuff is explained in the section dealing with Double Buffering and Screen Resolutions. The main thing to remember is that separate header and source files are employed to encapsulate

functionality specific to an identifiable facility, in this case the set up and use of the PlayStation graphics hardware.

Now I will explain the graphics source file, graphics.c. The first function, *InitGfx()* is pretty much the same function as in the last demo, however some bits have been added or amended.

```
void InitGfx(Graphics *gfx)
{
	// enable pal resolutions
	SetVideoMode(MODE_PAL);

	// initilise the graphics engine
	GsInitGraph(SCRNW, SCRNH, GsNONINTER|GsOFSGPU, 0, 0);

	// set up the display buffers coords
	GsDefDispBuff(0,0,0,SCRNH);

	// set up the ordering table lengths and the headers for them
	gfx->ot[0].length = OTLEN;
	gfx->ot[0].org = gfx->ottags[0];
	gfx->ot[1].length = OTLEN;
	gfx->ot[1].org = gfx->ottags[1];

	// initialise the OTs
	GsClearOt(0, 0, &gfx->ot[0]);
	GsClearOt(0, 0, &gfx->ot[1]);

	// fix the background clearing problem
	GsDRAWENV.isbg = 1;

	// display environment initial position stored in
	// variables to allow moving of display environment
	gfx->screenPosX = 5;
	gfx->screenPosY = 18;

	// set the initial display environment parameters
	// this is required as PAL resolutions require
	// extra initilisation code to display to the
	// full extent of the requested resolution
	GsDISPENV.screen.x = gfx->screenPosX;
	GsDISPENV.screen.y = gfx->screenPosY;

	// change the screen height & width from the libps
	// header to compensate for the PAL resolution
	GsDISPENV.screen.w = SCRNW;
	GsDISPENV.screen.h = SCRNH;

	// removed from LoadFont as multiple font ids
	// will not work due to an extra call to FntLoad()
	FntLoad(960,256);

	// create a font display area
	gfx->fntID[0] = LoadDebugFont(0, 0, SCRNW, SCRNH);
}
```

The first new line of this function sets the *isbg* member of the GsDRAWENV global structure instance (have a look at libps.h) to 1. This is a fix to a problem that the Yaroze has displaying graphics at particular resolutions, read the Yaroze FAQ for further information. The next six lines of code do effectively what four lines of code did in the previous iteration of the function, only this time the screen position is stored so that it can be changed later on. For an explanation of the purpose in this code refer back to the first program and chapter 6; Double Buffering and Screen Resolutions.

Next, as before we load the font from the Yaroze boot disk into VRAM, using the *FntLoad()* function. The next line simply opens a font stream using the helper function *OpenDebugFont()*. To this the origin and width and height of an area of the screen are passed.

Next is the *StartFrame()* function:

```
inline void StartFrame(Graphics *gfx)
{
	// get the index of the current display buffer
	gfx->buf = GsGetActiveBuff();

	// set the buffer pointer to the current display buffer
	gfx->pOT = &gfx->ot[gfx->buf];

	// set the packet work area to the packet area for the current buffer
	GsSetWorkBase((PACKET *)gfx->gpuPktArea[gfx->buf]);

	// initialise the current ordering table for drawing
	GsClearOt(0, 0, gfx->pOT);
}
```

The *StartFrame()* function (as above) is pretty simple, it is merely a transferral of the lines of code that were previously in the main function. Firstly we get the index of the active buffer, then set the ordering table pointer to the ordering table indicated by this index, as said earlier this is for convenience sake. Next the GPU work area is assigned and the ordering table is initialised for drawing.

We also have an *EndFrame()* function, which encapsulates everything that happens after we are finished entering our GPU drawing commands.

```
inline void EndFrame(Graphics *gfx)
{
	// draw all font calls to the screen
	FntFlush(gfx->fntID[0]);

	// execute all non-blocking GPU drawing commands
	DrawSync(0);

	// block until vsync occurs - makes sure everything
	// for current frame is drawn before going onto next
	VSync(0);

	// swap the double buffers
	GsSwapDispBuff();

	// clear the buffer to black
	GsSortClear(0, 0, 0, gfx->pOT);

	// execute ordering table drawing commands for
	// appropriate graphics buffer
	GsDrawOt(gfx->pOT);
}
```

This is basically ripped straight out of the main function in the same way that *StartFrame()* was. First the font calls are flushed, making sure they are drawn. DrawSync() is then called to make sure all non-blocking GPU commands are completed

before the frame ends. Next *VSync()* is called, this makes sure that the GPU completes a full vertical synchronisation from the top of the screen to the bottom before drawing of the next frame commences, this ensures that no glitches in the picture occur such as tearing, where one portion of the screen contains last frame's data and the other part this frame's data. The function finished up with the Ordering Table housekeeping, all this stuff is as described in the main function of chapter 6; Double Buffering and Screen Resolutions.

Both the *StartFrame()* and *EndFrame()* functions are inline functions, this means that at compile time the code inside these functions will be placed where they are called, i.e. in the main loop in main.c. This is purely for tidiness and because these functions are called every frame, so inline-ing them removes some of the function call overhead.

8.3. Sprite Setup

If you have followed previous versions of this tutorial you might remember that sprites were loaded using a single function called *InitSprite()*. Due to issues of modularity and separation of functionality this function has been split into two more logical functions, one that loads the image into VRAM, called *LoadSprite()* (as *LoadImage()* is reserved by the Yaroze API) and a second that initialises a sprite structure (*GsSPRITE*) to an area on VRAM (which would contain the previously loaded image).

Here is the LoadSprite() function from the sprite.c source file, split into two halves:

```
int LoadSprite(GsIMAGE *pImage, char *fileName, u_char *buffer)
{
	// use a rect to set up areas on VRAM for image loading
	RECT rect;

	// this stores the address of the sprite in main memory
	unsigned long *spriteAddr;

	// variable to hold MWbload return value
	// (used to check for errors in file loading)
	int returnVal;

	// load the file into the buffer for reading
	returnVal = MWbload(fileName, *buffer);

	// the specified file was not found or is corrupt
	if(returnVal <= 0)
	{
		printf("Error Loading %s\n", fileName);
		return(0);
	}

	// the specifed file was too big to fit into the buffer
	if(returnVal > sizeof(*buffer))
	{
		printf("Error Loading %s\nBuffer too Small\n", fileName);
		printf("Increse buffer size\n");
		return (0);
	}
```

This function takes as arguments a pointer to a *GsIMAGE* structure, a filename and a buffer that the data will be loaded into. This is where the Codewarrior and the GNU code versions differ, GNU loads everything, prior to execution, into the PSX memory as part

of a loading script, whereas the Codewarrior code treats loading as part of execution. The GNU version accepts a *GsIMAGE* structure pointer and an address, which is an unsigned long pointer.

Basically the Codewarrior version (as shown) requires a buffer, which is just a character (byte) array of the size necessary to hold the image.

