About the Bounds Green Campus

The Bounds Green Campus is located on Bounds Green Road just before the intersection with the North Circular Road. The building used to be a factory and has been turned into a sort of inverted Pompidiou Centre, as all the ventilation and girders etc are exposed and painted in primary colours. Bounds Green is basically a desert in the summer when there are no students around, and this is reflected in the poor catering provision at this time. The canteen is open until about 3pm but the food tends to go early on. There are a couple of drinks and snacks machines in the building and there is a pub and a couple of corner shops nearby. You may consider bringing lunch/refreshments with you!

Using Equipment at Middlesex

The Yaroze lab contains 15 PC’s connected to PSX’s and 15 machines that run 3Dstudio MAX. All machines are dual boot, they can boot up into Windows 3.1 or Windows NT. For PSX program development you need to use Windows 3.1, 3Dstudio and some PSX utilities (TIMtool and RSDtool) have to run under NT.

Booting into Windows 3.1 for PSX development.

Turn on the computer and the machine will boot up over the network. After some time a login screen will appear asking you for your novell account number, just press returns and after some time another menu will appear. Select Windows 3.1 from this menu and the machine will boot into Windows.

Finding the PlayStation Icon.

Click on the icon called 'find new icons'. When the process is finished you should find a link to PlayStation in the programming languages folder. When you click on this icon you get a list of four choices. Click on the car demo to check to setup is OK. All being well it should download the demo to the PSX and run. If it doesn't work then reboot the PSX and try again. If it still doesn't work get help.

For a typical session, when you click on the PlayStation icon, choose the first option and you will start a dos shell with the correct environment settings. NOTE environment settings will be wrong if you go to DOS by clicking on the DOS icon instead of the PlayStation icon. You should then work in the H:\work\data directory, or make yourself a temporary directory on the C: drive -but purge it when you finish.

Editing C programs

It is recommended that you use the Borland C++ IDE for editing you programs as it is a Windows based editor that is easy to use and highlights syntax making your programs easy to read. You will find it in the Programming Languages folder in Windows.

Booting NT

When you first turn on a machine it comes up with the message “Type h to boot from hard disk” and if you type ‘h’ it will boot up into NT from the hard disk, if you do nothing it will boot up over the network meaning that only Windows 3.1 will be available. When NT is booted up you will come to a logon screen. Type ‘guest’ as the user name and leave the password field blank to log on to NT. 3Dstudio MAX can be found under the Kinetix menu from the Start menu. TIMtool and RSDtool can be found on the desktop.

Programs and utilities

The Yaroze comes with a set of dos and windows tools to aid game development that can be run from the command line if you have gone to DOS from windows 3.1 via the PlayStation icon. These tools are introduced where appropriate in the following notes. TIMtool and RSDtool will only run under NT.

Also available under Windows 3.1 are the Corel Draw suite of tools that are useful for manipulating sprites and textures. The tools can be found in the ? folder.

Printing

If you want to get a program printout unfortunately you have to go to the computer reception (which is as far away as it could be, on the second level above reception at the main entrance) to pick it up.

Disks

The PC's connected to the Yarozes have Zip drives but the 3Dstudio machines unfortunately don't yet have them. It is recommended that you bring a zip drive with you to store your work on, although all machines have standard 31/2 inch floppy drives.

Video

If you don't have or are not intending to get a Yaroze machine then to document what you have achieved during the week you can get screen shots of your work. However if you supply a video you can video your game in action…

Introduction

Acknowledgements

Thanks to Paul Holman and Sarah Bennett at Sony Computer Entertainment Europe (SCEE) for supporting us with equipment, and Lewis Evans (of SCEE) for sorting out a couple of problems with the original 3D Tutorial. Thanks also to Ian Frost and Sean Butler for initial help and especially the Yaroze Community for their enthusiasm and creativity.

Aim

The aim of this tutorial is to get you quickly up and running using the PSX and getting results for 2D and 3D games. The PSX library seems obscure compared to say OpenGL, because it is closely tied to the hardware architecture. Consequently we have shielded you from much of the complexity by providing high level functions that sit on top of basic Yaroze functions. In some cases you may never need to look at or understand the details of these functions, in other case as you progress you may wish to understand and modify them. We are not expert PSX or games programmers and are likely to program in a less that efficient manner, however the aim is to produce understandable and useable code. This document is trying to be both a tutorial and reference work so that you can take a fast track approach and get results quickly or you can take a thorough approach and get to the point where you can rewrite our functions optimally for your own games and designs. Most sections in the document have a FASTRACK and SLOWTRACK section to

Net Yaroze – Brief Architecture and overview of capabilities

These notes are a selective summary of relevant information from the User Guide.

The CPU and Peripherals
R3000 CPU: This is the main processing unit.
Main memory the PSX has 2M of memory, and as far as we are concerned it goes from hexadecimal address `80090000` - `801fff00`.
OS rom Device boot ROM, not accessible to programmer.

PIO & SIO PIO - Parallel expansion port - reserved for future use. SIO Serial expansion port, used to connect to PC or to connect two machines back to back or to PC for programming.

The Graphics System
GPU -Graphics Processing Unit Like a graphics card on a PC - the GPU can draw to the screen and has high speed polygon drawing functions that draw on the frame buffer (3D acceleration). The GPU has a 1M frame buffer organised as 1024 x 512, which is managed by the GPU (you can't access it directly). You use the frame buffer to store your 2 buffers for double buffering to the screen and you also use it to store your images for sprites and textures - How much space you have for textures will depend on the PAL mode you are in (ie horizontal and vertical resolution) and the colour depth you use. See the manual for valid PAL modes, we will default to 320 x 240 non-interlaced mode in the tutorial. Colour depth may be 15 bit (32K colour) or 24bit (16M colours) we will default to 25 bit colour resolution .

Drawing capabilities. The GPU supports 3 or 4 sided polygon and straight line drawing, image mapping (ie texturing polygons and background images) to screen, gouraud and flat shading, and a number of other features like transparency, colour dithering and clipping).

Colour Lookup Tables (CLUTs) PSX supports 4-bit (16 colour) 8-bit (256 colour) and 15 bit direct mode (32K colours) CLUTs to be associated with textures. Each texture may have it's own CLUT which may be manipulated individually.

The Sound Processing Unit - SPU. We will only use sound for simple sound effects but the SPU can stream audio from the CD ROM and actually has 24 voices and 512K of its own memory to support music during gameplay (i.e. its possible to convert midi tracks to play on the PSX).

Other Systems.
Geometry Transfer Engine - GTE. The GTE is a kind of maths co-processor that deals with the 3D vector and matrix calculation necessary for 3D but uses fixed point arithmetic and works with the GPU.
Memory cards. Used to save game states.

Programming the Net Yaroze
Preparing 2D and 3D graphical data and sound data

2D image data used for implementing sprites, texture mapping and backgrounds. Images must be in Sony's TIM format , conversion utilities are provided to convert from BMP JPG TIF or PCX.

3D models must be converted to Sony's TMD format. Conversion utilities are provided to convert from DXF to an intermediate format RSD, and from RSD to TMD.

Facilities exist to convert sound samples (e.g.: .wav files) and sound sequences (e.g.: midi files) to formats supported by the PSX. We will only consider converting wav files to produce simple sound effects.

The development process

The basic coding process involves using an editor on the PC (ideally a nice windows one like BC++4.5) to edit your code, using the gcc compiler with a make file to compile your code, and using a batch file and the **siocons** communications software to download and execute your program on the PlayStation (PSX).

Makefiles and the gcc compiler

The compiler you will use for PSX development is the gcc shareware compiler, which runs under dos. (gcc stands for GNU C Compiler, and GNU is a shareware version of Unix thus GNU stands for GNU Not Unix). Before you can use the compiler you have to be in DOS and have certain environment variables set. Hopefully this will occur automatically if not you can copy a batch file off the file server to set the environment accordingly.

To use this compiler you have to use the 'make' function and 'makefiles' which you may or may not have come across before. Makefiles allow you control how your code is compiled and ensure that only changed files are recompiled. For further information on makefiles check the documentation, otherwise the minimal information you need to know is presented below. To create a makefile copy the example makefile of the server and edit it to make sure it looks like this:

```
CFLAGS       = -Wall
LINKER       = -Xlinker -Ttext -Xlinker 80140000

PROG         =     main
OBJS         =    main.o pad.o

all: $(PROG)

main    : $(OBJS)
      $(CC) $(LINKER) $(OBJS) -o $@
      strip $@

.c.o:
      $(CC) $(CFLAGS) -funsigned-char -c $<

clean:
      del $(PROG)
      del *.o
```

The important details in the makefile from your point of view are ‘`PROG = main’` which specifies the name of your program and ‘`OBJS = main.o pad.o’` which specifies the names of the intermediate object files which must be built for each of which there should be a corresponding source files in the current directory (eg: main.c and pad.c. Note that pad.c is a program to help you read which buttons have been pressed on the joypad, you can copy it and its associated header file pad.h from the file server). Once you have the c source files and the makefile you can compile your program type simply typing ‘make’. If you want to force recompilation of everything type ‘make clean’ and then type ‘make’. If your compilation was successful then the last line generated by the compiler should be:

```
strip main
```

If your code had an error in it, the last line will indicate it eg:

```
make.exe: *** [main.o] Error 1
```

and further up will be indicated the nature of the error and the line that caused it.

STEP 1. Hello world on the PC

In this step you will include the Yaroze development libraries, and write a ‘hello world’ type program that prints a message out to the PC screen when the program is run. Type in the following code and save it as ‘main.c’.

```
/****************************************************************
    main.c
****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <libps.h>
#include <string.h>
#include "pad.h"

void main()
{
printf("Program running, will now exit and reset Yaroze.\n\n\n");
exit(0);
}
```

Compile your program using ‘make’ as described in the ‘Makefiles and the gcc compiler’ section above. Errors and warnings generated by the compiler are written to the screen. (Note we didn’t really need to include pad.h as we don’t use the joypad yet).

Now create a batch file that specifies what is to be loaded down to the PSX. Type in the following and save it in you current directory as ‘auto’. (Note by convention no file extension is used but you can use one if you wanted to).

```
local load main
go
```

The SIOCONS communications software

To download your program to the PSX you run a program called **siocons** at the DOS prompt. If siocons manages to talk to the PSX properly you will get a prompt that looks like this: >>. If the prompt doesn't appear after pressing return a few times then reset the PSX, press esc and try again, if you still have problems get help.

```
D:\mygame\>siocons
siocons -- PlayStation debug system console program
   for DTLH3000 1996/05/10 00:00:03
   type  F1   ----> display help
   when hung up try type  ESC
 I/O addr = 0x02F8, IRQ=3(vect=0x000B,8259=20)
 BAUDRATE = 115200

Connected CIP ver2.0
Communication baud rate 115200 bps
OK
>>
```

Once siocons is running press F3 and type in 'auto' and press return. A load of information comes up on the PC screen as shown below:

```
>>
Auto[1]: auto
main [ .text] address:80140000-801402ef size:0002f0  0002f0:    0sec.
main [.rdata] address:801402f0-8014032f size:000040  000040:    1sec.
main [ .data] address:80140330-8014047f size:000150  000150:    1sec.
main [.sdata] address:80140480-801404ff size:000080  000080:    2sec.
  PC=80140130, GP=80148480, SP=801ffff0

>>go
Program running, will now exit and reset Yaroze.


ResetGraph:jtb=8004829c,env=800482e4
ResetGraph:jtb=8004829c,env=800482e4
PS-X Control PAD Driver  Ver 3.0
addr=8004d724

Connected CIP ver2.0
Communication baud rate 115200 bps
OK
>>
```

Simultaneously the PSX displays similar messages until the word 'go' appears which causes the PSX to execute the program. The PSX screen goes blank, the text '`Program running, will now exit and reset Yaroze.`' appears on the PC screen, the PSX resets itself, and the PC displays the siocons >> prompt again. Well done you've just created your first PSX program, and what a game it was...

STEP 2. Hello world on the PSX and double buffering

FASTTRACK

This program simply writes a line of text to the PSX screen and exits the program when the `'select'` button on the joypad is pressed. Most of the tricky stuff is hidden in the step2.c and step2.h files for the moment all you really need to understand is the main.c file. The task involves:

1. Calling the PSX `FntLoad` and `FntOpen` functions to prepare for writing text to the PSX screen.
2. Initialising the system to do double buffering etc by calling function `Initialise3Dgraphics();`
3. Initialising the joypad with `PadInit()` and reading it with `ReadPad()`
4. Defining a function `RenderWorld` which calls two functions: `RenderPrepare()` – to initialise rendering, and `RenderFinish()` to complete rendering.
5. Using the PSX functions `FntPrint()`and `FntFlush()`to write your text in the body of the `RenderWorld()`function.

This is the code for the main.c

```
/**************************************************************
       main.c
       ======
**************************************************************/

#include <stdio.h>
#include <libps.h>
#include "pad.h"
#include "step2.h"

// We need a variable to store the status of the joypad
u_long PADstatus=0;


// This function deals with double buffering (ordering tables
// etc) and writes text to the PSX screen.
void RenderWorld()
{     // get ready for rendering
      RenderPrepare();
      // Print your elegant message
      FntPrint("Hello World!\n");
      // force text output to the PSX screen
      FntFlush(-1);
      //finish off rendering
      RenderFinish();
}

int main()
{
      // set up print-to-screen font, the parameters are where
      // the font is loaded into the frame buffer
      FntLoad(960, 256);
      //specify where to write on the PSX screen
```

```
	FntOpen(-96, -96, 192, 192, 0, 512);
	// initialise the joypad
	PadInit();
	// initialise graphics
	Initialise3DGraphics();
	while(TRUE){
		// read the joypad
		PADstatus=ReadPad();
		// if the select button has been pressed then break
// out of while loop
		if (PADstatus & PADselect) break;
		//other wise do double buffering
		RenderWorld();
		}
	// clean up
	ResetGraph(3);
	return 0;
}
```

SLOWTRACK

In this step you get to write to the PSX screen but you have to do a lot more work and get exposed to the intricacies of PSX programming. This program simply writes a line of text to the PSX screen and exits the program when the ‘select’ button on the joypad is pressed. In order to do this we have to get into graphics mode and set up double buffering. The ability to print on the PSX screen is very useful for debugging as messages sent for printing to the PC screen can become garbled due to synchronisation problems between the PSX and PC.

An aside on PSX datastructures

We are about to start to learn about the PSX data structures necessary for getting on so a comment about them is timely. Looking at the manuals the structures often appear curious and incomprehensible - and looking at example programs some variables in structures never seem to get used by the programmer at all. Consequently the best initial approach is to develop an understanding of these datatypes on a ‘need to know’ basis eg: to initialise an ordering table header (see function `Initialise3DGraphics` below) we have to set it’s length and *org variables, (if fact `Initialise3DGraphics` does this for us), we can ignore the rest.