The Codewarrior version uses a function called *MWbload()* which is provided as part of the Codewarrior debug library, it accepts the filename of the file to be loaded and a buffer into which it is loaded. The code is fairly self explanatory and the comments outline the code's purpose.

```
        // grab the address of the buffer for sprite loading
        spriteAddr = (unsigned long *)buffer;

        // increment address to point past TIM header
        // (+4 bytes i.e. sizeof(unsigned long))
        spriteAddr++;

        // put the TIM info inside the GsIMAGE structure "image"
        GsGetTimInfo(spriteAddr, pImage);

        // set an area in VRAM to load the image onto
        setRECT(&rect, pImage->px, pImage->py, pImage->pw, pImage->ph);

        // load the image into the specified area in VRAM
        LoadImage(&rect, pImage->pixel);

        // if the image has a colour lookup table
        if((pImage->pmode >> 3) & 0x01)
        {
                // set the area in VRAM to load the CLUT into
                setRECT(&rect, pImage->cx, pImage->cy, pImage->cw, pImage->ch);

                // load the CLUT onto the VRAM
                LoadImage(&rect, pImage->clut);
        }

        // wait for the completion of all non-blocking GPU commands
        DrawSync(0);

        // return success
        return(1);
}
```

The second half deals with reading the data and placing it on the VRAM. The first line is the typecast which basically puts a handle on the data so we can access it in the same way the GNU version would (note that the GNU version does not contain the loading section because the data is already loaded and an address is passed). The second line skips the first 4 bytes of the data, the image header info. We are incrementing once, but because its an unsigned long pointer and an unsigned long is 4 bytes big, we are skipping 4 bytes.

The next line retrieves the TIM information and stores it inside the passed GsIMAGE structure. Next, we set an area on the VRAM into which the image will be loaded, this is done using the *setRect()* macro. This rectangle is only used so that in the next line when *LoadImage()* puts the pixel data onto the VRAM it knows where and how big this area is.

Next we have to check if the image has a CLUT (Colour Look-Up Table). We have to utilise a bit of maths here… first we have the *pmode* value. The value for a 4-bit TIM is 0x08, 8-bit is 0x09, 16 bit is 0x02 and 24-bit is 0x03, so respectively 4 bit = 8, 8 bit = 9, 16 bit = 2 and 24 bit = 3, contrary to what the Library Reference has to say on this matter (0 = 4 bit, 1 = 8 bit, 2 = 16 bit, 3 = 24 bit).

By saying; 0x08 which is the same as 8, eight in binary is 1000, this bit-shifted right 3 times is 0001, which is the same as 1. One AND 0x01 (which is hex for 1) is one, so *if(1)* would execute. If you perform the same bit of maths with the 8, 16 and 24-bit modes you will find that 8 executes but 16 and 24 don't.

Anyway, if this is the case then we have to load the CLUT into the VRAM. So first we have to prepare the *rect* to hold the CLUT (remember at this stage the image data i.e. *imageInfo.pixels* has already been loaded into the VRAM so we can re-use the *rect* for the CLUT). Then we want to load the CLUT into the VRAM in the area by calling the *LoadImage* function with our source and destination.

If the image has a palette the VRAM area is set again and the palette is loaded onto the VRAM using *LoadImage()*. In the Codewarrior version we have to return 1 at the end to denote a successful execution, the GNU version is of type void because there are no error codes to deal with in that version.

At the end of the function, we have a *DrawSync()* call which waits until the *LoadImage()* function has finished. This has to be done, because according to the Yaroze User Guide pages 50 and 51, the *LoadImage()* function is a non-blocking function, which means that other tasks can go on while it is executing. We do not want to exit the function until the *LoadImage()* function has finished so *DrawSync()* comes to the rescue and ensures that the function does not finish until the frame buffer command has been completed.

After the *LoadSprite()* function has been called we are left with a *GsIMAGE* structure that holds all the data we need to create new sprites from this data. So now we move onto the *InitSprite()* function, in two halves again.

```
void InitSprite(GsSPRITE* pSprite, GsIMAGE *pImage)
{
	// check for bit depth
	switch(pImage->pmode)
	{
		// 4 bit image quarter of actual image width
		case FOURBITIMAGE:
			pSprite->attribute = FOURBITCOLOUR;
			pSprite->w = pImage->pw * 4;
			pSprite->tpage = GetTPage(0, 0, pImage->px, pImage->py);
			break;

		// 8 bit image half of actual image width
		case EIGHTBITIMAGE:
			pSprite->attribute = EIGHTBITCOLOUR;
			pSprite->w = pImage->pw * 2;
			pSprite->tpage = GetTPage(1, 0, pImage->px, pImage->py);
			break;

		// 16 bit image takes up full image width
```

```
                default:
                        pSprite->attribute = SIXTEENBITCOLOUR;
                        pSprite->w = pImage->pw;
                        pSprite->tpage = GetTPage(2, 0, pImage->px, pImage->py);
                        break;
        }
```

InitSprite() requires a pointer to a *GsSPRITE* and a pointer to a *GsIMAGE* structure. The information about the image resides in the *GsIMAGE* pointer and we want to use it to setup the *GsSPRITE*.

The first thing we do is look at the *pmode* value of the *GsIMAGE* structure. You will notice a few defined macros here, such as FOURBITIMAGE, FOURBITCOLOUR, etc. These basically make the code easier to read by removing the 'magic numbers', you will find these values defined in the sprite.h source file. For example FOURBITIMAGE is equal to 8 as specified in sprite.h:

```
#define FOURBITIMAGE (0x08)
```

Basically different *pmodes* effect how the image should be set up. The attribute is set which will tell the GPU what bit-depth the image is when drawing. As a side note, one which will come in useful in the future, you can set different image appearances by using this 'attribute' value. There is a list of attributes in the sprite.h file ready for use, so if you wanted a 50% transparent sprite whose RGB brightness values you could alter you would add the line:

```
pSprite->attribute = (RGB_ADJUST_ON | SEMITRANS_ON | HALFBACK_PLUS_HALFFRONT);
```

It is not recommended to put such code in the generic *InitSprite()* function, as such features should be enabled on a per-sprite basis. You can change the attribute safely after the *InitSprite()* function has been called, just remember what you enabled and what values are there. You can always test what sprite functionality is enabled by using a piece of code similar to the following:

```
if(pSprite->attribute & RGB_ADJUST_ON)
{
        // sprite rgb brightness adjustment is on
}
```

If the image is 4-bits then the width is a quarter of what it should be, so the width value is multiplied by 4 to get the correct width. 16 bits would have the image at full width and you would not have to expand the image. 8 bits is half the image width, hence the multiplication by two in the EIGHTBITIMAGE case. The width of a 4-bit image has to be a multiple of 4 and an 8-bit image has to be an even number. If this is not the case then you will get vertical lines through your image as described in the Yaroze FAQ. Read the Adobe Acrobat document, fileformat.pdf page 31, which can be obtained from the Net Yaroze Servers.

Next *GetTPage()* is called which returns the index of the 256x256 texture page that the image is on in VRAM, the PlayStation was designed with texture pages, I suppose to

make it easier to organise images in VRAM. The first parameter to *GetTPage()* is the bit mode, next is the transparency rate and the last two are the x, y coordinates in VRAM of the image; *pImage->px*, *pImage->py*, which were set by our call to *GsGetTimInfo()* in *LoadSprite()* earlier. Here is the second part of *InitSprite()*.

```
        // set sprite default attributes
        pSprite->x = 0;
        pSprite->y = 0;

        pSprite->h = pImage->ph;

        // sprite offset adjust for animation
        pSprite->u = 0;
        pSprite->v = 0;

        // default brightness RGB
        pSprite->r = 128;
        pSprite->g = 128;
        pSprite->b = 128;

        // CLUT position in VRAM
        pSprite->cx = pImage->cx;
        pSprite->cy = pImage->cy;

        // ensure that image is of even width
        pSprite->mx = pSprite->w/2;
        pSprite->my = pSprite->h/2;

        pSprite->scalex = ONE;
        pSprite->scaley = ONE;

        pSprite->rotate = 0;
}
```

Now we want to set up the rest of the *GsSPRITE* structure, this consists mainly of assigning default values.