Ordering Tables and Ordering Table Headers

The PSX implements a version of the painters algorithm for hidden surface removal, where all polygons are sorted in order of depth before being drawn to the screen. Using such as system the most distant polygons are drawn first and nearer ones last so that the hidden surface removal is easily achieved. A draw back to this approach is that problems occur when two polygons overlap in depth. The usual solution to this problem is to subdivide the polygons that overlap as seen from the viewpoint, however the PSX doesn’t implement this so you occasionally see polygons overlapping incorrectly during game play. The PSX does support object (as opposed to view based) polygon subdivision which can minimise overlap but doesn’t cure it generally and is expensive. We have to specify the precision, which is used for depth sorting which can be from 2^1 to 2^{14} levels of depth. If the precision is too low we risk having every thing randomly drawn onto the same plane, whilst higher precision will gobble more memory.

We define the precision as a constant in the program so that if we want to change it we only have to do it in one place, eg:

```
#define ORDERING_TABLE_LENGTH (12)
```

The structure into which polygons are sorted is referred to as an ordering table and the ordering table is actually accessed via a structure called an ordering table header. We need two ordering tables, one for each buffer. When we implemented 2D double buffering we only had to consider the video memory needed to store each buffer. With the PSX we have to allocate the full set of datatypes associated with ordering tables for each buffer. Thus a 'buffer' now involves more memory that just the video buffer. Associated with an ordering table is an ordering table header, which we reference in dealing with ordering tables, and an array of packets (discussed below). We will need two of each of these three structures, one for each buffer. For convenience we use arrays, so the two ordering table headers are defined as an array of `GsOT,` the structure for ordering table headers:

```
GsOT        OTable_Header[2];
```

and the ordering tables are defined as an array of `GsOT_TAG,` the structure for ordering tables:

```
GsOT_TAG    OTable_Array[2][1<<ORDERING_TABLE_LENGTH];
```

Note that the ordering tables are defined by doubly index arrays, the first index references the buffer (0 or 1) whilst the second indexs an array of `GsOT_TAGs` which actually comprise the ordering table. Also note that the number of tags in the array is denoted by `[1<<ORDERING_TABLE_LENGTH]` ie; the number 1 bit shifted to the left by 12 places which is 2^{12} which equals 4096 levels of depth. Having defined the ordering tables and associated them with their headers (see `Initialise3DGraphics` below) we reference them by their headers and can otherwise forget about them.

Packets

A PACKET is the smallest unit of drawing commands that can be dealt with by the Graphics Processing Unit (GPU). As part of the rendering process the GPU generates these packets so that drawing to video memory may be achieved. This only need concern you in as much as its up to the programmer to specify and allocate this area. This temporary drawing area is defined by an array of PACKET structures, so again we need to create an array of these, one for each buffer. Again we specify a doubly indexed array, the first index addressing either of the two buffers, whilst the second index addresses the PACKETS actually to be used.

The tricky thing is that we have to decide on how many PACKETS will be needed and this number will vary from program to program and over time within a program. The number we have used here is way and above what we really need so far, but will stand us in good stead as our scenes become more complex. Our definition is as follows:

```
#define MAX_NO_PACKETS  (248000)
```

```
PACKET          Packet_Memory[2][MAX_NO_PACKETS];
```

You can optimise this later if you know the number of primitives (where a single polygon, line, sprite, etc is one primitive) in your world as follows:

```
PACKET Packet_Memory [2][MAX_NUMBER_PRIMITIVES * SIZEOF_PRIMITIVE]
```

where the average size that a primitive can be is 24 (ie: set `SIZEOF_PRIMITIVE` to 24). Note that allocating insufficient packet space is a very common cause of programs crashing.

Reading the Joypad

The pad.h include file defines the bit patterns that correspond with the pressing of the various joypad buttons whilst the pad.c file contains a function to initialise the joypad and another to read it. In the program we need a variable to store the status of the pad when we read it, thus we have defined:

```
u_long  PADstatus=0;
```

In the main loop we can then set the `PADstatus` variable by reading (using the `ReadPad()` function) it and test if the select button has be pressed in which case the following evaluates to true:

```
(PADstatus & PADselect)
```

Most of the PSX specific stuff is hidden in the step2.c and step2.h files. You can get by for now with only a cursory understanding of most of these PSX functions. For further information check out the manuals. For reference here is the program as it was originally written in one file but excludes the main function which already appears as the start of this section.

```
/**************************************************************
       main.c
       ======
*************************************************************/

#include <stdio.h>
#include <libps.h>
#include "pad.h"

#define ORDERING_TABLE_LENGTH (12)
#define MAX_NO_PACKETS  (248000)

#define SCREEN_WIDTH  (320)
#define SCREEN_HEIGHT (240)


// We need two Ordering Table Headers, one for each buffer
GsOT OTable_Header[2];
// And we need Two Ordering Tables, one for each buffer
GsOT_TAG        OTable_Array[2][1<<ORDERING_TABLE_LENGTH];
```

```
//We also allocate memory used for depth sorting, a block for
// each buffer
PACKET          Packet_Memory[2][MAX_NO_PACKETS];
// We need a viewing system for 3D graphics
u_long PADstatus=0;

// this function initialises the graphics system
void Initialise3DGraphics()
{
//Initialise The Graphics System to PAL as opposed to NTSC
      SetVideoMode(MODE_PAL);
      // Set the Actual Size of the Video memory
      GsInitGraph(SCREEN_WIDTH, SCREEN_HEIGHT,
          GsNONINTER|GsOFSGPU, 1, 0);
      // Set the Top Left Coordinates Of The Two Buffers in
      //video memory
      GsDefDispBuff(0, 0, 0, SCREEN_HEIGHT);
      // Initialise the 3D Graphics...
      GsInit3D();
      // Before we can use the ordering table headers,
      // we need to...
      // 1. Set them to the right length
      OTable_Header[0].length = ORDERING_TABLE_LENGTH;
      OTable_Header[1].length = ORDERING_TABLE_LENGTH;
      // 2. Associate them with an actual ordering table
      OTable_Header[0].org = OTable_Array[0];
      OTable_Header[1].org = OTable_Array[1];
      // 3. initialise the World Ordering Table Headers and Arrays
      GsClearOt(0,0,&OTable_Header[0]);
      GsClearOt(0,0,&OTable_Header[1]);
}

// This function deals with double buffering (ordering tables // etc)and
writes text to the PSX screen.
void RenderWorld()
{
// This variable keeps track of the current buffer for double
//buffering
int currentBuffer;
      //get the current buffer
      currentBuffer=GsGetActiveBuff();
      // set address for GPU scratchpad area
      GsSetWorkBase((PACKET*)Packet_Memory[currentBuffer]);
      // clear the ordering table
      GsClearOt(0, 0, &OTable_Header[currentBuffer]);
      //print your elegant message
      FntPrint("Hello World!\n");
      // force text output to the PSX screen
      FntFlush(-1);
      // wait for end of drawing
      DrawSync(0);
      // wait for V_BLANK interrupt
      VSync(0);
      // swap double buffers
      GsSwapDispBuff();
      // register clear-command: clear to black
      GsSortClear(0, 0, 0,&OTable_Header[currentBuffer]);
      // register request to draw ordering table
      GsDrawOt(&OTable_Header[currentBuffer]);
}
```

By now you should be able to do some daft things like: change the text message on screen, change its location, change the screen width and height (but make sure it's a valid mode by checking the manual), and see the effect of having a `Packet_Memory` array length of 0 - have a go...

Before going deeper in to discussion of 2D graphics we need to introduce two tools which are going to be very useful to you.

The MEMTOOL utility for memory addresses

The PSX allows you to load 3D models (.TMD files), 2D images (.TIM files) and sound effects files (.VH and .VB files). Unfortunately you have to specify in hexadecimal where in memory these files need to go. You have to specify these addresses twice, in the auto file that loads the data to memory and in your program so that you can use the data. This is a laborious and error prone process, which MEMTOOL makes easy by assigning the addresses and more.

MEMTOOL is a simple tool to manage address assignments for images (sprites and textures), 3D models and sound files. To use it all your data files should be in a directory called DATA which is an immediate subdirectory of the directory where your program and makefile are.

MEMTOOL should be called from the DATA directory and searches this directory for a file, which must be called MEMTOOL.DAT. Data files are listed one per line in MEMTOOL.DAT. The #defined symbols are generated by replacing the '.' in filenames with a '_'

MEMTOOL:

- generates an batch load file called 'auto' that automatically assigns addresses from 0x80090000 upwards and aligns addresses on long boundaries
- generates an include file 'addrs.h' that #defines these addresses so your program knows where the data is
- generates a file 'loadtims.c' that defines a function LoadTextures() which you can call from your program to automatically load textures to the frame buffer
- generates a dos batch file "rsd2tmd.bat" that will automatically convert all .rsd models to tmd models when run. (handy if you use TIMtool to arrange your textures associated with models in memory as any change to TIMs or CLUTs means you have to run rsdlink …)

Example MEMTOOL.DAT file:

```
car01.tmd
square1.tmd
com.TIM
square1.TIM
block1.tmd
```

```
sandy1.tim
tree4.tmd
tree.tim
tree5.tmd
fir.tim
bush.tmd
bush.tim
waves.vh
waves.vb
```

…will generate the file ‘auto’:

```
local dload data\CAR01.TMD 80090000
local dload data\SQUARE1.TMD 800923A0
local dload data\COM.TIM 80092410
local dload data\SQUARE1.TIM 80092850
local dload data\BLOCK1.TMD 80093090
local dload data\SANDY1.TIM 800931E8
local dload data\TREE4.TMD 80093A28
local dload data\TREE.TIM 80093B30
local dload data\TREE5.TMD 80094370
local dload data\FIR.TIM 80094478
local dload data\BUSH.TMD 80094CB8
local dload data\BUSH.TIM 80094DC0
local dload data\WAVES.VH 80095000
local dload data\WAVES.VB 80095E20
local load main
go
```

… and the file ‘addrs.h’:

```
#define CAR01_TMD 0x80090000
#define SQUARE1_TMD 0x800923A0
#define COM_TIM 0x80092410
#define SQUARE1_TIM 0x80092850
#define BLOCK1_TMD 0x80093090
#define SANDY1_TIM 0x800931E8
#define TREE4_TMD 0x80093A28
#define TREE_TIM 0x80093B30
#define TREE5_TMD 0x80094370
#define FIR_TIM 0x80094478
#define BUSH_TMD 0x80094CB8
#define BUSH_TIM 0x80094DC0
#define WAVES_VH 0x80095000
#define WAVES_VB 0x80095E20
#define FREE_MEM 0x80096540
```

void LoadTextures();
int LoadTexture(long addr);

… and the file ‘loadtims.c’

```
#include "addrs.h"

void    LoadTextures(){
        LoadTexture(COM_TIM);
        LoadTexture(SQUARE1_TIM);
        LoadTexture(SANDY1_TIM);
        LoadTexture(TREE_TIM);
        LoadTexture(FIR_TIM);
        LoadTexture(BUSH_TIM);
}
```

… and the file 'rsd2tmd.bat':

```
rsdlink -v -o CAR01.tmd CAR01.rsd
rsdlink -v -o SQUARE1.tmd SQUARE1.rsd
rsdlink -v -o BLOCK1.tmd BLOCK1.rsd
rsdlink -v -o TREE4.tmd TREE4.rsd
rsdlink -v -o TREE5.tmd TREE5.rsd
rsdlink -v -o BUSH.tmd BUSH.rsd
```

To load your textures to the frame buffer call the LoadTextures() function from some appropriate initialisation point in your program.

Ie: your include in your main file looks like:

```
#include "addrs.h"
```

and somewhere in your initialisation you load the textures by calling:

```
LoadTextures();
```

Also your makefile needs to make LOADTEXS.C (which includes the LoadTexture() function) and LOADTIMS.C described above. Eg: edit the OBJS line of your makefile to:

```
OBJS    =    main.o pad.o loadtims.o loadtexs.o
```

Note: you should always do a 'make clean' before making your program to make sure it compiles in the latest addresses.

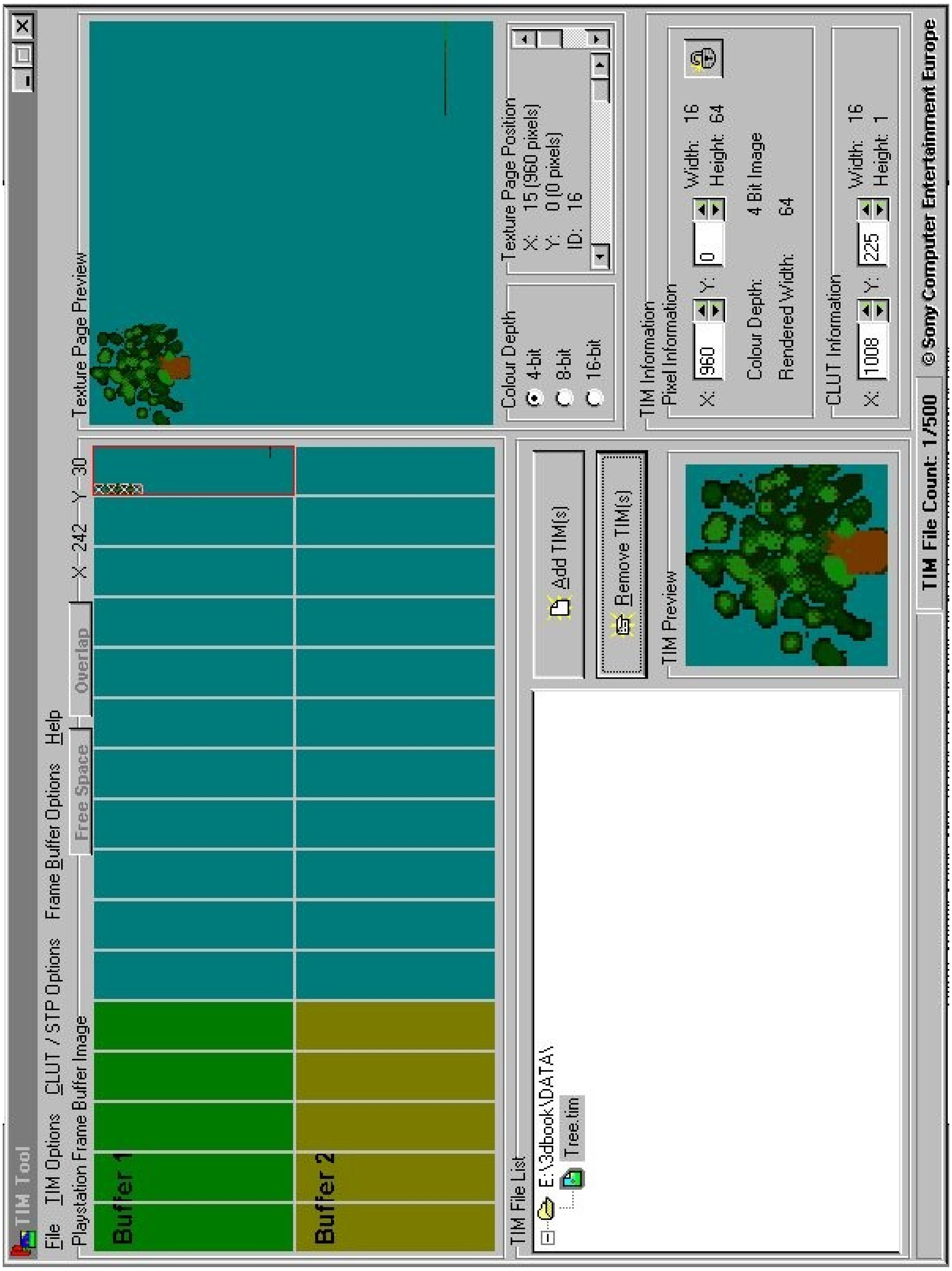
Summary of steps to use MEMTOOL

1. Have all your images, sound, and models in a subdirectory called DATA
2. Create a file in the DATA directory called 'MEMTOOL.DAT' which lists all your model. texture files, and sound files.
3. Run MEMTOOL;
4. Include `"data\addrs.h"` in your main.c program and call LoadTextures() from a suitable point. (Automatically done in the examples)
5. Edit your makefile to include the files LOADTEXS.C and LOADTIMS.C. (Automatically done in the examples)
6. If you have any textures associated with 3D models then run the batch file rsd2tmd.bat in the data directory.
7. Recompile your main program .

Thus you can easily add and load models and textures without worrying about memory address allocation at all by following these steps.

Using TIMtool to convert and position images.

Before you can use images on the PSX you must convert them from their existing format to the proprietary PSX .TIM format. You also need to organise them (and their CLUTs if they



X: 242 Y: 30

Texture Page Position
X: 15 (960 pixels)
Y: 0 (0 pixels)
ID: 16

Colour Depth
[✓ 4-bit] 8-bit 16-bit

TIM Information
Pixel Information
X: 960 Y: 0 Width: 16 Height: 64
Colour Depth: 4 Bit Image
Rendered Width: 64

CLUT Information
X: 1008 Y: 225 Width: 16 Height: 1

E:\3dbook\DATA\
Tree.tim

TIM File Count: 1/500

© Sony Computer Entertainment Europe

use them) in screen memory. TIMtool is a handy WIN95/NT utility that helps you to do this. Once you’ve arranged your TIM’s in memory you can save your work as a project (.PSX file) and also save modified TIM files.

A screen shot of TIMtool is shown on the previous page. The display is laid out into five sections. In the top left half of the screen the video memory is shown graphically. The leftmost part of this section shows the memory used to contain the two buffers used for double buffering in different shades of green. The pale blue grid that is superimposed over the video memory divides the screen up into 32 texture pages. Note that at above the screen memory are two buttons called Free and Overlap. The free button shows free video memory in dark blue when pressed, with occupied parts of memory being shown in various other colours. The overlap button shows areas of memory that are overlapping in red.

To the upper right section of this screen shows the texture page preview window. It graphically shows the images loaded into a texture page selected in main memory. A texture page is selected by clicking on it and is indicated by a red border in screen memory. In the screenshot an image of a tree has already been loaded into memory and the texture page which it is in is highlighted in screen memory, we can see the image in the preview window. Note that the image (though small) can also be seen in the screen memory, but is much longer than it is wide because of the way that pixels are stored in the screen buffer.

The lower left section of the screen shows the paths and file names of TIM files currently loaded .

The lower middle section shows a preview of the currently selected TIM file and immediately above this are two buttons for adding and removing TIM files from a project.

The lower right section gives information about the currently selected TIM, it’s width height, colour depth, etc.

***Converting an image file to TIM format*.**

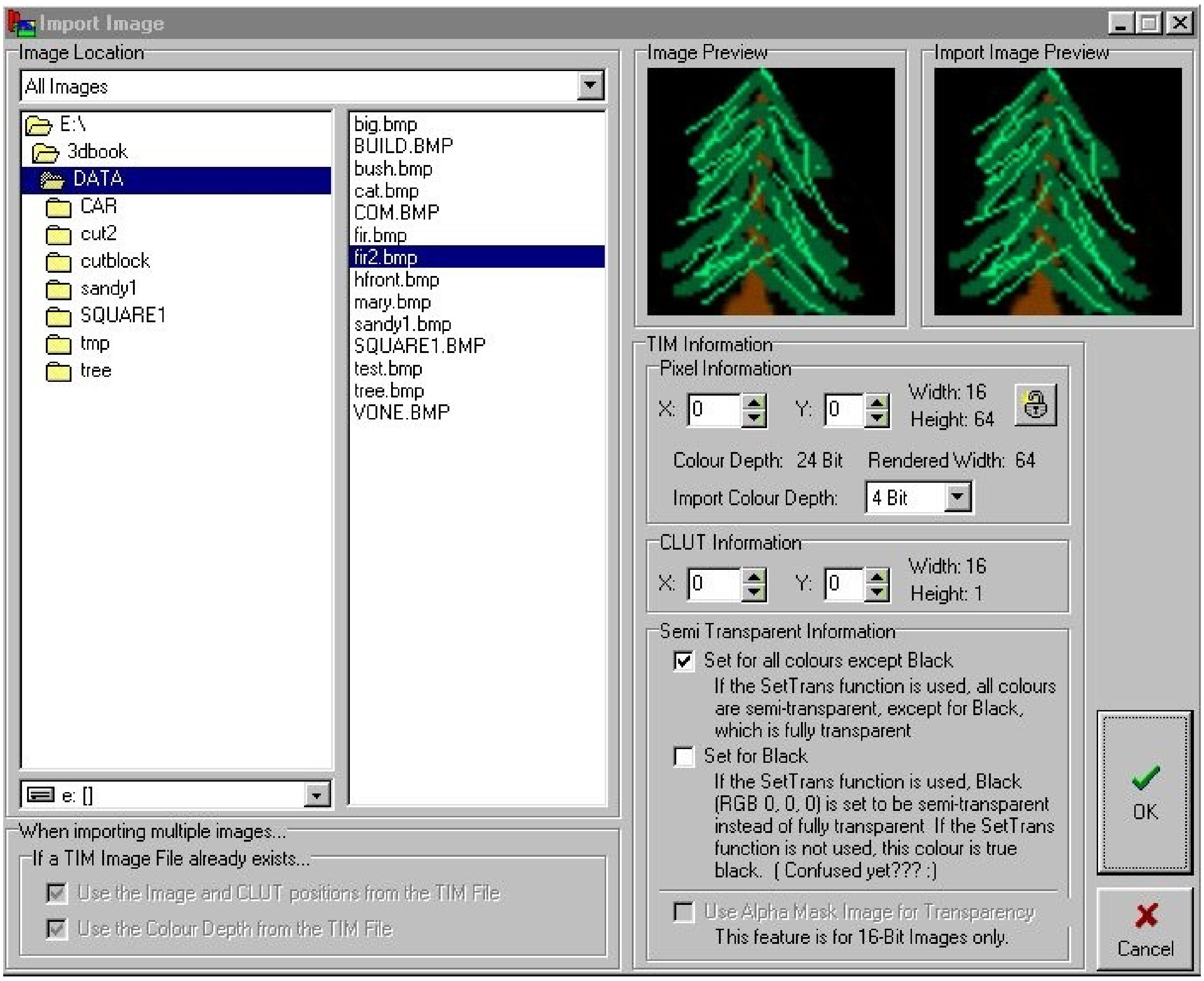

Import Image

Image Location

All Images

E:\
3dbook
DATA
CAR
cut2
cutblock
sandy1
SQUARE1
tmp
tree

e: []

big.bmp
BUILD.BMP
bush.bmp
cat.bmp
COM.BMP
fir.bmp
fir2.bmp
hfront.bmp
mary.bmp
sandy1.bmp
SQUARE1.BMP
test.bmp
tree.bmp
VONE.BMP

When importing multiple images...

If a TIM Image File already exists...

☑ Use the Image and CLUT positions from the TIM File

☑ Use the Colour Depth from the TIM File

Image Preview

Import Image Preview

TIM Information

Pixel Information

X: 0 Y: 0 Width: 16 Height: 64

Colour Depth: 24 Bit Rendered Width: 64

Import Colour Depth: 4 Bit

CLUT Information

X: 0 Y: 0 Width: 16 Height: 1

Semi Transparent Information

☑ Set for all colours except Black

If the SetTrans function is used, all colours are semi-transparent, except for Black, which is fully transparent

☐ Set for Black

If the SetTrans function is used, Black (RGB 0, 0, 0) is set to be semi-transparent instead of fully transparent. If the SetTrans function is not used, this colour is true black. (Confused yet??? :)

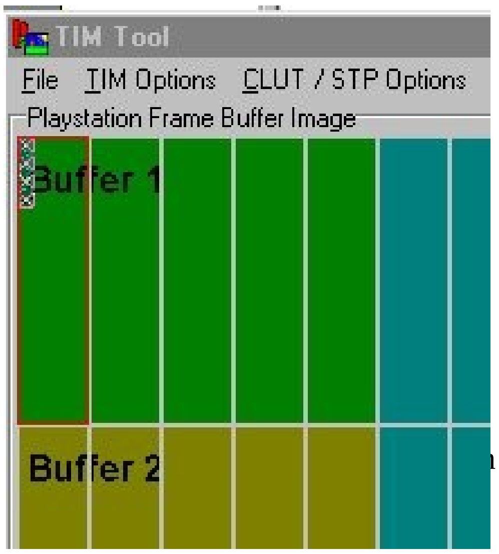
☐ Use Alpha Mask Image for Transparency

This feature is for 16-Bit Images only.

OK

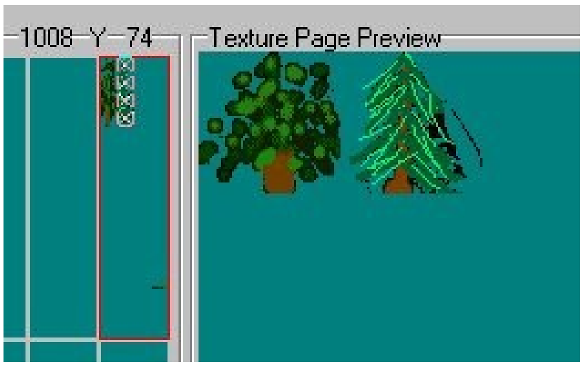
Cancel

To convert a BMP file to TIM and lay it out in screen memory and choose the FILE|Import Images File(s) menu option. A dialogue box pops up that allows you to select a .BMP, .JPG,.TIF, or .PCX file which is then previewed on the right. You can choose the colour depth (4,8, or 16 bits pixel) on this screen. The semi transparency check boxes are also important be we will discuss them later. When you click on OK your texture is shown in the video memory starting from position 0,0 (ie: in the BUFFER 1 area) overlaid by a pink rectangle with cross hatching.



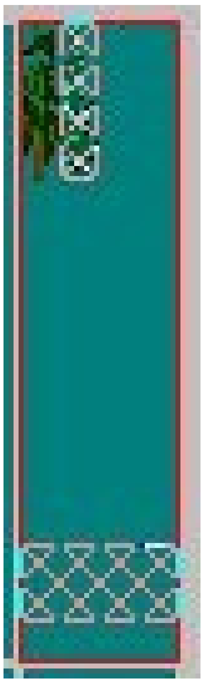
Arranging TIMs in video memory.

Your TIM is place in the area of memory reserved for BUFFER 1 which is obviously a bad idea. To move it in to a free part of frame buffer memory just select the image and drag it to a free memory. This image below shows the new tree image arranged next to the first one.



When the TIM was converted we set the transparency for black flag so what was black in the original image is shown in the background colour. The short dark mark in the near the bottom of the texture page roughly under the characters ‘74’ is the CLUT of the first tree image. The new tree image also has a CLUT which we would like to locate in this texture page.

To do this, create a bounding box towards the bottom of the texture page by holding down a shift key and clicking and dragging with the mouse. The box will be shown in pink and checked as shown below.



Now select the menu item CLUT/STP Options|Auto Sort 4-bit CLUTs into selected area and the CLUTs will be translated into this area. Note there is a similar option for 8-bit CLUTs but note that 8-bit cluts are 256 entries long and therfore span 4 textures pages.

To save your work choose menu item TIM Options|Save Modified TIMs… , which saves TIM files and the information on how they and their CLUTs are mapped to memory. Also select menu item File|Save Project As to save your project to disk. When you open a project it automatically open up all TIMs referenced in the project and display the location of the TIMs and their cluts in video memory.

To find out more about TIMtool use the online help system.

There is another Windows based tool, TIMUTIL which also converts from BMP to TIM, but it is not as versatile as TIMtool so is not documented here.

Summary of steps in using TIMtool.

1. Create an image in .BMP, .TIF, .JPG or .PCX format.
2. If you want to add the image to an existing project open that project first.
3. Run TIM tool and convert the file to .TIM format using File|Import Image File(s).
4. Position the TIM file in memory by dragging and dropping it.
5. If the image is 4-bit or 8-bit
    - select an area in memory by holding down shift and dragging
    - then select CLUT/STP Options|Auto Sort 4-bit CLUTs or CLUT/STP Options| Auto Sort 4-bit CLUTs as appropriate
6. Select Tim Options|Save modified TIMs to save the TIM files
7. Select File|Save Project or File|Save Project As… to save your project.

2D Step 3: Understanding video memory

FASTTRACK

1Mb of video memory for screen buffers, texture images and CLUTs
Video memory is 1024x512 pixels in size
Visible screen resolution is standard at 320x240 and 15bit true colour
CLUTs can be 16x1 in size (4bit 16 colour CLUT) or 256x1 (16bit 256 colour CLUT)
A texture image must fit in one texture page of size 256x256.
Texture pages start horizontally every 64 pixels in video RAM and at 0 and 256 vertically

SLOWTRACK

Use of video memory

The video RAM is one megabyte in size and exists purely for high speed image processing and it can not be manipulated in the same way that the main memory can be. Within it must be stored the frame buffers used for the screen display (either 1 or 2 depending on whether you are using interlace mode or not). Textures for 3d models and image data for 2d sprites are also stored in the video RAM, along with any colour lookup tables (CLUTs) for these textures.

Layout of video memory

The video RAM is designed for easy access from your code. It is a two-dimensional space and is 1024 pixels wide by 512 pixels high. The limits of this buffer are co-ordinates (0,0) in the top left and (1023,511) in the bottom right. Each pixel is 16 bit and as standard each is made up as follows:

> 5 bits denoting RED hue,
> 5 bits denoting GREEN hue,
> 5 bits denoting BLUE hue,
> 1 bit denoting translucency.