We set the height, which is pretty much self-explanatory. The page offset needs a bit of explaining though. The *u*, *v* page offset variables are initially set to the origin of the image i.e. the top left. These can be used to pick out certain parts of the image to ‘clip’ and display. The width of the sprite clipping rectangle is determined by the width and height values, we will go over this in a later tutorial.

Next we specify what brightness the image is initially set at with the *r*, *g*, and *b* components of the *GsSPRITE* structure. The values 128 are medium brightness of each colour. Setting them to 0, 0, 0 would turn the image black and setting them to 255, 255, 255 would make the image completely white. In order to be able to change these values dynamically at run-time you would have to enable the brightness regulation in the attribute part of the *GsSPRITE* structure, RGB_ADJUST_ON and RGB_ADJUST_OFF.

The Colour Look-Up Table coordinates are set next, this is the location of the palette that is needed if the image is 4 or 8 bits per pixel. This information is stored in the *GsIMAGE* structure that was set by the *GsGetTimInfo()* call in the *LoadSprite()* function.
The *mx* and *my* components set the centre of rotation for the sprite, here they are set to the centre of the image; you can of course change them to any point in the image. Then

we set the scaling components for the x and y axis. These are set to 4096, which means 1 in the Yaroze native 20-12 fixed point precision integer number format. If you were to set this to 2048 the image would be squashed to ½ its original size, and, respectively, 1024 would squash the image to ¼ of its original size.

Finally we set rotation to 0, meaning no rotation. You can turn rotation on yourself by setting the attribute member of the *GsSPRITE* structure to include the ROTSCALE_ON value, this enables scaling and rotation, ROTSCALE_OFF disables this. Remember though, as the manual says, 4096 equals one degree (20-12 fixed point values again). So to rotate the sprite 180 degrees set *pSprite->rotate* to 4096 * 180.

That is the sprite.c and sprite.h files out of the way, now we can concentrate on the heart of the program, main.c.

Firstly you will notice this line:

```
// player sprite's drawing priority
#define PLAYER_OT_PRI  2
```

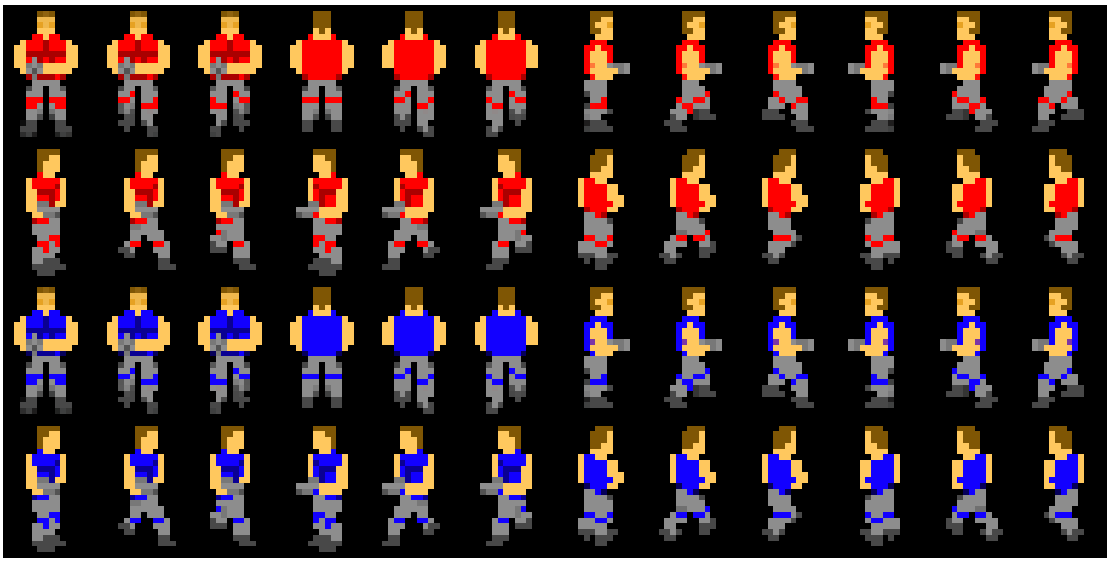
This is a ‘device’ that I will use from here on out, due to the complexity we will have with organising the priorities of sprites on the ordering tables. If we can keep all our priority numbers in one place then we will be able to manage them better, instead of hunting around for multiple drawing calls, it will all become clear when we start using backgrounds.

9.0. Sprite Animation

9.1. Creating Animation Frames

The first thing you want when you are making an animated character for a game is a set of frames of animation. This can be an extremely tiresome and drawn out process; trying to get each frame looking right and keeping it within a set amount of frames… lets face it the VRAM is at a premium on the Yaroze, but that's what video gaming is all about – working to constraints. Remember also that with our bit mode (4-bit) we can fit a fair amount of stuff in due to the way the images are stored in V-RAM.

You can create your frames any way you want… you can create a model and have a pre-rendered walk cycle (all you need is the 3D app to pump out your rendered frames) or you can hand draw them and scan (with a scanner) or trace (with a digitiser) them in then touch them up. I have used the latter method and the results are in part4\gfx\ mananim.tim. I will be using the frames, as shown below, for this part of the tutorial.



9.2. The Animation Algorithm

The method we will be using for sprite animation is the image-offset method. If you can remember back to when we set up the *GsSPRITE* structure in the Manipulating Sprites part of the tutorial we set up both the *u* and *v* attributes of the *GsSPRITE* structure to zero so as to point to the top left of the texture page, which also happened to be where the origin of the loaded sprite was (*u*, *v* coordinates are relative to the texture page origin, not the image origin).

In order to implement animation what we want to do is change the sprite width and height to the size of one frame after we have initialised the sprite (initialising a sprite using *InitSprite()* sets the width and height to the dimensions of the whole sprite) then modify the *u* and *v* values to point to the next frame of animation every few frames.

The algorithm that we are going to use may appear complex at first, but given time it will become second nature (we are going to use the same method for custom backgrounds later on). So here it is, in pseudo-code form:

```
U =  (((playerDirection * framesPerAnim) +
     (frame)) % numColumns) * spriteWidth
V =  (((playerDirection * framesPerAnim) +
     (frame)) / numColumns) * spriteHeight
```

To start explaining this algorithm, take a look at the image above. It is composed of two colours, red and blue (one for each player of a two-player game). Each colour has eight directional animations, up, down, left, right, and diagonals there of. Lastly, each directional animation is composed of three frames, so 2 players * 8 directions * 3 frames = 48 frames in total.

Now look at the algorithm… we have a playerDirection variable which tracks the direction in which the player is moving (hence which animation to play), this is an enumerated type (‘enum’ – check your C textbook) with values zero to seven.

We have a framesPerAnim variable which is a constant of three (since there are three frames per directional animation). These two multiplied gives us the starting index frame of our animation set.