Inherently each pixel can be one of (2^{15}) 32768 colours (excluding semitransparency information that is only used with texture images).

The frame buffer

In the area of video RAM used for the screen buffers it is always used as stated above (ie, 5 bits for red, green and blue), and the transparency bit is ignored. How big your frame buffers are depends on what resolution you want the screen to be on the TV. With a higher resolution comes sharper pictures and textures, but the downside is that higher resolutions take longer to process, resulting in a program slowdown, and as they will use more video RAM, there is less remaining to create a wider variety of texture images. Valid screen resolutions are as follows: (all of these work for both PAL and NTSC programs)

Horizontal resolutions can be	256 pixels,
	320 pixels,
	384 pixels,
	512 pixels, and
	640 pixels.
Vertical resolutions can be	240 pixels,
	256 pixels,
	480 pixels, and
	512 pixels.

The standard size used in all the programs given to you here is very nearly the lowest resolution possible (320x240 pixels). This may seem low, but is fairly standard for commercial PSX games and has the advantages outlined above (speed and least memory used). Each of the resolutions above require two frame buffers in video RAM except for any resolution with either 480 or 512 pixels vertically. These are interlaced resolutions, offering higher resolutions and have only one frame buffer. The drawback to these resolutions is that the entire screen must be updated 50 times a second (or 60 in NTSC) without fail, otherwise huge graphical glitches occur.

Texture images and CLUTs

Texture images can be mapped onto a face of a polygon for three-dimensional effects, or used for two-dimensional sprites. Texture images can use the video ram in slightly different ways in order to get better memory usage and also to speed image processing up significantly.

Texture images can be one of three types –
16 colour images (4 bit),
256 colour images (8 bit), and
32768 colour images (16 bit).

All three of these are covered by the same graphics file format (.tim). Of these, the last uses the video ram in the same way as the frame buffer (5 bits each for red, green and blue). The other two types are slightly different. With these, each pixel in the texture image is not an absolute colour, but is instead an index to an array that specifies the absolute colour. The reasons for this is that by limiting an image to less colours, then the information required for each pixel in the image decreases, and leads to a saving in space, and hence an increase in processing speed. For example, a 16x16 pixel image with 32768 colours (16 bit) takes up 4 times as much memory as a 16x16 pixel image with 16 colours (4 bit) and is also processed nearly 4 times slower. In a games environment where speed is paramount, these become important factors to consider when designing graphics. What about this array that actually contains the colour information and is accessed when drawing the sprite? This is the CLUT (colour look-up table), and there are two sorts, one for 16 colur texture images and one for 256 colour texture images. These two sorts of texture images and their CLUTs are stored in video RAM as follows:

16 colour images (4 bit)
Each pixel in video RAM is 16 bit, and as each pixel in the image requires only 4 bits, then the video RAM can store four texture image pixels per video RAM pixel. Therefore, a texture image that is 32x32 pixels large, takes up only 8x32 pixels in video RAM. The CLUT for a 4 bit texture image is of size 16 pixels wide by 1 pixel high in the video RAM, and can be placed anywhere.

256 colour images (8 bit)
With this, each pixel in the image requires 8 bits, so the video RAM can store two texture image pixels per video RAM pixel. Therefore, a texture image that is 32x32 pixels large, takes up only 16x32 pixels in video RAM. The CLUT for an 8 bit texture image is of size 256 pixels wide by 1 pixel high in the video RAM, and can also be placed anywhere.

CLUTs are created when converting images into the file format used by the PSX (.tim format). It is at this point that you specify where they will be in video RAM. TIMtool and TIMutil are the two utilities that you can use to convert pictures into the PSX format and assign video RAM locations. Each pixel in the CLUT is an absolute colour value using the 15 bits for colour (5 each for red, green and blue) and 1 bit for transparency information (As opposed to it being used for translucency in texture images)

Texture pages and texture image considerations

The most obvious point when creating texture images and their CLUTs is to make sure that there are no overlaps in video RAM between any of the elements and each other, and also the frame buffer(s). Overlaps are one of the causes of major problems in games devlopment, and can lead from wrong colours on texture images, to them not being displayed at all and the screen being blank. Within the rule to avoid overlapping in video RAM there are also some other points to consider. The most bizarre of which is the idea of a texture page. Basically, the video RAM is split up into 32 smaller tiles for the convenience of the graphics processing. The effect on assigning video RAM locations to texture images and CLUTs are as follows:

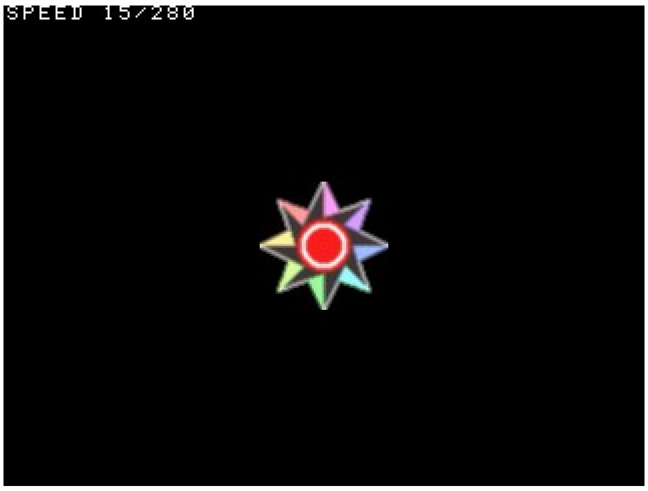
Texture images MUST be 256x256 in size or less. If you wish to use a bigger texture image, you must split it into smaller images that can be joined together in a 3d model, or by using more than one sprite in 2d.

Texture images MUST fit completely onto one texture page. Basically, there are 32 texture pages on the PSX, and their co-ordinates can be calculated as follows

Texture Page top left X co-ordinate	a multiple of 64 between 0 – 960 inclusive.
Texture Page top left Y co-ordinate	either 0 or 256.
Texture Page width and height	always 256.

Using TIMtool you have to be aware of these problems and make sure that you don’t break those rules yourself. If using TIMutil, you can visibly see where one texture page ends and starts and thus are able to more carefully judge video RAM locations.

2D STEP 4: Getting a sprite on screen



FASTTRACK

Convert sprite from windows .bmp to PSX .tim format with TIMutil or TIMtool
Use MEMTOOL to create the auto file, LoadTextures() function and create #defines for sprites
Add loadtex.o and lib2d.o to makefile OBJS

```
#include “lib2d.h”
#include “addrs.h”
GsSPRITE    mySprite;

void main()
```

```
        {
        LoadTextures();
        SetSpriteInfo(&mySprite, EXAMPLE_TIM, 160, 120);
        mySprite.x = mySprite.y = 80;
        }

void RenderWorld()
        {
        DrawSprite(&mySprite, 0);
        }
```

“lb2d.h”	include 2d graphics function
“addrs.h”	include #defines for all tim files
GsSPRITE	create instance of sprite structure called mySprite
LoadTextures	function to copy all texture images from main memory to video RAM
SetSpriteInfo	associate a texture image (EXAMPLE_TIM) to mySprite and set initial co-ordinates to (160,120)
mySprite.x	alter the x and y members of mySprite to move it around
DrawSprite	draw mySprite on screen, with depth priority of 0

SLOWTRACK

To make development time faster, all of the ordering table code has been put into functions that reside within lib2d.h and lib2d.c. This means that there are a few function calls that you have to make. At the very start of void main() you must call the Initialise2DGraphics() function, and at the very start of your RenderWorld() function call RenderPrepare(), and at the end call RenderFinish(). Loading up the template main.c file will show you how a basic program would start and all of the demo 2D programs use these functions to save time and space.

A sprite is a flat image on screen, defined only by an X and Y co-ordinate. As they have no Z co-ordinate they have no ‘depth’ to them and you must order sprites to avoid incorrect overlapping. Sprites are the basis of almost every 2d element on a screen. They can be altered, moved and updated in a variety of ways using the built in capabilities of the PSX graphics processor. In order to get a sprite on screen, the following must be done:

1. Create the graphics as windows .bmp file using any standard art package
2. Convert it into the PSX .tim format with TIMtool or TIMutil
3. Load the .tim into the PSX main memory when the program is run
4. Copy the texture image from main memory into video RAM at the start of the program
5. Create a GsSPRITE structure to hold the sprite information.
6. Specify where the GsSPRITE structure will look in video RAM for its texture image and CLUT (if necessary)
7. Tell the graphics processor to draw a copy of the sprite on screen every time the screen is updated

In this runthrough, I will assume the example texture image will be called ‘example.tim’ and will be in the ‘data’ directory.

Converting a .bmp into a .tim

Use TIMtool or TIMutil to convert your graphics as shown previously.

Loading the .tim into the PSX main memory

As shown before, to copy a file from the PC to the PSX, a batch file is used in conjunction with Siocons. The standard name for this file is auto (with no suffix). The most basic auto file (as used in the two 'hello world' examples) was as follows –

```
local load main
go
```

The auto file controls what is copied into the PSX main memory before the program is run. Now that we want to send over a graphics file, we use MEMTOOL to set up all of the graphics loading procedure. To do this we have to create the MEMTOOL.DAT file that is used to specify which files are loaded. Go into the data directory and either create the MEMTOOL.DAT file and/or edit it if it exists already until it just contains the following text:

```
example.tim
```

From DOS, run MEMTOOL.exe from within the data directory and it will create the appropriate header files, ready to be included in your program. The auto file now looks like the following, which will correctly copy the tim into main memory upon running:

```
local dload data\EXAMPLE.tim 80090000
local load main
go
```

The new line has two sections. The first is more obvious, it's the path and filename of the .tim file relative to the directory that the auto file is in. The second is an address in hexadecimal (even though it is missing the usual 0x prefix to denote that it is in hexadecimal). This refers to where in main memory (not video RAM) the file is to be loaded. Main memory on the PSX is split up into a few areas, of which you can access between memory location 0x80090000 and 0x801fff00. This translates into about 1.5 megabytes of space. Memory usage usually has to be monitored at most times, but with MEMTOOL this bookkeeping is automatic.

Copy the texture image from main memory to video memory

Remember the address that the texture image was loaded into from the previous bit? Its important now, as your code needs to know where to look for it. Unfortunately, you can't load a .tim file from the PC straight into the video memory, so the two step approach has to be taken. When you ran MEMTOOL, it created the right code to automate the creation of #defines for your program, and a function to copy them all from main memory to video RAM. To include them in your code, you need to add the following #include to the start of your program:

```
// include texture #defines and texture loading routine
#include “addrs.h”
```

With this included, you only need to use EXAMPLE_TIM whenever you want to reference that texture image when assigning it to sprites. In void main() add the following line at the start:

```
// call function to copy all texture images from main memory
// to video RAM
LoadTextures();
```

The final thing you have to do now is to make the loadtex.c file be compiled. To do this you have to edit the makefile, so that next time you recompile the program, it will include the code from MEMTOOL into the final compiled main file. Edit the fifth line of makefile from:

```
OBJS    =    lib2d.o main.o pad.o
```

to the following

```
OBJS    =    loadtex.o lib2d.o main.o pad.o
```

That takes care of everything. If you want to add more files to be loaded later, then the only thing that ever needs altering is the MEMTOOL.DAT file, after which you rerun MEMTOOL, then recompile and that's it.

Create a GsSPRITE

The PSX datastructure that handles sprites is called GsSPRITE. It looks something like this:

```
struct GsSPRITE
      {
      u_long      attribute;
      short       x,y;
      u_short     w,h;
      u_short     tpage;
      u_char      u,v;
      short       cx,cy;
      u_char      r,g,b;
      short       mx,my;
      short       scalex,scaley;
      long        rotate;
      };
```

That's quite a lot for just a ship on the screen, but fortunately the SetSpriteInfo() function will take care of everything in there until you want to change it yourself. For now, just create an instance of this structure and call it mySprite, by adding the following lines at the top just before the void main() function, and what is actually in the GsSPRITE will be explained later:

```
// create an instance of GsSPRITE and call it mySprite
GsSPRITE    mySprite;
```

Assign the texture image to the sprite

Now, let's assign the just-loaded texture image to that sprite (otherwise you won't get any image on screen, if the program ran at all!). Just add this line in void main() before the main game loop, and it will nicely initialise all the stuff in the GsSPRITE structure for you.

```
// initialise all information in the GsSPRITE and center the sprite
SetSpriteInfo(&mySprite, EXAMPLE_TIM, 160, 120);
```

The first argument is the address of the GsSPRITE you want to initialise. The second tells the program which texture image to map to that GsSPRITE and should be a valid memory address with a texture image there, otherwise who knows what rubbish you'll end up with. EXAMPLE_TIM comes from the addrs.h file, which was created by MEMTOOL. The third and fourth arguments are simply the X and Y co-ordinates of where you want the sprite to start. Seeing as we are using screen resolution 320x240 as standard, the co-ordinate (160,120) puts the sprite nicely into the middle of the screen.

Draw the Sprite

And lastly, a command to draw the sprite on screen. Move into the RenderWorld() function, and between RenderPrepare() and RenderFinish() just add the line-

```
// draw the sprite on screen
DrawSprite(&mySprite, 0);
```

The first argument is obviously the sprite, but the second refers to the priority of the sprite. Priorities of drawing commands is much more important in 3d environments, but even in 2d you need to be able to specify which objects are in front of others. For example, keep your score in front of everything else, and the background behind everything else. On the PSX, the lower value you give the priority, the later it is drawn. Priority value 0 means it would get drawn after everything else, so that's OK here.

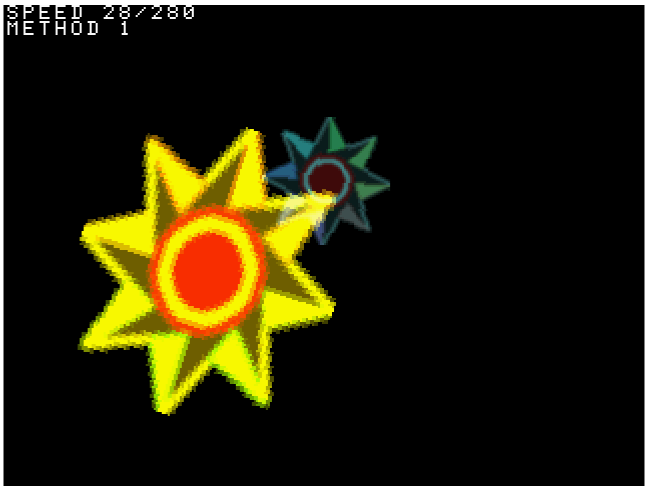
Run your program again, and hey presto, hopefully a nice little sprite in the middle of the screen. If not, its advisable to remember 90 percent of problems occur either because the sprite overlaps something else in video RAM which TIMutil should help avoid, or when loaded into main memory, which MEMTOOL avoids. A quick addition is to get the sprite moving around based on user input. This is handled from the ProcessUserInput() function:

```
// move sprite around up,down,left,right from user
if (PAD& PADleft) mySprite.x--;
if (PAD& PADright) mySprite.x++;
if (PAD& PADup) mySprite.y--;
if (PAD& PADdown) mySprite.y++;
```

You can now recompile and watch as you move the sprite around. You can add checks to make the sprite not move off screen, and you should be able to get more than one sprite on screen as well and perhaps program some sort of automatic movement into them.

2D STEP 5: Manipulating a sprite

FASTTRACK



```
void main()
      {
      BitSet(mySprite.attribute, (1<<30)+(3<<28));
      SetSpriteRGB(&mySprite, 32, 128, 255);
      mySprite.scalex = ONE*2;
      mySprite.scaley = -ONE;
      mySprite.mx = mySprite.my = 0;
      }

UpdateWorld()
      {
      mySprite.rotate += ONE;
      }
```

BitSet	set sprite transparency to on and use method 3
SetSpriteRGB	set brightness modulation of mySprite as RGB. 128 = default brightness
mySprite.scalex	rescale mySprite in x direction to twice normal size
mySprite.scaley	flip mySprite in y direction
mySprite.mx	move handle of mySprite to co-ordinate (0,0) of texture image
mySprite.rotate	rotate mySprite clockwise by one human degree (ONE = 4096 PSX units)

SLOWTRACK

The simplest thing that can be done with a sprite is simply to draw it on screen as it is. The PSX libraries allow you to alter the sprite in a variety of ways, e.g. rotate the sprite, make it semitransparent and scale the sprite larger or smaller as well as flip it. An important point to note about the transformations is that they operate about the current 'handle' of the sprite. The handle is a point on the sprite by which it is controlled. As standard, the handle of each and every sprite is in the center, meaning if you put at sprite at screen co-ordinate (0,0) then you would only see the bottom right part of the sprite. (The top part and left part would be off the visible screen boundaries, so would not be drawn). This has been set up to allow more intuitive control over the sprites. The reason this is important is that if later you wish to move the handle, then the sprite may transform in a way unexpected (and also cause problems using the standard collision detection routines). As it is for now, just remember that everything is applied via the handle in the center of the sprite.

Sprite Scaling, Flipping and Rotation

As standard, each sprite is drawn at the same size as the texture image applied to it. This can be overridden, so that the texture image will shrink or stretch. This is easily achieved by using the scalex and scaley members of the GsSPRITE structure. The initial values for both of these are 4096, which the PSX understands as '100%'. If you alter both of these values to 2048 (which is half of 4096) then the sprite on screen will appear at '50%' size. This would reduce a sprite that is 32x32 to 16x16. To double the size of a sprite you would set scalex and scaley to 8192 (double 4096). Points to note about scaling are that scalex and scaley do not have to be the same (you can stretch a sprite in only one direction, for example), and that the maximum amount you can scale a sprite in either the X or Y direction is to 800% (eight * 4096 = 32768). One of the hardware considerations is that the PSX does not like scaling large sprites even larger. There reaches a certain point where the scaling operation will either a) not work, or b) slow the machine's operation down severely. There are architectural reasons for this, but in most cases neither of these symptoms will arise through normal use.

Lets start messing around with your sprite. Taking the basic sprite program you had earlier, you want to be able to alter the scale of the sprite. The best way to do this is to add the following lines to the ProcessUserInput() function after the checks for moving the sprite:

```
// triangle will increase the y scale of the sprite
if ((PAD& PADtriangle) && (mySprite.scaley<(7*ONE))) mySprite.scaley +=256;
// cross will decrease the y scale of the sprite
if ((PAD& PADcross) && (mySprite.scaley >0)) mySprite.scaley -=256;
// circle will increase the x scale of the sprite
if ((PAD& PADcircle) && (mySprite.scalex<(7*ONE))) mySprite.scalex +=256;
// square will decrease the x scale of the sprite
if ((PAD& PADsquare) && (mySprite.scalex >0)) mySprite.scalex -=256;
```

When recompiled and run, you will find you can mess around with the size of the sprite quite happily and scale it up to seven times as large as it is originally. That really is all there is to scaling the sprite! You may understand better about the handle after running the program as well; you can see that the sprite scales from the center.

Flipping the sprite in either the horizontal or vertical directions is just as simple. You may have noticed that the scalex and scaley members of the GsSPRITE structure are signed shorts, which means they can have negative values. Guess what happens if you set the scalex of a sprite to -4096? The sprite gets flipped left to right. Of course, you can combine scaling and flipping as well, so a negative value of 8192 for the scaley member would not only flip the sprite top to bottom, but stretch it to twice the original hieght as well. To see this in action, we'll just edit a couple of the lines in the code that were just put into the ProcessUserInput () function:

change

```
if ((PAD& PADcross) && (mySprite.scaley>0)) mySprite.scaley -= 256;
if ((PAD& PADsquare) && (mySprite.scalex >0)) mySprite.scalex -= 256;
```

to

```
if ((PAD& PADcross) && (mySprite.scaley (-7*ONE))) mySprite.scaley -= 256;
if ((PAD& PADsquare) && (mySprite.scalex>(-7*ONE))) mySprite.scalex -= 256;
```

When recompiled and run, you will find you can shrink objects and then keep going, so that the sprite is now flipped along one of its axes.

The last transformation that can be applied to sprites is to rotate them. Note that an object that has been both scaled and rotated will be scaled first by the computer, and then rotated which may be important in some cases. Rotating a sprite is as simple as scaling, you just have to mess around with rotate member of the GsSPRITE. Again, owing to the PSX not using floating point numbers wherever possible, you have to use some strange values. Humans use 360 degrees, but the PSX uses a completely different unit, whereby one human degree is equivalent to 4096 units. So to rotate a sprite clockwise by 45 human degrees, you would set the rotate member to (45 * 4096) 184320! These numbers can be too large to handle, so again we make use of the constant ONE, which is equivalent to 4096. Add the following code to the ProcessUserInput () function as before:

```
// use L1 to rotate the sprite anti-clockwise by one human degree
if (PAD& PADL1) mySprite.rotate -= ONE;
// use R1 to rotate the sprite clockwise by one human degree
if(PAD& PADR1) mySprite.rotate += ONE;
```

With this in place, you can now move, rotate, scale and flip the sprite to your hearts content. There is processor slow down involved with these operations, but this is fairly negligible (unless you want to scale and rotate several hundred sprites!)

Sprite Semitransparency Effects

Another really useful operation that can be applied to sprites is to turn them semitransparent, causing the background to be partly visible through the sprites. Something to bear in mind is that using semitransparency means it takes three times as long to draw a sprite, but like scaling and rotation, used judiciously it can give a very pleasing effect. The PSX gives four different methods of drawing a semitransparent sprite on screen, each with their own good and bad points. These four methods are as follows:

Method	
0	final screen pixel col. = current screen pixel col. *50% + sprite pixel col. * 50%
1	final screen pixel col. = current screen pixel col. + sprite pixel col.
2	final screen pixel col. = current screen pixel col. - sprite pixel col.
3	final screen pixel col. = current screen pixel col. + sprite pixel col. * 25%

Table for semitransparency methods

Trying to explain how these appear on screen can be slightly awkward, so watching the effects on screen is the easiest way to understand them, but here is an attempt anyway. Method 0 takes the average of both the background and the sprite. E.g. if the background was light grey and the sprite dark grey, the result would be a middle grey. This transparent effect will make the result tend towards middle grey all the time, regardless of either colour. Method 1 adds both colours together, but leads to the funny result of possible having colours that are above 100% white. Obviously, this isn't possible, so any result above maximum is clipped to the maximum. In this case if the background was green and the sprite red, the result would be a yellow. Colours will always tend towards white. Method 2 is a subtractive method, and causes the results to tend towards black. If the background was white, and the sprite was green, the result would be purple (from the remaining red and blue). Again, this can give results that are below black, but in that case they are just drawn black. This method gives the strangest results as well. Method 3 is similar to method 1 but the sprite colour used is only 25% of what it should be. This tends towards white, but is less likely as with method 1.

method	screen pixel RGB	sprite pixel RGB	final pixel RGB
0 = 50% + 50%	(128,192,192)	(0,128,64)	(64,170,128)
1 = 100% + 100%	(128,192,192)	(0,128,64)	(128,255,255)
2 = 100% - 100%	(128,192,192)	(0,128,64)	(128,64,128)
3 = 100% + 25%	(128,192,192)	(0,128,64)	(128,255,208)

Table for semitransparency methods with examples.
Each pixel colour is given as an (RGB) value

Using these different methods is quite simple. First of all you have to tell the sprite to be drawn semitransparently, and the attribute member of the GsSPRITE controls this. Bit number 30 is clear when the sprite is to be drawn solidly, and set when it is to be drawn semitransparently. Include this line in your void main() function of the program after the sprites have used SetSpriteInfo().

```
// turn on your sprite transparency
BitSet(&mySprite.attribute, 1<<30);
```

What that second line does is use a macro to edit only bit 30 of the sprite. 1<<30 uses bit shifting, whereby you start with the number 1 (which in binary is the same) then shift all bits along 30 places, which leaves you with 1000000000000000000000000000000 in binary, correctly indicating which bit to set. Bitshifting is a very fast operation and used in many situations for quick multiplication and division by powers of two, (bitshifting a number left by one position doubles that number).

As default, the sprite will now be drawn using method 0. To alter that you need to edit bits 28 and 29 of the attribute bit. However first, lets think about what is necessary here. What you really need is something other than a plain coloured background over which you can see the semitransparency effects more clearly. Also, a way of selecting one of the four methods is required. Drawing another sprite on screen would give a patterned background, so that's just what we will use. Below the line where you declare mySprite at the top we will declare another, called backSprite.

```
// declare another sprite, simply to show off the other sprites effects
GsSPRITE    backSprite;
```

In void main, underneath the lines where you set the information for mySprite, add the following line:

```
// set up start information for backSprite
SetSpriteInfo(&backSprite, EXAMPLE_TIM, 160, 120);
```

This will make it use the same texture image as mySprite, and put it at the center of the screen.

Lastly for this part, add a line to draw the sprite. In RenderWorld() add the following line somewhere between RenderPrepare() and RenderFinish():

```
// draw backSprite
DrawSprite(&backSprite, BACK);
```

BACK is a macro that makes sure the sprite is drawn behind everything else in the scene. Sorting sprites into layers like this is covered later, but for now be rest assured that mySprite will always be drawn on top of backSprite. Now to control the semitransparency of mySprite. Declare a global variable at the top (after where you declare both sprites):

```
// declare a variable to keep track of mySprite's semi-transparency
u_long            methodValue = 0;
```

Notice it is declared as a u_long. As mentioned before, the PSX processor is 32bit, and the fastest variable type to use is an unsigned long as other types require padding when in the registers. Time to add some lines to the ProcessUserInput () function to allow the user to alter the state of mySprite. As the only buttons left unused at the moment, L2 and R2 would seem to be ideal to rotate through the methods from 0 to 3. Unless the user is a superhuman who can hold down one of these buttons for only 1/50th of a second, there has to be a way of keeping track of whether these buttons have just been depressed, or are still depressed from the previous frame. To do this, another variable will be declared within ProcessUserInput (), called L2R2depressed. Add the following line at the start of that function:

```
// variable to hold whether buttons have just been depressed or not
static u_long     L2R2depressed = FALSE;
```

The initial state indicates that neither button is depressed at the moment. Now to add the lines to alter the method state from within the ProcessUserInput () function. Add the following lines:

```
// first check whether L2/R2 are not already depressed
if (L2R2depressed == FALSE)
        {
// if not then check if L2 pressed down and if so then decrease
// methodValue and signify L2/R2 depressed
        if ((PAD& PADL2) && (methodValue>0))
                {
                methodValue--;
                L2R2depressed = TRUE;
                }
// if not then check if R2 pressed down and if so then increase
// methodValue and signify L2/R2 depressed
      if ((PAD& PADR2) && (methodValue<3))
            {
            methodValue++;
            L2R2depressed = TRUE;
            }
      }
// if L2/R2 are flagged as already depressed then check if either
// L2 or R2 is still being pressed, and if they aren’t, then clear
// the L2/R2 flag
else
      {
      if (!(PAD& PADL2) && !(PAD& PADR2)) L2R2depressed = FALSE;
      }
```

So that bit of code will let you press R2, and as long as you hold it down, the methodValue will only increase once. In order to get it to increase again, you have to release R2 and press it again. No more lightning touches are required to the pad! The two things left to do are show on screen which of the method values are currently selected, and also to actually set the sprite to use this value! In the RenderWorld() function, include the following lines:

```
// print on screen the currently selected methodValue
switch(methodValue)
      {
      case 0:     FntPrint(“Method 0: 50PC + 50PC\n”); break;
      case 1:     FntPrint(“Method 1: 100PC + 100PC \n”); break;
      case 2:     FntPrint(“Method 2: 100PC - 100PC \n”); break;
      case 3:     FntPrint(“Method 3: 100PC + 25PC \n”); break;
      }
```

Into the UpdateWorld() comes the section to set the sprite semitransparency (finally!). The method will be to reset the method used to 0, then back to the correct one every frame. This avoids having to keep track of the old method used when switching. Add the following lines:

```
// clear bits 28 and 29
BitClear(&mySprite.attribute, 3<<28);
// set bits 28 and 29 to the correct value
BitSet(&mySprite.attribute, methodValue<<28);
```

When recompiled and run you should now be able to use the two shoulder buttons to flick through the four different semitransparent methods, and by moving around over the other sprite on screen you can see exactly how they work. Semitransparency rates are not usually changed all that much in games, as for finer control you alter the sprite colour information, which fortunately is next. What your program also demonstrates at the moment is terrible design, in that the functions on the joypad are overwhelming. A well-planned and intuitive control system will make a game that bit more enjoyable, while a poor one will only annoy gamesplayers.

Sprite Colour Information (r, g and b members)

After all the information at the start about how sprite CLUTs are stored I'm sure you'll be glad to hear that there is another way of affecting the colour of your sprite. These are the r, g and b members of the GsSPRITE structure. These work by modifying the values read from the CLUT when the sprite is drawn on screen. By default the r, g and b members are set to 128, which means 'draw the sprite, with no colour modifications to the CLUT'. If you set all of the r, g and b members to 64 (half of 128) then when the sprite is drawn the CLUT entries are halved as well. When all three members are set to 0, then the CLUT is dimmed the most, and all you will see on screen is a black shape. Conversely, when the r, g and b values are set above 128 then the CLUT will be edited so the sprite appears brighter than it should. At the maximum value of 255, then sprite will appear twice as bright as normal. Each of the r, g and b members can be set independently. For example, a sprite with r set at 128, and g and b at zero will be drained of all green and blue, and appear as though through a red filter. You can set these members directly, or use the setSpriteRGB() function to set all three at once. As an example, add the following lines to the program, in the void main() function after both sprites have been initialised:

```
// alter the colours of the two sprites using the rgb members
// mySprite will appear bright yellow
SetSpriteRGB(&mySprite, 255, 255, 0);
// backSprite will appear a dimmer blue
SetSpriteRGB(&backSprite, 32, 64, 64);
```

Messing around with these values can give you even tighter control over how you want your sprites to appear, and can be extremely useful in a variety of circumstances. For example, if you have a two player game, rather than keeping two texture images of the same ships in video memory, each a different colour, you can store one grey texture image. With this you can then modify the GsSPRITES for each player to give one player a red ship, and one a blue ship. Another common example is in a shoot-em up game, when you shoot a ship, you can set the r, g and b members to 255 for one screen redraw only, giving the ship a white flicker that indicates it was shot.