The frame variable will take a little more explaining… it is used as an offset into the current directional animation that is to be played, so it’s value can never be more than 2, since there are only three frames (remember zero indicing). The frame variable is calculated using the following algorithm:

```
frame = ((frameCounter++) / frameRate) % framesPerAnim
```

The frameCounter variable is incremented every frame and basically indicates what frame the program is currently processing. The frameRate variable is the number of frames we wish to elapse before we change the animation frame, this is so that the animation does not play at full speed and hence gives a better illusion of movement. Dividing frameCounter by frameRate means the animation is slowed down, try tampering with the frameRate variable to find an animation rate you are happy with. The last part, modulusing by framesPerAnim makes sure, as stated before, that the frame variable is always between 0 and 2.

The next part of the equation differs between the *u* and *v* calculations. The *u* coordinate calculation used a modulus operator. We will use an example to demonstrate the effect of the modulus. If for example the first part of the equation ((playerDirection * framesPerAnim) + (frame)) gives 17, this will exceed the number of columns we have, so using the modulus simply gives us a frame within the 0 to 11 range, look at a C text book to see how the modulus operator actually works. Lastly with the *u* coordinate we multiply

what we now have (the index to the column that our desired frame is in) by the frame width (in pixels) which gives us the final *u* offset.

The calculation of the *v* coordinate acts similarly to the *u* coordinate except what we want is the row the frame is in so when the first part of the equation is greater than the number of columns there are we want to go onto the next row down. So as in the previous example if we have 17, a division by 12 (the number of columns) gives 1 (in whole number terms), which is the second row down, which is what we want. We then multiply the row number we have by the frame height to get the final *v* offset.

9.3. The Source Code

The source code for animation has been integrated into a ‘player’ setup and can be found in player.c and player.h. This is a grouping of functionality specific to the player character so I thought it best, to aid modularity, to stick the player initialisation and drawing all in one group of functions.

You will notice a player structure in the code, we will look at this briefly to see what it contains and what relevance it holds to what we are trying to achieve.

```
typedef struct
{
	// holds the sprite information
	GsSPRITE spr;

	// used in animation to determine direction
	enum directions dir;

	// used in animation to determine which frame
	// inside each 3 frame set the sprite u,v should equal
	u_short frame;

	// flag to tell wither to animate or not each frame
	u_short anim;

	// the player number in multi-player
	u_char playerNum;
}player;
```

The first member is a *GsSPRITE* structure, this is initialised using *InitSprite()*.This is just so that we have a local copy of a *GsSPRITE* structure (instead of a global *GsSPRITE* instance as we have had in previous programs) which will contain the latest frame’s u, v cords as well as the player position on the screen etc…

The next members are all related to animation.

- First we have an instance of the enumerated type directions. This allows us to store the current direction in which the player sprite is moving, so we know which three-frame animation cycle to choose.
- Next we have the frame variable; this tells us which frame of the current 3 frame animation set we should be displaying, in the range 0 to 2.
- Next is a variable we use as a flag, anim. This basically tells the animation code wither it should be updating the animation or not, because if the player has not pressed any directional buttons to move the character then it should not animate, hence we set this flag when no directional input is received.
- The playerNum, variable simply stores the number of the player; this is either 1 or 2. This could be increased for more players of course.

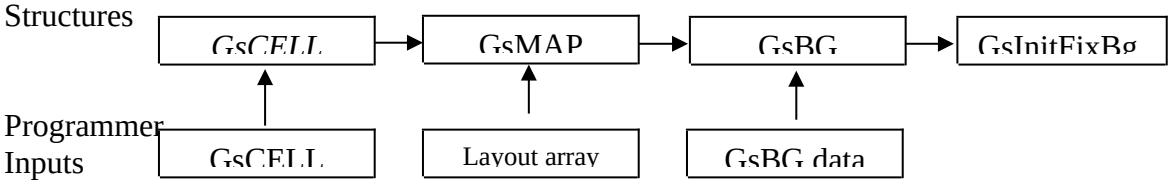
10.0. Background Creation

10.1. PlayStation Handling of Backgrounds

First we will discuss the theory of backgrounds then the way that the PlayStation handles backgrounds before we put it into code.

The PlayStation has a BG handler, *GsBG* , which basically holds all the data after set-up of all the tile images that the background will be made of. A map, *GsMAP*, of where the images are to be placed is needed so that the Yaroze knows where to place each of the image tiles in the background. This map holds a reference to the actual details of individual image tiles, which are *GsCELL*s.

If you look at your manual you will see (or maybe not) that these are the three data structures concerned with implementing backgrounds. The first one is *GsCELL*. This data-type holds elements, which describe an individual BG tile, such as the image attributes, the inversion control flag (this flips the image along the horizontal or vertical plane), u, v texture page offsets and the texture page number itself. An array of these is needed for the amount of image tiles the background is made up of. The *GsMAP* holds the details of the make-up of the background, i.e. where all the tiles go. This contains a pointer to the array of *GsCELL*s. Also very importantly, a pointer to the array, which holds the layout of cells, this is merely numbers so don't get scared or anything! The big daddy background handler is *GsBG*. This holds universal variables, by that I mean, attributes that apply to all the tile images in the BG. These include the x, y top left hand co-ordinates of where it is to be displayed, width and height values, image attributes (identical to those of *GsSPRITE*) etc… but most importantly is the pointer to the *GsMAP* data. The GsBG, as you might have guessed by now, looks to this for the layout of the cells (tile images).



This diagram emphasises the linearity of the set-up of a background on the Yaroze. At each stage you have a certain amount of data coming in from the programmer as well as the last structure that was defined. The *GsInitFixBg16* function will initialise the BG once we have finished setting it up. Once the set-up is complete we can display the BG in the main loop by calling *GsSortFixBg16* with our initialised BG, the ordering table, work area and the priority (how soon it is to be drawn – remember z sorting above) passed to it.

Now that we have cleared up the theory, lets see how easy it is to put into code…

Once again I will put in all the code required so that you can easily spot the new stuff and where it should go. So first we start off with the 'includes' and 'defines'.

```
#include <libps.h>            // PlayStation Specific Library

#include "pad.c"              // pad handling functions

#define SCREEN_WIDTH   320            // holds the screen width
#define SCREEN_HEIGHT  256            // holds the screen height

#define NUM_TILESX     20                    // SCREEN_WIDTH/16 = 20
#define NUM_TILESY     16                    // SCREEN_HEIGHT/16 = 16

#define PACKETMAX      2048                  // maximum GPU packets
#define PACKETMAX2     (PACKETMAX*24) // used to set drawing environment

#define OT_LENGTH      9                     // ordering table length

#define BARNEY_ADDR    (0x80090000)   // location of our sprite in VRAM
                                                          // this must be the
same as in the auto2.sio file
#define TILES_ADDR (0x80090550)              // location of our backdrop image
```

Pretty standard stuff as usual, new additions are our ‘defines’ for *NUM_TILESX* and *NUM_TILESY*. These are the number of tiles that span the width and height of the screen. The width and height of an individual tile is 16, this is a non-negotiable number (unless you make your own background handling procedures – we will look into this later). Its set and that’s it, you can’t define bigger or smaller tiles. It doesn’t make much difference to us anyway. We don’t have to make individual images for every tile in the BG. Just make a big backdrop load it in to VRAM and use the texture page offset *u*, *v* co-ordinates in *GsCELL* to point to the next 16x16 tile. Anyway, these two defines are used when telling *GsMAP* the number of tiles horizontally and vertically on the screen. The next new line is the address of our backdrop image. This is of course so we can access the image to put it into VRAM for use.

```
int frameNo;
int PLAYING = 1;

GsOT WorldOT[2];                      // ordering table headers
GsOT_TAG WorldTags[2][1<<OT_LENGTH];  // the ordering tables units

static PACKET packetArea[2][NO_OF_SPRITES*sizeof(GsSPRITE)]; // GPU PACKETS AREA

GsSPRITE sprite;              // this is basically a header structure for the TIM

GsBG bgData;                  // declare background handler
GsMAP bgMapData;              // declare a BG composition map
GsCELL bgCellData[16];        // declare cells for amount of 16x16 tiles

unsigned short mapLayout[16]; // holds the layout of the BG

u_long bgWorkArea[(((SCREEN_WIDTH/16+1)*(SCREEN_HEIGHT/16+1+1)*6+4)*2+2)];
```