Other Sprite Attributes

After scaling, flipping, rotation, altering semitransparency and colour modification, there isn't that much left to do with a sprite. The following sections covers those remaining members and functions and also takes a closer look at some of the ones that have been used so far. Let's take another look at the GsSPRITE structure.

```
struct GsSPRITE
      {
      u_long      attribute;
      short       x,y;
      u_short     w,h;
      u_short     tpage;
      u_char      u,v;
      short       cx,cy;
      u_char      r,g,b;
      short       mx,my;
      short       scalex,scaley;
      long        rotate;
      };
```

The GsSPRITE attribute member

There are plenty of other useful parts to the attribute member and their effects are as follows.

attribute bit	Effect	Values
6	Brightness regulation	0 = on, 1 = off
27	Rotation/scaling processing	0 = on, 1 = off
28-29	Semitransparency method	0 to 3 = different methods
30	Semitransparency processing	0 = off, 1 = on
31	Sprite display	0 = displayed, 1 = hidden

Table showing useful attribute member bits

Brightness regulation means simply whether the values in the r,g and b members of the GsSPRITE are taken into account when the sprite is drawn. If bit 6 is set, then r,g and b are ignored and a value of 128 is used in their place.

Rotation/scaling processing means whether to use or ignore the values in the scalex, scaley and rotate members. If bit 27 is cleared, then the sprite can be rotated and scaled and flipped. If bit 27 is set, then none of these functions are possible, but it does speed up processing of the sprite. (So ideally should be set if those functions are not required)

The semitransparency method has been described above. Note that bit 30 has to be set for the transparency method to take effect.

Sprite display means whether or not the sprite is drawn. It is possible to set bit 31, and then use the DrawSprite() function, and it will simply ignore it. If suitable to the application, it can be quicker to set and clear this bit to control sprite display, rather than do a check for every sprite in memory, see if it should be drawn, and if so then use the DrawSprite() function.

Use the BitSet() and BitClear() functions to edit the attribute member.

The GsSPRITE ‘handle’

As stated above, the default position for the sprite handle is in the center of the sprite. This allows simple collision detection routines to be programmed, and also for rotation and scaling effects to operate in an expected manner. But there are cases where you may want these operations to occur about a point on the sprite, other than the center. This is what the mx and my members are for. It is these that specify (as offset from the top left corner of the sprite) the co-ordinates of the handle. Initially mx = sprite width / 2, and my = sprite height / 2. By simply assigning other values, the handle can be moved. Keep the handle between the top left corner of the sprite (0,0) and the bottom right (width, height) as otherwise strange effects may occur. To show how this works, we will set the background sprite to rotate about it’s top right corner. To do this add the following lines to void main(), somewhere after the sprites have been initialised:

```
// move handle of backSprite to top right corner (sprite is 32x32)
backSprite.mx = 31;
backSprite.my = 0;
```

Now to make the sprite rotate. In the UpdateWorld() function, add the following line at the end:

```
backSprite.rotate += ONE*2;
```

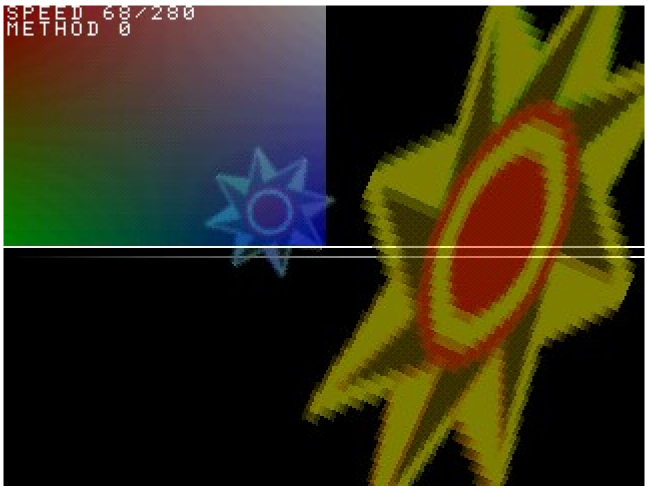
Remember that angles on the PSX are between 0 and 360*ONE, so this will rotate the sprite by two degrees each screen redraw. If you do edit the handle of the sprite, remember that this will more than likely affect how you use the standard collision detection routines. This is all covered later on, though, so don't worry about it for now.

CLUT manipulation and the GsSPRITE cx and cy member

The video RAM contains the frame buffers, all texture image information and any CLUTs that you want the texture images to use. These CLUTs are pointed to from within the GsSPRITE structures by the cx and cy members. You can legally reassign these members to any valid location in the video memory and the values there will then be used for that GsSPRITE. This means that you can have one texture image, and create multiple CLUTs for it, with different GsSPRITEs using different CLUTs.

Using the r, g and b members of a GsSPRITE allows individual sprite colour changes, but editing a CLUT entry will alter every sprite using that CLUT, and so has alternative uses.

2D STEP 6: Other graphics primitives



```
SPEED 68/280
METHOD 0
```

FASTTRACK

```
GsLINE      myLine;
GsGLINE     myGradLine;
GsBOXF      myBox;

void main()
      {
      SetLinePos(&myLine, 0, 0, 320, 200);
      SetLinePos(&myGradLine, 0, 40, 320, 240);
      SetBoxPos(&myBox, 160, 0, 320, 120);
      SetLineRGB(&myLine, 255, 255, 255);
      SetGradLineRGB(&myGradLine, 255, 0, 0, 0, 0, 255);
      SetLineRGB(&myBox, 128, 255, 128);
      }

RenderWorld()
      {
      DrawLine(&myLine, 1);
      DrawGradLine(&myGradLine, 2);
      DrawBox(&myBox, 3);
      DrawGradBox(0, 120, 160, 120, 4, -1,
            255, 0, 0, 0, 255, 0, 0, 0, 255, 255, 255, 255);
      }
```

GsLINE	instance of GsLINE line structure called myLine
GsGLINE	instance of GsGLINE gradient line structure called myGradLine
GsBOXF	instance of GsBOXF filled box structure called myBox
SetLinePos	set GsLINE and GsGLINE start and end points (x0,y0,x1,y2)
SetBoxPos	set GsBOXF position (x,y,width,height)
SetLineRGB	set GsLINE and GsBOXF colour (255 = full brightness)
SetGradLineRGB	set GsGLINE endpoint colours (r0,g0,b0, r1,g1,b1)
DrawLine	draw myLine at depth priority 1
DrawGradLine	draw myGradLine at depth priority 2
DrawBox	draw myBox at depth priority 3
DrawGradBox	draw gradient filled box at (0,120) of size (120,160), depth priority 4, solid filled, with corner colours (255,0,0), (0,255,0), (0,0,255), (255,255,255)

SLOWTRACK

Sprites are the single most commonly used 2d graphics primitive. The PSX does support several others, some of which are more useful than others. Their basic usage is similar to that of sprites; create an instance of the associated structure, set the important members of the structure, and send it to the ordering table with the correct function call. One area that can initially be confusing is learning to manipulate the attribute members of sprites, but fortunately all the graphics primitives use exactly the same method for controlling the important ways drawing of that primitive (most notably semitransparency).

Line drawing with GsLINE

The simplest drawing primitive is the line. The members of the GsLINE structure are as follows:

```
struct GsLINE
      {
      u_long      attribute;
      short       x0,y0;
      short       x1,y1;
      u_char      r,g,b;
      }
```

To draw a line, the least you have to set is the start coordinates of the line (x0,y0), the end point of the line (x1,y1) and its colour (r,g,b). We'll add a line in the middle of the screen to the current program as an example. At the start of the program, declare the following:

```
// Create an instance of a straight line called myLine
GsLINE      myLine;
```

then in void main() set it up as follows:

```
// initialise myLine structure
myLine.x0 = 0; myLine.y0 = 120;
myLine.x1 = 320; myLine.y1 = 120;
myLine.r = myLine.g = myLine.b = 255;
```

Alternatively you can use the macros SetLineInfo() and SetLineRGB() as shown in the fasttrack section. All that is needed to draw this white horizontal line is a call to the DrawLine() function from within RenderWorld():

```
// draw myLine (single colour line)
DrawLine(&myLine, 1);
```

The first argument (as you would expect) is the address of the line to be drawn, and the second is the priority. Note by giving it a priority of 1, it will get drawn behind mySprite (priority of 0) and in front of backSprite (priority of BACK). One thing to remember is that the r,g and b members are not identical to that of a sprite; a sprite is drawn at normal brightness at 128, with 255 double that. In other primitives, 255 is the brightest possible. For more interesting line drawing, it is possible to draw it semi-transparently in exactly the same way as a sprite. To turn on semi-tranparency processing, bit 30 has to be set, and then you must choose which of the four methods of semi-transparency to use.

Gradient line drawing with GsGLINE

GsGLINEs are the same as a GsLINE except that you specify one colour at the start point, and one at the end point, and the PSX will draw a smooth gradient between the two colours along the line. This can create very nice effects. The members of the GsGLINE are as follows:

```
struct GsGLINE
      {
      u_long      attribute;
      short       x0,y0;
      short       x1,y1;
      u_char      r0,g0,b0;
      u_char      r1,g1,b1;
      }
```

The attribute member functions in exactly the same way for the other graphics primitives. To add one to the demo program, declare the following:

```
// create an instance of a gradient line called myGradLine
GsGLINE     myGradLine;
```

In void main(), set up the member variables as follows:

```
// initialise myGradLine structure
myGradLine.x0 = 0; myGradLine.y0 = 125;
myGradLine.x1 = 320; myGradLine.y1 = 125;
myGradLine.r0 = myGradLine.g0 = myGradLine.b0 = 255;
myGradLine.r1 = myGradLine.g1 = myGradLine.b1 = 0;
```

or alternatively set it up using the macros SetLineInfo() and SetGradLineRGB() as shown in the fasttrack section. Then include a call to the drawing function from within RenderWorld():

```
// draw myGradLine (gradient line)
DrawGradLine(&myGradLine, 1);
```

The arguments are the same as for drawing a normal line.

Filled box drawing with GsBOXF

The final graphics primitive draws a filled box on screen. As you might expect, the structure is as follows:

```
struct GsBOXF
      {
      u_long      attribute;
      short       x,y;
      u_short     w,h;
      u_char      r,g,b;
      }
```

This structure allows you to draw a box from x,y with width of w, and hieght of h. Note that the width and height must be positive. The associated function to draw a GsBOXF is:

```
// use this is you want to draw a filled box on screen
// drawBox(& of GsBOXF, priority);
```

You can use the SetBoxInfo() macro and bizarrely enough, the SetLineInfo() macro to set the colour.

Interesting effects with the graphics primitives

Two nice things that are possible with these primitives are using a filled box to fade out the screen to black as you see in a lot of games and on TV rather than simply cutting to the next scene in your game, and also to draw a box on screen with different colours in different corners.

For the first effect you need to create a GsBOXF the size of the screen, set its r,g and b members to zero, and its drawing method to semitransparent, and using the subtractive method 2 (`BitSet(&myBox.attribute, (1<<30) + (2<<28));`). When you draw it, make sure it gets drawn in front of everything else on screen. The first time it is drawn there will be no effect on the screen, but each frame if you increase the r, g and b members by 1, then each successive drawing of the screen will get darker and darker as the box reduces the brightness of graphics on screen. When r, g and b are 255, then the screen has fully faded to black.

To draw a box with colours fading from each corner you can use the already created function DrawGradBox(). The way this works is to draw gradient lines horizontally for each line in the box, using modified start and end colours to ensure the colours fade both across the box and from top to bottom.

```
DrawGradBox(x, y, w, h, priority, draw method,
      r0, g0, b0,
      r1, g1, b1,
      r2, g2, b2,
      r3, g3, b3);
```

The box will be drawn at (x,y) with width and height of (w,h) and with priority (priority). The draw method encapsulates solid drawing and semitransparency in one argument. To draw the box solidly, draw method should be –1. To use one of the semitransparent drawing methods, simply pass the semitranparency method number without bitshifting it first. (r0,g0,b0) is the colour in the top left corner, (r1,g1,b1) the top right, (r2,g2,b2) the bottom left, and (r3,g3,b3) the bottom right. For example, add this inside your RenderWorld() function:

```
// draw a gradient coloured box on screen
DrawGradBox(0, 0, 320, 240, BACK, -1,
      0, 0, 255, 255, 0, 0, 0, 255, 0, 255, 255, 255);
```

2D STEP 7: Creating a simple 2d shoot-em up

At the moment you’ve got a program that simply illustrates all of the possible things you can do with 2d graphics. From now on we will be striving towards writing a working game, based on the design document (or website) for the 2d game. This will allow a more practical demonstration of new topics as they arise. The most important aspect to programming a game (especially in the tight time limits imposed on the course) is to have a clear and detailed plan for the game.

When coding programs yourself you tend to get a basic version running, and then keep adding bits and pieces until the product resembles what you want. In this example, most of the code will be added all at once, with only the later versions being executable. Having a decent plan and writing code that is flexible allows you to code more at any one time, rather than adopting a more usual hit and miss approach.

The game code template and program flow

As a starting point, we will start with the basic program that does nothing. It is almost identical to the normal template main.c file. The game will have two main loops; the first being the loop to handle the title screen, and the second to handle the actual game. The easiest way to do this is to have two sets of each of the ProcessUserInput (), UpdateWorld() and RenderWorld() functions. Without it these functions would become messy. The first lot of functions will have Title appended to the end, with the second lot having Game appended to the end to aid readability. The main program flow will now continually switch from the title screen to the game and back again, until a key pressed on the title screen will cause the whole thing to exit.