Here we have the usual ‘global’ variables. However you will spot the new lines. The first one is our *GsBG* instance to handle the complete background. The next is our *GsMAP* instance, which holds the configuration of the map. Then we have the *GsCELL* instance. If you have read the above explanation, everything should be clear. Lastly we have the *mapLayout* array, this basically hold the arrangement of how the cells will appear on the screen. We will see this working later on in the *main* function. The next line sets up the size of the working area where the BG is placed while the Yaroze is busy calculating

what the BG should look like on the screen. Tiles are sorted away from the players view and this is the area of memory in which this is done. The actual size of this seems unclear I have asked around as to what size it should be, pretty much everyone stated the same calculation but couldn't explain it.

The algorithm as it stands is *(((SCREEN_WIDTH/CELL_WIDTH+1) * (SCREEN_HEIGHT/CELL_HEIGHT+1+1) *6+4) * 2+2)* quite a fiddle eh? Anyway, with our set up you will find that it equates to 3682. I have been playing around with the size of this and have come upon the formula (20 x 16) * 12 which is 3840 – close enough. I though of this as number of cells that fit into the horizontal resolution times the number of cells that fit into the vertical resolution times the bit index of the image (plus 4 just for extra space just to be safe) I admit that this is a very 'stab in the dark' situation and that any formula could produce this result with vague relevance to the set up but until we actually hear it straight from the horses mouth (i.e. Sony) I won't know but I hope to be able to tell you in future increments of this tutorial.

The next two functions *Initialise* and *InitSprite* remain the same as the above program listing. So I won't list them here… why, that would just be wasting space!

What we do need now is a function that will load a background image into a *GsIMAGE* structure for us to use in the compilation of our background.

```
GsIMAGE *ReadTIM(u_long *addr)
{
	static GsIMAGE tim;
	RECT rect;

	// skip id and initialise image structure
	GsGetTimInfo(++addr, &tim);

	// prepare RECT to hold TIM image
	rect.x = tim.px;
	rect.y = tim.py;
	rect.w = tim.pw;
	rect.h = tim.ph;
	// load the image into the RECT on VRAM
	LoadImage(&rect, tim.pixel);
```

Righty then… what we want is a pointer to a *GsIMAGE* structure to return to the place we call it from. We use a pointer because it means we don't have to transfer data around inside memory we can just reference the address of the data and it will look to that location to find the data – some stuff you should find in a good C or C++ book. The *GsIMAGE* is declared as static because we want to be able to access the information after the function has ended. Next we declare a *RECT* structure into which we are going to hold the image data and the CLUT for transfer into VRAM.
Now we want to read the information in the image in memory to the *GsIMAGE* structure ready for transfer to VRAM, so we make a call to *GsGetTimInfo*. Next we have to set up the area in VRAM to where the image will be transferred by setting up the *RECT* to the dimensions of the image the *x* and *y* members of the *RECT* structure are the location in VRAM to where the image will be transferred (this was set up in TimTool3). After it is set up we load it from memory to VRAM using the call to *LoadImage*. The *rect* argument

is the area that we have just set up in VRAM and the data we want to load in is the *tim.pixel* argument. Now for the next bit…

```
        DrawSync(0);   // wait for LoadImage function to finish executing

        // check if CLUT exists and transfer it to VRAM
        // prepare RECT to hold CLUT
        if( (tim.pmode >> 3) & 0x01 )
        {
                rect.x = tim.cx;
                rect.y = tim.cy;
                rect.w = tim.cw;
                rect.h = tim.ch;
                // load the CLUT into the RECT in VRAM
                LoadImage(&rect, tim.clut);
        }

    DrawSync(0);       // wait for LoadImage function to finish executing

    return(&tim);      // return the initialised image data

}// end ReadTIM
```

The first call is to *DrawSync* this will just wait until the GPU has finished the previous *LoadImage* call. Now we have to load in a CLUT. We are using a conditional statement to check that the image we are loading is 8-bit (if you want a 4-bit background change the 3 to a 2… do the math its in the earlier section 'Getting the TIM on the Screen'). If it is in fact 8-bit then we want to set up a region on the VRAM where the CLUT will go. Then it is loaded into the VRAM with the call to *LoadImage*.

Next we perform a *DrawSync* so that execution will pause until the *LoadImage* function has finished. At the end of the function we return the *GsIMAGE* structure to the caller.

Right now its time for the BG functions. The initialisations of the three parts of the BG structures have been divided into three distinct functions as below.

```
void InitBG (void)
{
        bgData.attribute = 1<<24;
        bgData.x = 0;
        bgData.y = 0;
        bgData.w = SCREEN_WIDTH;
        bgData.h = SCREEN_HEIGHT;
        bgData.scrollx = 0;
        bgData.scrolly = 0;
        bgData.r = 128;
        bgData.g = 128;
        bgData.b = 128;
        bgData.map = &bgMapData;
        bgData.mx = 0;
        bgData.my = 0;
        bgData.scalex = ONE;
        bgData.scaley = ONE;
        bgData.rotate = 0;
}
```

This function is fairly straightforward. First we assign the attribute, which is in this case setting the image attribute to 8 bit through use of the bitwise operator (this is the same as in the *InitSprite* function in the Getting the TIM on the Screen section). We then set the x and y start coordinates of the BG in this case it's the top left of the screen. We then set

the height and width to be that of the screen. We set the scrolling values to 0, se the brightness of each colour to their default values (i.e. 128 half of full brightness (255)). We link in the map data from the *GsMAP* structure, which is initialised next. The next two lines set the centre of rotation and scaling for the whole BG this is set to 0, 0 i.e. the top left of the screen. We set the scaling of the BG to ONE, which is defined as 4096 this means in the PlayStation way of speaking a whole or 1. This will set the scale value of the default value i.e. not scaled. Finally the rotate value is set to 0.

```
void InitMap (void)
{
        bgMapData.cellw  = 16;
        bgMapData.cellh  = 16;
        bgMapData.ncellw = 4;
        bgMapData.ncellh = 4;
        bgMapData.base   = &bgCellData[0];
        bgMapData.index  = &mapLayout[0];
}
```

This is the function for setting up the map data. This will set up the composition of the map data in the BG.

First off we set the call height and width to 16. The Sony libraries will only accept cell sizes of 16 (bummer, but that is one of the reasons why we will implement our own BGs later). The next two assignments are the number of cells high and wide our map will be. This map will of course be repeated up to the maximum width and height of the BG. When you are making your BGs in a paint program you have to make sure it is a multiple of 16. The multiplier is what you set *ncellw* and *ncellh* to.