The bare program will look like this to begin with, hopefully with everything easy to understand:

```
// include PSX, Middlesex libraries + #defines for texture images
#include <libps.h>
#include “lib2d.h”
#include “addrs.h”

// function prototypes for title screen
void UpdateWorldTitle();
void RenderWorldTitle();
u_long ProcessUserInputTitle();

// function prototypes for in game shooting
u_long UpdateWorldGame();
void RenderWorldGame();
u_long ProcessUserInputGame();

// main program loop
void main()
      {
// variable to hold if the program exit condition has been reached
      u_long GameRunning = TRUE;
// variable to hold the exit condition from both the title screen and
// game loop. This is required to avoid running through the game
// loop once the program exit condition has been reached in the title loop
      u_long LoopState = 0;

// prepare GPU for displaying buffers from video RAM
      Initialise2DGraphics();
// copy textures from main memory to video RAM
      LoadTextures();

// main program loop
      while (GameRunning == TRUE)
            {
// this loop controls the title screen
            while (LoopState == 0)
                  {
// Joypad function returns 0 for continue with title screen,
// 1 for start game, and 2 for quit from program
                  LoopState = ProcessUserInputTitle();
                  UpdateWorldTitle();
                  RenderWorldTitle();
                  }

// if title screen indicates quit, then set the gamerunning flag to
// exit, otherwise reset loopstate to zero to make sure the game loop
// runs
            if (LoopState == 2) GameRunning = FALSE;
            else LoopState = 0;

// this loop controls the game section
```

```
            while (LoopState == 0)
                  {
                  LoopState = ProcessUserInputGame();
                  LoopState += UpdateWorldGame();
                  RenderWorldGame();
                  }

// reset loopstate ready for running the title screen loop
            LoopState = 0;
            }
// reset the PSX ready for program quitting
      ResetGraph(0);
      }
```

None of the other functions have been designed yet, but so far the only major difference is that the ProcessUserInput() functions and the UpdateWorldGame() function return a value. In the title screen, the only exit conditions will come from pressing certain keys on the joypad, so that function will return a value to indicate when that occurs. In the game loop there are two exit conditions. The first is from pressing a certain key to return to the title screen, and the other is when the player runs out of lives. This should all make sense as we start to add the basic functions. The easiest at the moment is to do the title screen functions, as at the moment all that does is display some text on screen and wait for the user to press a key. Try not to get too excited at the title screen; you can add a picture and proper menu system later if you want.

```
// specify the text to be drawn on the title screen
void UpdateWorldTitle()
      {
      FntPrint(“Shoot-em up demo\n\n\n”);
      FntPrint(“Press –X- to start the game\n\n”);
      FntPrint(“Press –START- and –SELECT- to quit”);
      }

// the renderworld function needs nothing added, as it already
// outputs the contents of the text stream
void RenderWorldTitle()
      {
      RenderPrepare();
      RenderFinish();
      }

// the joypad function controls when the program moves from title
// loop to game loop. This is reflected in the returned value
u_long ProcessUserInputTitle()
      {
      u_long ReturnedValue = 0;
      u_long PAD = ReadPad();

// these are the only conditions that will affect the title loop
      if (PAD& PADcross) ReturnedValue = 1;
      if ((PAD& PADstart) && (PAD& PADselect)) ReturnedValue = 2;
      return ReturnedValue;
      }
```

That was pretty easy to get out of the way, and the rest of game will evolve from now on, allowing the introduction of more programming tuition.

2D STEP 8: Depth sorting of 2d primitives

FASTTRACK

Priorities are used to align 2d graphics on z sorted planes
Priorities for 2d objects range from 0 (drawn in front of everything) to BACK (drawn behind everything). BACK = (1<<ordering table length) – 1.
Ordering table length in "lib2d.h" is 10. BACK in this case = 1023
Priorities specified at time of registering drawing command
One drawing command will appear in front of another if given same priority but registered before the other

SLOWTRACK

As you already have noticed, when you register your intention to draw a 2d primitive (sprite, lines or boxes) you not only have to specify which of your objects to draw, but also a depth or 'priority' argument. The rules for using depth sorting are simple within a 2d environment and are used to ensure which objects get drawn in front of each other. In other graphics system, the order that objects are drawn is a result of the order in which the drawing commands are executed, but with the PSX drawing commands are stored as a list pointed to by the ordering table, allowing greater flexibility during coding. When you create the ordering table (in the case of the supplied libraries, carried out in either Initiailise2DGraphics() or Initialise3DGraphics() functions) you define how many layers or priorities you want to create. The possible number of these layers range between 2 (2^1) to 16384 (2^{14}), but in the supplied functions it is either 1024 (2^{10}) or 4096 (2^{12}). This means that you can specify a priority of anywhere between 0 and 4095 inclusive. The larger the priority you give the drawing command, then the earlier it gets drawn, and thus obliterated by any overlapping drawing commands with lower priority values. For example, a sprite with priority of 5 will appear to overlap a sprite with priority of 15. With 2d games, there is no great advantage in having a huge amount of priorities, but (as you will see later) the more you can have with 3d graphics, the better. For now, simple use of this depth sorting means you won't get bits of the background obscuring sprites that should be at the front. A nice way of ensuring something is at the back of the screen is to use BACK instead of passing a number as a priority value. BACK is automatically set to the largest priority possible.

When you specify more than one sprite to use the same priority, then the order that you specify the drawing commands does matter. The ordering table is a linked list, and every new drawing command is added to the start of the list for each priority.

```
// example of drawing two sprites with the same priority
// this will be drawn over the other sprite
DrawSprite(&spriteOne, 10);
// this will be drawn under the other sprite
DrawSprite(&spriteTwo, 10);
```

The result of that code would appear on screen as spriteOne overlapping spriteTwo. As spriteTwo is added to the start of the linked list for priority value 10, then it will be processed before spriteOne, and therefore be drawn first, with spriteOne being drawn over the top. Obviously, if there are certain sprites you want to ensure the order of overlapping, you would be better off assigning a different priority value. For something like bullets, where the order doesn't matter, you could safely assign them all the same priority.

Text is always drawn over everything else. As it is handled slightly differently to other drawing, there is no simple way to affect that ordering.

Planning priorities for the game

With this is mind, here are some rough ideas for priorities for the 2d game that we're going to design:

Priority	Feature
0	Used for the rectangle that can fade the screen in and out (not necessarily to be included, but it is easier to allow room for adding more stuff at the beginning)
10	Used for information output to the screen (such as lives remaining and power left)
20	Explosions
30	Your ship
40	Enemy ships
50	Your bullets
60	Enemy bullets
BACK	The scrolling background

Table for planned priorities for game elements

Notice that I've gone up in steps of 10, simply to allow any fine-tuning later on. Seeing as there are a possible 4096 priorities, I don't think it matters too much!

2D STEP 9: Preparing your ship, bullets and enemy sprites

Now to include the code for the different sprite based elements in the game (your ship, bullets, enemies and explosions). At the start of the program, add the following (based on the design document):

```
// create a structure for your bullets and enemies
typedef struct
      {
      u_char      state;
      GsSPRITE    image;
      short       xadd, yadd;
      } GENERICTYPE;

// create a structure for explosions
typedef struct
      {
      u_char      colour;
      GsSPRITE    image;
      short       xadd, yadd;
      } EXPLOSIONTYPE;

// create a structure for scaled enemy bullets
typedef struct
      {
      u_char      state;
      GsSPRITE    image;
      long        xpos, ypos;
      long        xadd, yadd;
      } SCALEDTYPE
```

Now to create the instances of these structures based on the numbers from the design document.

```
// create instances for you, bullets, enemies and explosions
GsSPRITE          yourShip;
GENERICTYPE       yourBullets[25];
SCALEDTYPES       enemyBullets[25];
GENERICTYPE       enemyShips[25];
EXPLOSIONTYPE     explosions[25];

// create instances of sprites for the lives left indicator
GsSPRITE          livesLeftSprite[3];
```

These all need to be set up with the correct texture image information as defined in the design document. In this case the following routine, added to the start of void main() will do the trick. This includes some of the previously mentioned attributes of sprites, such as for semitransparency.

```
// create a variable for the initialisation loops
u_long      counter = 0;

// initialise all the sprites
// note that for now the specified x and y is irrelevant
setSpriteInfo(&yourShip; YOURSHIP_TIM, 0, 0);
for (counter=0; counter<25; counter++)
      {
      setSpriteInfo(&yourBullets[counter].image; BEAM_TIM, 0, 0);
      setSpriteInfo(&enemyBullets[counter].image; BULLET_TIM, 0, 0);
      setSpriteInfo(&enemyShips[counter].image; ENEMY_TIM, 0, 0);
      setSpriteInfo(&explosions[counter].image; EXPLODE_TIM, 0, 0);
// set the speed of your bullets now, as they never alter
      yourBullets[counter].xadd = 16;
      yourBullets[counter].yadd = 0;
      }
// setup lives left sprites in top left corner
for (counter=0; counter<3; counter++)
      setSpriteInfo(&livesLeftSprite[counter], LIFELEFT_TIM,
            counter*16+16, 16);
```

This will set up the texture images for the sprites, but what is needed is a routine to make sure that any of the members of these structures that can be altered during the game get reset to the initial condition. This is required to avoid data residing from the previous time it was played, which might cause problems. To do this, we will create a new function called

InitialiseGameData(). Add the following prototype and code to your program and it should make sense:

```
// function prototype for resetting game data before playing
void InitialiseGameData();

// function for resetting game data before playing
void InitialiseGameData()
      {
      u_long counter;
      for (counter=0; counter<25; counter++)
            {
// turn all bullets and enemies off, and make explosions bright
            yourBullets[counter].state = 0;
            enemyBullets[counter].state = 0;
            enemyShips[counter].state = 0;
            explosions[counter].colour = 255;
// position your ship in the center of screen to begin with
            yourShip.x = 160;
            yourShip.y = 120;
            }
      }
```

Now, the place to call this is in void main() after the title screen loop has been exited and before the game loop begins. Stick this call to the new function in:

```
// clear all game data ready for new game
InitialiseGameData();
```

2D STEP 10: Miscellaneous Code

Some of the stuff still to be added to the program has either already been introduced or doesn't require a large explanation. This part will add some more code that is specific to this game. What is needed is a way of keeping track of the number of lives, controlling the ship, and controlling which bullet gets released when you press the fire button, and also the bullets fired from the enemies. What usually happens in a shooting game is that when you get shot, you reappear, but invulnerable for a few seconds. Some way of keeping track of this, and representing it on screen is required as well. We'll use the r,g,b members of the ship to turn it a nice green colour while it is invulnerable (this will be handled within the UpdateGameWorld() function in the next step). Add the following code to the start of your program with the other global variables:

```
// declare life counter variable
static u_long     livesLeft;
static u_long     invulnerable;
```

This needs to be set up each time you play the game section, so add the following to the InitialiseGameData() function:

```
// set up number of lives per game
livesLeft = 3;
// set invulnerability to on at the start of each game
// as this will decrease by one each frame, a value of 100
// makes the ship invulnerable for 2 seconds
invulnerable = 100;
```

The bare bones of the UpdateWorldGame() function will be created, which at the moment will just handle the invulnerability of the ship. At the moment this function will always return a zero, because if you remember, a one returned will indicate that all the players lives are lost and will return the game to the title screen. As there is no way the player can die at the moment, this is superfluous for now. We will also print out the timer counter to show how much fast or slow the game is. The variable VerticalSync gets created and set automatically from within the included 2d libraries.

```
// function to handle everything that requires updating
u_long UpdateWorldGame()
      {
// return value indicates if the game is over (1 when this is TRUE)
      u_long returnedValue = 0;

// print out screen timer
      FntPrint(“time %d/280\n”, VerticalSync);
// decrement invulnerability counter
      if (invulnerable > 0) invulnerable--;
// set ship colour
      yourShip.r = 128-invulnerable;
      yourShip.g = 128+invulnerable;

      return returnedValue;
      }
```

The firing of the player’s ship will happen as follows: When the player presses the ‘X’ button, the ship will fire, and then will not fire again until the button has been released and pressed again. Seeing as it is unlikely that the player can press the fire button 25 times before the first button has disappeared off the screen (and thus become ready to be refired again) then a simple counter will register which bullet to release. Each time a bullet is fired, the counter will increment, and then point to the next ready bullet. This will be wrapped up in a FireYourBullet() function which will be called from the ProcessUserInput Game() function, which we will define now:

```
// process the user’s input during the game section
u_long ProcessUserInputGame()
      {
      static u_long yourBulletCounter = 0;
      static u_long isCrossPressed = FALSE;
      u_long ReturnedValue = 0;
      u_long PAD = ReadPad();
// simple ship moving routine
      if ((PAD& Padup) && (yourShip.y>8)) yourShip.y -=2;
      if ((PAD& PADdown) && (yourShip.y<231)) yourShip.y +=2;
      if ((PAD& PADleft) && (yourShip.x>8)) yourShip.x -=2;
      if ((PAD& PADright) && (yourShip.x<311)) yourShip.x +=2;

// if this is the first time you’ve pressed cross, then fire else dont
      if ((PAD& PADcross) && (isCrossPressed == FALSE))
            {
```

```
            isCrossPressed = TRUE;
            FireYourBullet();
            }
// check if you are not pressing cross
      if (!(PAD& PADcross)) isCrossPressed = FALSE;

// if you press START set returnedvalue to indicate you want to reset
      if (PAD& PADstart) ReturnedValue = 1;

      return ReturnedValue;
      }
```

Now to add the FireYourBullet() function. Add this function to your program:

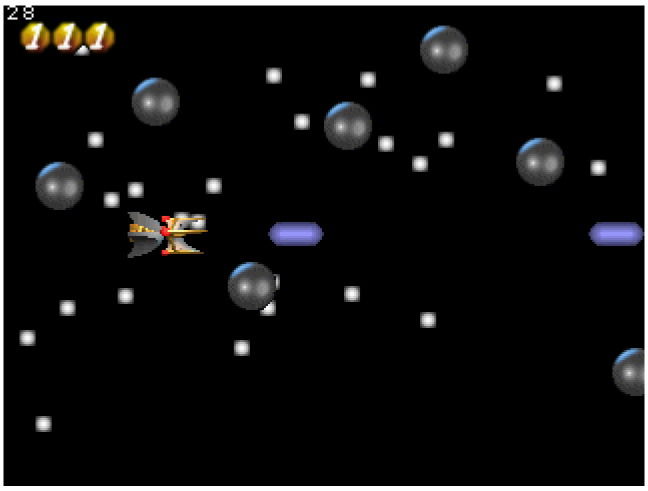
```
// function prototype for firing a player bullet
void FireYourBullet();

// function to set up a player bullet when fired
void FireYourBullet()
      {
      static currentBullet = 0;

// set the state member to ON, and move the bullet to under your ship
      yourBullets[currentBullet].state = 1;
      yourBullets[currentBullet].image.x = yourShip.x;
      yourBullets[currentBullet].image.y = yourShip.y;
// increment (and if needed loop back to 0) the current bullet
      currentBullet = (currentBullet+1) % 25;
      }
```

The section of code to turn bullets off will be in the UpdateWorldGame() function, which will be handled later, as it is here all movement and checks are done. Let's add code to draw something on screen. The RenderWorldGame() function will include a few if statements, which only draw sprites if they are flagged as active.

```
// draw all sprites that are currently active
void RenderWorldGame()
      {
      u_long counter;

      RenderPrepare();

// draw player ship
      DrawSprite(&yourShip, 30);
// draw enemy's and your bullets, enemies and explosions
      for (counter=0; counter<25; counter++)
            {
            if (yourBullets[counter].state == 1)
                  DrawSprite(&yourBullets[counter].image, 50);
            if (enemyBullets[counter].state == 1)
                  DrawSprite(&enemyBullets[counter].image, 60);
            if (enemyShips[counter].state > 0)
                  DrawSprite(&enemyShips[counter].image, 40);
            if (explosions[counter].colour > 0)
                  DrawSprite(&explosions[counter].image, 20);
            }
// draw lives left
      switch(livesLeft)
            {
            case 3:     DrawSprite(livesLeftSprite[2], 10);
            case 2:     DrawSprite(livesLeftSprite[1], 10);
            case 1:     DrawSprite(livesLeftSprite[0], 10);
            }
      RenderFinish();
      }
```

You can run the program now, but don't expect too much excitement!