For example: if you have a BG image of 64 width and 32 height we would set up *ncellw* and *ncellh* as below because 64/16 = 4 and 32/16 = 2:

```
        bgMapData.ncellw = 4;
        bgMapData.ncellh = 2;
```

or for another example if we had a BG map of 128 width and 64 height we would set up *ncellw* and *ncellh* as below:

```
        bgMapData.ncellw = 8;
        bgMapData.ncellh = 4;
```

You'll have the idea by now, hopefully. We then link in the cell data to the map structure as seen in the linear diagram earlier. This one is simple as it is just a pointer to the base address of the cell data. Next we have a more complicated assignation, the *mapLayout* array is an array for the structure of the BG. This will be covered in the explanation of the *main* function a little later.

Right now we have our final BG function, which reads the cell data in (using *ReadTIM*).

```
void InitCELLs (void)
{
        GsIMAGE *timImage;
        u_short texPage;
        u_short CLUT;
```

```
        int count=0,
            x, y;

        // get info
        timImage = ReadTIM((u_long *)TILES_ADDR);    // read in TIM
        texPage = GetTPage(1,0,timImage->px,timImage->py); // get TPAGE
        CLUT = GetClut(timImage->cx,timImage->cy);   // get CLUT

        // rows in tpage
        for( y=0; y<=3; y++ )
        {
                // columns in tpage
                for( x=0; x<=3; x++ )
                {
                        bgCellData[count].u     = x*16;       // offset x for every column
                        bgCellData[count].v     = y*16;       // offset y for every row
                        bgCellData[count].cba   = CLUT;
                        bgCellData[count].flag  = 0;
                        bgCellData[count].tpage = texPage;
                        count++;
                }// end for x
        }// end for y
}
```

Ok, first we have a pointer instance of a *GsIMAGE* structure; this will be used to store the address to the image we read in using *ReadTIM* further down. We also have a texture page variable to store the texture page in Yaroze VRAM where the image is. Also we have a variable to store the CLUT for the image. These are both u_shorts as these are what the *GetTPage* and *GetClut* functions return. Then we have some generic variables used for setting up the cell data structure array as declared in the declarations code earlier. So the first call we have to make is to *ReadTIM* to get the TIM data into the VRAM, we pass in the address in main memory (the address we used to send it to when we upload the data to the Yaroze i.e. the same one as in the auto.sio file) and it does the rest, our pointer to GsIMAGE data is filled with the address of where the image data was loaded to. Next we get the texture page for assignation to the cell structure later. Read the manual for a description of the arguments – its rather straight forward. Now we have to find where the CLUT was loaded so that we can tell the structure where it is so that is uses the correct palette when displaying the BG.

Finally we can actually set up the cell data array. We use two *for* loops to go through; first the row then each column in that row of 16 x 16 cells hence the *y for* loop coming first. Inside the two *for* loops you will notice that each time we set the *u*, *v* texture page offsets to +16 or *x* or *y* times 16 each time, this is so that the correct 16 x 16 cell can be selected. You see… we have a whole image but the BG is made up of 16 x 16 sections of that image. The next three lines are simple, just look at the manual to get an idea of what is going on. Lastly, as we go along we increment the count variable as we set up the next structure in the cell structure array.

This set up does not flip the cells, however if you wanted to flip the cells at all you would use one of the following assigned to the *flag* member of the cell structure:

Vertical Flip:	[1<<0]
Horizontal Flip:	[1<<1]
Flip Both:	[1<<1]+[1<<0]

The *DealWithPad* function is the same as before so I will not waste space by putting it in here. Instead we will jump directly to the main function.

```
void main(void)
{
	int vsync = 0,
	    side, i;

	// layout set-up algorithm
	for(i = 0 ; i <= 16 ; i++)
	{
		mapLayout[i] = i;
	}

	Initialise();

	// initialise BG
	InitCELLs();
	InitMap();
	InitBG();

	InitSprite();

	GsInitFixBg16(&bgData,bgWorkArea);

	side = GsGetActiveBuff();

	while(PLAYING)
	{
		FntPrint("frame: %d\n", frameNo);

		frameNo++;		// increment frame count

		DealWithPad();

		GsSetWorkBase((PACKET *)packetArea[side]);

		//clear out the present OT ready for drawing
		GsClearOt(0,0,&WorldOT[side]);

		// register the BG in the OT
		GsSortFixBg16(&bgData,bgWorkArea,&WorldOT[side],1);

		// register the sprite in the ordering table
		GsSortFastSprite(&sprite,&WorldOT[side],0);

		DrawSync(0);
		vsync = VSync(1);
		VSync(0);		// get the vertical sync interval

		FntPrint("HSYNC: %d\n", vsync);

		GsSwapDispBuff();
		GsSortClear(0,0,0,&WorldOT[side]);

		// draw the current off screen buffer to the screen
		GsDrawOt(&WorldOT[side]);

		side = side^1; // change side to 1 or 0
		FntFlush(-1);  // print all font calls
	}
	ResetGraph(0);
}
```

Scanning the function you will see very few lines added. First at the initialisation stage we have the set up of the *mapLayout* array.

If you take a look at the manual it will say that the layout array must be the same size as *ncellw* x *ncellh* i.e. the number of 16 x 16 cells that make up your BG image. This is therefore NOT a two dimensional array as might have been more logical but it will most likely be this way because its easier. As I'm sure you noticed earlier we have an array declared globally, it will accept 16 elements the way we have declared it. What we have done with the *for* loop is merely assign each element 1 to 16 with a number from 1 to 16 so array element one is 1, two is 2 etc… This may seem simple but this is only because we have a simple set up. More complex set-ups can be quite difficult to work with. The general idea is to have an array for the map layout the size of the area you are going to build (in 16 x 16 tiles). Then you would reference the number the tile appears in your BG image you intend for using.

So say for example you want one side of the screen to be filled with one 16 x 16 tile and the other side to be filled with another 16 x 16 tile you would have a BG image either 16 x 32 or 32 x16.

For arguments sake lets stick with the current resolution we have which is 320 x 256. We have 16 x 16 tiles so we would have 16 tiles vertically as 256/16 is 16 and 20 tiles horizontally as 320/16 is 20. To get the number of tiles in the screen we say 20 x 16, which is 320. So divide this in half to get the first half of the screen and we have 160.

Now your array would be full with 1s for one half (160 elements) of the array and 2s (the other 160 elements) for the next part of the array. Its very simple when you get the idea, from then on its just time consuming setting them up the way you want.

As you can see we can fill up a screen quite easily with quite a small BG image to work with. Of course it does not provide much variety but it frees up VRAM.

Now we have a working program and if you run the file 'auto3-1.sio' in the part3\part3-1 directory you will see the BG working. You will also see that the player image still has the black background – this looks ugly and it has to be removed.

10.2. Setting Sprite Background Transparency

The good news is we do not have to touch any code to do this. TimTool 3 being the very useful and brilliant tool it is will allow you to remove the black background from the image very simply.

Ok, first start up your TimTool 3 application and load up the image you want to get the black background off. The image should be aligned in the VRAM (if its not then align it making sure it overlaps no other images you will be using in the same program) so click on the actual image, this will select the image and put cross-hatching over it to signify it is selected. Click on menu item 'CLUT / STP Options -> CLUT Options' this will bring up a window with four main panels. Look to the bottom left panel entitled 'Original Palette' you will see the palette of the image and three rectangular buttons above this.

- Button 1: Set Image to normal, nothing is transparent.

- Button 2: Set Image to ‘Everything except for black is transparent’ (used for masking).
- Button 3: Set image to ‘Black is transparent’ (the one we want and the one you will most commonly use)

We want the background (black) to be transparent so press the third button. The preview will update to show the image without the black background. This is now how it will appear on the Yaroze screen. Exit the window by pressing ‘OK’ then select ‘TIM Options -> Save Modified TIMs’ its as simple as that.

The updated soldier image is in the part3\part3-2 folder, run ‘auto3-2.sio’ it and you will see that the black background on the soldier image has been removed.

10.3. Background Scrolling

There are many different way to scroll a background and indeed scrolling a background can mean a lot of different things to a lot of different people. The background scrolling I will be looking at is scrolling a background in relation to the position of the player character, i.e. the player character will always be in the middle of the screen.

To have this functionality in our code is not going to take that much work. The first thing we will have to do is increase the size of our background as we currently only have enough background for one screen. What we have now is a 320 x 256 pixel background. We want to have a bit of scrolling but just enough to show how it is done. I have decided to make a 2 x 2 screen layout, i.e. 4 screens altogether, so in the end we will have a 640 x 512 pixel background. The change comes into the *InitSprite()*, *DealWithPad()*, *InitBG()* and *main()* functions as follows.

First the *InitSprite()* function amendments.

```
        sprite.x = (SCREEN_WIDTH / 2) - (sprite.w / 2);
        sprite.y = (SCREEN_HEIGHT / 2) - (sprite.h / 2);
```

To avoid putting the whole *InitSprite()* function in again (which, if you remember, is huge) I have listed only the lines to be put in. Take the above lines and insert them immediately after the *sprite.h = imageInfo.ph;* line. These two lines just set up the sprite of our soldier to be in the middle of the screen.

The *DealWithPad()* function; its all been included to show where the new stuff is and where the old stuff has been taken out.

```
void DealWithPad (void)
{
        long pad;

        pad = PadRead();

        if (pad & PADstart && pad & PADselect)       // quit
        {
                PLAYING = 0;
        }
```

```
        if (pad & PADstart)                             // pause
        {
                while (pad & PADstart)
                {
                        pad = PadRead();
                }
        }

        if (pad & PADLup)                       // d-pad up
        {
                bgData.scrolly -= 2;
        }

        if (pad & PADLdown)                     // d-pad down
        {
                bgData.scrolly += 2;
        }

        if (pad & PADLleft)                     // d-pad left
        {
                bgData.scrollx -= 2;
        }

        if (pad & PADLright)                    // d-pad right
        {
                bgData.scrollx += 2;
        }
}
```

The statements that have changed are the conditionals for testing the d-pad inputs. Instead of moving the soldier sprite, we simply scroll the background. Note that everything is in relation to the world, meaning that the screen does not scroll over the background as you may have thought, the background scrolls around independently, in the very same way that the soldier sprite moves around, by modifying x, y position values, the only difference is that the x, y values for the background are called scrollx and scrolly.

```
void InitBG (void)
{
        bgData.attribute = 1<<24;
        bgData.x = 0;
        bgData.y = 0;
        bgData.w = SCREEN_WIDTH * 2;
        bgData.h = SCREEN_HEIGHT * 2;
        bgData.scrollx = 0;
        bgData.scrolly = 0;
        bgData.r = 128;
        bgData.g = 128;
        bgData.b = 128;
        bgData.map = &bgMapData;
        bgData.mx = 0;
        bgData.my = 0;
        bgData.scalex = ONE;
        bgData.scaley = ONE;
        bgData.rotate = 0;
}
```

Four changes have taken place. The first and second being the addition of the **2* after *SCREEN_WIDTH* and *SCREEN_HEIGHT* for the values of *bgData.w* and *bgData.h*. This simply doubles the size of the background horizontally and vertically, so now we will have 4 screens worth of BG.

Finally, in the main section I have included an FntPrint statement to tell you the current scroll values of the background.

```
FntPrint("scroll: X: %d Y: %d\n", bgData.scrollx, bgData.scrolly);
```

This goes in right after the frame counter line, *FntPrint("frame: %d\n", frameNo);*.

That’s it set up now for scrolling, the program for this section is in part3/part3-3. Its not hard to modify the scrolling to operate however you want it, for instance you could have a single screen scroll set-up which makes the screen scroll to the next background when the player reaches the edge of the screen (Zelda-esque).

11.0. Custom Backgrounds

It may seem a bit pointless to create custom background handlers at first but when you use the ones provided you may notice that they are fairly limited in what they allow you to do. For example, you have to code your game with a fixed 16x16 format, instead of an arbitrary size that you may prefer, such as 32x32. It's also impossible (as far as we are concerned) to implement isometric or hexagonal tiles.

Custom background handlers give you the option of what tile size you would like as well as opening up other possibilities such as scaling and rotation. With the implementation of your own background handler, the *GsBG*, *GsMAP* and *GsCELL* structures will be replaced by a much simpler system involving a single *GsSPRITE* structure and an array of shorts (16 bits) containing u, v offsets.

The idea is that an image with the tiles in it is loaded into VRAM, we set a *GsSPRITE* structure to the sprite area on VRAM. We also include tile index data, which holds the index of the tile that should be placed at the current position on the screen. A set-up function is employed to convert these indices into usable u, v image offsets and store them in an array for use while rendering. As the map is rendered we step through the u, v offsets and set the *GsSPRITE* structure u and v members to the relevant members of the offset array then call *GsSortSprite()*. You may notice that this code is somewhat similar to the sprite animation algorithm as mentioned earlier.

A point worth mentioning is that if your BG handler does not include rotation and scaling then *GsSortFastSprite()* should be used to speed things up.

The tile image that we will use for the program is shown below. As well as loading this, we need data which specifies the tiles that appear at certain positions as they will appear on the screen; this is the tile index data that I mentioned earlier. This data is generated by a 2D map creation program, such as Mappy (www.tilemap.co.uk) which I use, although there are many others. The result is typically a 2D array of values, which we will paste into a header file and use to set up the tile offset array.

11.1. Map Initialisation

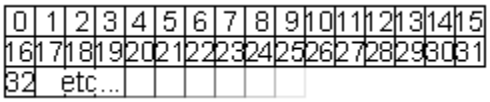
To calculate the u, v offsets we use the tile index array. The code is shown below.

```
void InitMap(GsIMAGE *bgImg)
{
	int x;
	int y;

	// get the image to be used as the background tiles
	InitSprite(&bgSprite, bgImg);

	// the width and height of the tiles
	bgSprite.w = TILEDXY;
	bgSprite.h = TILEDXY;

	// fill all the tile structures with the U and Vs
	for(x = 0; x < 20; x++)
	{
		for(y = 0; y < 16; y++)
```

```
                {
                        // obtain the u and v offset from the linear interpretation
                        // of the tiles in the image from the background 2d array
                        tiles[x][y].u = ((maparray[y][x] - 1) % IMG_WIDTH_TILES) * TILEDXY;
                        tiles[x][y].v = ((maparray[y][x] - 1) / IMG_WIDTH_TILES) * TILEDXY;
                }
        }
}
```

First of all we pass in the *GsIMAGE* structure which will contain the information about the image that contains all of the tile images. There are two loop variables, x and y, declared. The sprite next has to be initialised so we pass the *GsIMAGE* structure to *InitSprite()* as well as the *GsSPRITE* structure. We then set the width and height of the image to the width and height of one tile; this is because when we draw the sprite we only want to draw one tile from the sprite, so defining an offset, u and v coordinate with the width and height will define a rectangle around the tile on the image that we want to draw.

We then loop through the horizontal and vertical of the map array while filling in the u, v offsets. The first line of code here sets the u coordinate. The modulus acts like a carriage return, to see how such an analogy applies we will consider an example. The following diagram shows how the tiles are laid out on the image with their number indices.



If the tile index I had been given was 11 and the image was 16 tiles wide then the modulus operation would return 11 because it does not exceed a multiple of 16. 11 times the width of the tile gives us the u coordinate. If however, the tile index we had been given was 21 the modulus operation would give 5, multiplying this by the tile width gives the u coordinate.

Calculation of the v coordinate is almost identical except a divide operator replaces the modulus. Here's the explanation; if the tile index was 11 and the image was 16 tiles wide then the operation would return 0, because 11 goes into 16 0 times (in whole number terms), which is what we want, if however, we are given 21 then the operation would give 1, this multiplied by the tile height gives the v coordinate. The different between the divide and modulus operator is that the modulus operator returns the remainder of the equation.

Note that in the equation we subtract one from the tile index, this is because we require 0 indexed values and Mappy gives values indexed from 1.

11.2. Map Rendering

Rendering the map is a simple affair as shown in the code below.

```
void DrawMap(GsOT *pOT)
{
        // holds the current loop through the two dimensional map array
```

```
        int x, y;

        // step through the tiles array and draw them
        for(x = 0; x < MAPDX; x++)
        {
                for(y = 0; y < MAPDY; y++)
                {
                        // set the u, v offsets into the image
                        bgSprite.u = tiles[x][y].u;
                        bgSprite.v = tiles[x][y].v;

                        // set the x, y position on the screen left
                        // shift 4 is the same as multiplying by 16
                        bgSprite.x = x << 4;
                        bgSprite.y = y << 4;

                        // register the background tile into the ordering table
                        GsSortSprite(&bgSprite, pOT, BG_OT_PRI);
                }
        }
}
```

First we pass in the pointer to the current ordering table, so that the tiles can be registered from drawing on the screen.

In the for loops all we need to do is step through the offsets array, assigning the u, v values to the corresponding members of the *GsSPRITE* used for the background image. We then set the x, y coordinates of the *GsSPRITE* using a simple bit-shift operation, which is the same as multiplying by 16, so that it is drawn in the correct position. Finally we call *GsSortSprite()* to render the tile. The BG_OT_PRI is a macro that specifies a value that is used for the ordering table z-sorting, this is a new system that has been employed to make it easier to manage priorities.

11.3. Scrolling a Custom Background

12.0. Collision Detection

A lot of people seem to have trouble with collision detection, the difficulty is justified, but the basics are relatively straightforward. I admit I had difficulty understanding the concepts as well. What I find though is that people think ‘how do I do collision detection in my game?’ when they should be thinking ‘how do I find out if there is a collision between two sprites’, this may sound fickle but if you just concentrate on how to determine if two sprites are touching then the rest is easy. All you need to do this is already supplied in the *GsSPRITE* structure; the *x* and the *y* coordinate and the width and height. However, collision detection and response can get ugly when you are dealing with sprites that can change size and rotation.

I am assuming that you will just want to disallow sprites from entering other sprites space; this is normally the case with collision detection.

X.0. Appendices

X.1. Converting PAL Yaroze games to NTSC

Many people cannot run some of the games released by other members because they may be in a different region setting. It is simple to change provided you have the original source files.

The two main parts of the game you have to change are the graphics (.tim files) and the initialisation code for the Yaroze graphics display.

X.1.1. Changing TIM Files

If the game contains a project file for Tim Tool 3 then your laughing, just load it up and change the Frame Buffer Options -> Set Graphics Mode option to a resolution in the NTSC section of the sub-menu. Make sure that the option you choose is one near the same option of the resolution you changed from or else your images will be dwarfed (or oversized) depending on the resolution. Some artwork can be accustomed to the size of the screen and you will never have the game looking exactly as it would in PAL. Remember that the vertical low resolution for PAL is 256 and in NTSC its 240 and in high-res its PAL:512, NTSC: 480, 16 and 32 pixels respectively is quite a difference.

X.1.2. Changing the Code

This can range from extremely easy to near impossible. It all depends on how much of the code depends on the resolution of the display. But the way you would go about changing the examples of this tutorial is to root out the *Initialise()* function. Some programs may not have a *SetGraphicsMode()* call so the idea is to put one in the initialise function. If there is one, then you simply change the *MODE_PAL* to *MODE_NTSC*. You also have to change the resolution that is used in the code. This should be the same resolution you set the TIM images to, e.g. 320 x 240. I have included an example in the tutorial data directory; *2DTutoData\GraphMode*.

X.2. Glossary

I found that after a while I was building up quite a lot of techno babble so I decided to put this section in so that anyone not knowing what certain words are, they can simply look here.

Yaroze

The name of our beloved little black box of tricks, Japanese for Lets Create!

V-RAM, VRAM, Video RAM

This stuff is where we keep the images we are going to use in the code. If you load up TimTool 3 you will see the main window… this is a mapping of the Yaroze's VRAM its where you arrange your images before they are loaded to the Yaroze for use.

Video Mode

The world is divided up into several territories called regions. We are in the second region of the world where we have PAL displays. Countries like Japan and America are in region one where the video mode is NTSC. PAL and NTSC are video modes. You will find that PAL images are crisper but have a lower frequency (at 50 frames per second) and NTSC are faster (at 60 frames per second) but tend to be more blurred. Some PAL TVs can accept NTSC signals and some NTSC TVs can take PAL signals, it all depends on the feature set of your TV.

BG, Background, bg

This is the general term for the background of a two dimensional game. It gets thrown around a lot and can mean a lot of things; in Yaroze talk it just basically means the background of the game or the actual BG structure *GsBG*.

Image, Sprite

These are both names for a representation of something on the screen that moves and is usually controlled by the player. In games images are usually referred to as 'sprites' but in computer art they are referred to as 'images' they both mean pretty much the same thing. On the Yaroze, there are two structures that can be easily confused that relate to this; *GsSPRITE* and *GsIMAGE*. You can view the *GsSPRITE* structure as the 'header' of the data and the *GsIMAGE* structure as the data itself.

X.4 Is this where we part Company?

This maybe unfortunate for you, but nope, this is not where we part company; I have tonnes more I want to add to this tutorial. As the version history continues to grow I will be adding stuff like sprite animation, a bit of sound (maybe), artificial intelligence (should be a laugh), some advanced sprite techniques (i.e. transparency, scaling and rotation) and by the end of it all I hope to have made a complete game similar to Mercs on the Sega Mega Drive and Master System – a top down shooty-bang-bang game. That is of course if you choose to accommodate me ☺

For now though, over and out!

11.5. Special Thanks!

- Marc Lambert, for helping me tie up loose ends where explanations are concerned.
- Derek Da Silva, for explaining the GsIMAGE.pmode values.
- Matt Verran, for the superb font module.
- Greg Cook for supporting me, keeping me going and for the very helpful suggestions.
- Everyone I have referenced in some way throughout the tutorial for helping me understand more and in turn helping you understand more (hopefully ☺).
- Sony, for making the coolest little piece of amateur dev kit I have ever seen.