2D STEP 11: Computer-controlling enemies



Creating the enemies and enemy bullets during the game

As written in the design document, the easiest way to release enemies at the player is simply to create a new one a certain amount of time after the previous one has been released. This will be handled by the UpdateWorldGame() function. If the situation arises when all 25 enemies are on screen at once, then trying to create a new one will simply result in failure, which is fine. At a guess (which can be optimised later) there will be a counter that attempts to release a new alien every quarter of a second. Add this line to the other declarations in UpdateGameWorld():

```
// variable to hold timer to release aliens
static u_long alienReleaseCounter = 0;
```

Add this just after the declaration in the same function:

```
// check to release an alien. If the game is updating 50 times a second,
// then every 12th time UpdateWorldGame() function called should be roughly
// a quarter of a second
      if (alienReleaseCounter == 0) CreateEnemy(1);
      alienReleaseCounter = (alienReleaseCounter+1) %12;
```

Only one of the types of aliens will be programmed at the moment, and it will be the simplest of them all, the one that enters the screen at the right, and then moves in a straight line to the left. The function will have to be general purpose, as it later has to be expanded to cope with 3 different alien types. The function only has to set up the initial position of the alien, it's movement vector and set the type of alien it is.

```
// function prototype for creation of aliens
void CreateEnemy(u_long type);
void CreateEnemyOne(u_long index);

// function for creation of an enemy (generic) calls others
void CreateEnemy(u_long type)
      {
      u_long counter;

// loop to check all enemies and find one not in use
      for (counter=0; counter<25; counter++)
            {
// if a not in use ship is found, then create a new one using it
            if (enemyShips[counter].state == 0)
                  {
                  switch (type)
                        {
// call CreateEnemyZero() function
                        case 1: CreateEnemyOne(counter); break;
// lines to create other enemies (commented out at the moment)
//                      case 2: CreateEnemyTwo(counter); break;
//                      case 3: CreateEnemyThree(counter); break;
                        }
// set counter so high as to cause the loop to finish
                  counter = 26;
                  }
            }
      }

// function to create enemy type 1 – moves right to left
void CreateEnemyOne(u_long index)
      {
// set its x position to off screen to the right
      enemyShips[index].image.x = 360;
// set its y position at random
      enemyShips[index].image.y = rand() %240;
// movement vector is (-4,0). Moves right to left
      enemyShips[index].xadd = -4;
      enemyShips[index].yadd = 0;
// set state to show it is alien type 1
      enemyShips[index].state = 1;
      }
```

Enemy bullets

Enemy bullets will follow a different creation routine to that of the player. When an alien fires a bullet, it will search through the list to find if there is one that is registered as 'not in use' and if there is, then it will fire that one. If there are no free bullets, then the alien simply will not be able to fire one. The reason the same routine as for the player's bullets isn't used is that with the player, it was unlikely that 25 of their bullets would be onscreen at once. With enemy bullets, you can't guarantee this, and it would appear strange upon creation of bullet number 26, that one already on screen would simply vanish. Time to think about what might need passing to a function that creates bullets. Unlike the player's bullets (where the movement of them is always constant and horizontal) an enemy's bullet will head towards the players. The function would need to know from what point on screen the bullet came from, and where it was heading. Seeing as they will always head towards the player that need not be passed, but the location on screen where it came from would alter all the time. How to calculate the movement vectors for the bullet? Now you might see the reason why we need the scaling system, as we are going to take the vector that would move the bullet to the player's position and divide by however many steps we want to take it there. This will need to be quite a large number otherwise the bullet will move too quickly to be avoided. And therein lies the problem; dividing integers by integers causes rounding problems, and you would find that the bullets would not move accurately. By scaling up the distance between player and bullet and dividing that value, then the rounding error is much smaller, and all you have to do is scale back down the answer at a later time. The routine should show how this works:

```
// function prototype for enemy firing routine
void FireEnemyBullet(short startx, short starty);

// function to position new enemy bullet and calculate its heading
void FireEnemyBullet(short startx, short starty)
      {
// specify where the player ship is, for bullets to head to
      short endx = yourShip.x;
      short endy = yourShip.y;
      u_long counter;

// loop the checks for a free (not used) bullet
      for (counter=0; counter<25; counter++)
            {
            if (enemyBullets[counter].state == 0)
                  {
// if a free bullet is found, it is moved to the enemy ship
                  enemyBullets[counter].image.x = starty;
                  enemyBullets[counter].image.y = startx;
// scale up the current position by 7
                  enemyBullets[counter].xpos = startx<<7;
                  enemyBullets[counter].ypos = starty<<7;
// as mentioned before, normally you would find the vector from bullet to
// player, then divide by the nummer of steps. As the vector has been
// increased by 2^7 (128), even without scaling the xadd and yadd it
// works out ok
                  enemyBullets[counter].xadd = endx-startx;
                  enemyBullets[counter].yadd = endy-starty;
// check bullet isn't moving too slow otherwise turn it on
                  if ((abs(endx-startx) > 32) && (abs(endy-starty) > 32))
                        enemyBullets[counter].state = 1;
// set the counter value high enough to cause the loop to exit
                  counter = 25;
                  }
            }
      }
```

Updating the enemies and all bullets

Now to add this bit to actually move the enemies and bullets and there should be something that you can run! During the update function the bullets and enemies are not just moved across the screen, but have to be checked if they 'dead' and are ready to be reused. The normal way for an alien or bullet to be switched off is when it goes off-screen, and the other is when it is involved in a collision, which will be dealt with in the next step. For now, a seperate routine that checks if a sprite is on screen or not would be useful. As the current x and y positions of a sprite are stored in the GsSPRITE structure, a pointer to that would be the most useful argument to pass, with a return result indicating whether it is still on or off screen.

```
// function prototype to check if a sprite is on screen
u_long CheckOnscreen(GsSPRITE &test);

// function to check if a sprite is on screen
u_long CheckOnscreen(GsSPRITE &test)
      {
// check if sprite is too far to left or right
      if ((test->x < -16) || (test->x > 336) ||
// check if sprite is too far up or down, if so return 1 otherwise return 0
      (test->y < -16) || (test->y > 256)) return 1;
      else return 0;
      }
```

With this routine written, we can now start adding to the UpdateWorldGame() function, which will move the sprites that are active, and set them to a not in use status when they go off screen. At the start of the UpdateWorldGame() function add the following variable declaration:

```
u_long counter;
```

and at the end of the function add this:

```
// loop that will update and check each of the object types
for (counter=0; counter<25; counter++)
      {
      if (yourBullets[counter].state == 1)
            {
// move your bullets if active
            yourBullets[counter].image.x += yourBullets[counter].xadd;
            yourBullets[counter].image.y += yourBullets[counter].yadd;
// turn them off if they are off screen
            if (CheckOnscreen(&yourBullets[counter].image) == 1)
                  yourBullets[counter].state=0;
            }
      if (enemyBullets[counter].state == 1)
            {
// move enemy bullets if active (remembering to scale them >>7
            enemyBullets[counter].image.x =
                  (enemyBullets[counter].xpos +=
                  enemyBullets[counter]. xadd) >> 7;
            enemyBullets[counter].image.y =
                  (enemyBullets[counter].ypos +=
                  enemyBullets[counter]. yadd) >> 7;
// turn them off if they are off screen
            if (CheckOnscreen(&enemyBullet[counter].image) == 1)
                  enemyBullet[counter].state=0;
            }
      if (enemyShips[counter].state > 0)
            {
// move enemy ships if active
            enemyShips[counter].image.x += enemyShip[counter]. xadd;
            enemyShips[counter].image.y += enemyShip[counter]. yadd;
// turn them off if they are off screen
            if (CheckOnscreen[counter].image) == 1)
                  enemyBullets[counter].state=0;
            }
      }
```

And with that, you should be able to run the program, and be amazed at this early stage of the game. You should be able to fire bullets and watch as ships simply appear at the right of the screen and fly off to the left. Of course, they don't fire bullets yet. When a particular alien fires will differ according to the alien type, and there is also a variety among how the alien types move. The programmed alien just moves from right to left at a constant velocity, but other aliens will require their movement vectors to be edited. A simple switch statement could call separate routines for each alien type at the end of the UpdateWorldGame() function:

```
// call the different movement alteration and firing routines
for (counter=0; counter<25; counter++)
      {
// call different controlling functions based on the type of alien
      switch (enemyShip[counter].state)
            {
            case 1:     ControlEnemyOne(counter); break;
// at the moment there is only one programmed enemy type, so the
// next two lines are commented out
//          case 2:     ControlEnemyTwo(counter); break;
//          case 3:     ControlEnemyThree(counter); break;
            default:    break;
            }
      }
```

This means you can now code a different control function per alien type, and when called it will tell you which alien to edit as well. As a quick method for getting aliens to fire, we will make them let off a bullet every quarter of the screen, which hopefully will give enough to look good. As the enemies appear at x co-ordinate 336, and move 4 pixels left each frame, then a simple way to trigger the bullets is when the x co-ordinate equals 320, 240, 160, 80 and 0. This is fortunate, as it leads to the very simple following function:

```
// function prototype for controlling enemy type one
void ControlEnemyOne(u_long enemy);

// function to cotrol and fire a bullet for enemy type one
```

```
void ControlEnemyOne(u_long enemy)
      {
// if ship x coordinate == 320, 240, 160, 80 to 0, then call fire function
      if ((EnemyShips[enemy].image.x %80) == 0)
            FireEnemyBullet(  EnemyShips[enemy].image.x,
                              EnemyShips[enemy].image.y);
      }
```

You can recompile the code now, and watch as the ships will fly across the screen, releasing bullets as they go, which should all home in on the player. You can start noting things to touch up later, such as are there enough bullets, do they move too fast/slow, how can you make the aliens more impressive? For now, as long as the bullets don't move at a lightning speed which are impossible to dodge and will cause problems when collision detection is included then you can continue. It is always easy getting side tracked into fine-tuning your program as you go along but this is not always a good use of time, as adding later code might alter the characteristics of the game, negating all the time you spent earlier as you only have to retune the game again.

2D STEP 12: Collision detection

FASTTRACK

```
void useinanyfunction()
      {
      RECT  rect1, rect2;
      SetBoxInfo(&rect1, 0, 0, 20, 20);
      SetBoxInfo(&rect2, 10, 10, 20, 20);
      CheckRectOverlap(&rect1, &rect2);
      }
```

RECT	create two instances of RECT for the collision detection
SetBoxInfo	use to set (x,y,width,height) of the two RECTs
CheckRectOverlap	returns 0 if the two RECTs don’t overlap, otherwise returns 1 (as in above example)

SLOWTRACK

Collision detection is one of the most important aspects of a game. Ultimately a game can succeed or fail on how accurate the player feels the game is. There is nothing more annoying in a racing game to collide with something that is nowhere near the player, or to find an enemy in a shoot-em up surviving a bullet that passed through it! With this in mind, there are now two considerations when programming collision detection. The first is for the actual routine. What limitations does it impose on the game? A pixel-perfect approach to collision detection may be desirable, but the overall costs on processing time when there are more than a few checks to be done soon mount up. The generally easiest and fastest implementation for collision detection is to use the idea of 'bounding volumes'. The second consideration occurs mostly during testing, when you fine-tune the routine to keep the games player happy. For example, in a shoot-em up any check between you and an enemy bullet may err on the side of caution and require the bullet to pass very close to the center of the ship to register a hit, while a player's bullet need only graze an enemy ship for it to be destroyed.

Bounding rectangles are the simplest method of checking for collisions in 2d, and the following example will explain what lies within the included collision detection functions. The general function requires eight parameters to be passed to it (the top left and bottom right co-ordinates of the two rectangles to be checked). A quick look at the maths means that collision detection is going to take a lot of processing time (25 of your bullets * 25 enemies = 625 checks plus 25 enemy bullets to check against your ship = 650 checks in total). This gives an indication of why collision detection routines have to be fast and somewhat simplified as the problem can spiral out of control rapidly. For example, 20 bullets and 20 aliens creates 400 checks. Adding 5 enemies and 5 more bullets doesn't sound much, but adds an extra 225 check! pseudocode can show the basic routine, and perhaps offer solutions for speeding up the routine:

```
// dont add this code to the game
PseudoOverlaPCheck(     short x0, short y0, short w0, short h0,
                        short x1, short x1, short w1, short h1)
      {
// if right of rect 1 is further left than left of rect 2
      if ((x0+w0) < x1) then not possible to overlap
// if bottom of rect 1 is further up than top of rect 2
      if ((y0+h0) < y1) then not possible to overlap
// if left of rect 1 is further right than right of rect 2
      if (x0 > (x1+w1)) then not possible to overlap
// if top  of rect 1 is further down than bottom of rect 2
      if (y0 > (y1+h1)) then not possible to overlap

      if (none of the four options above is TRUE) then overlapping
      }
```

Each of the four main checks is exclusive to the others. If one rectangle is too far to the left of the other one, then regardless of how the other checks result, the rectangles will not overlap. This leads to doing one check at a time, and only trying the others if the previous are successful, hopefully resulting in an average of doing around only two of the four necessary checks each time.

Implementing the routine

Passing eight parameters 650 times will take up a fair bit of time, so passing some sort of pointer to a collection of the data would be more useful. Fortunately the PSX supports a structure that is used quite regularly, called RECT. The four member variables are an x,y,w and h, so only two arguments will need to be passed. The new general overlapping routine looks like the following. This function is already included in "lib2d.h" so typing it into your program will cause an error, but is still shown below to allow greater understanding:

```
// function prototype for general bounding rectangle function
u_long CheckRectOverlap(RECT *rect1, RECT *rect2);

// function for general bounding rectangle overlap checks
u_long CheckRectOverlap(RECT *rect1, RECT *rect2)
      {
      u_long returnedValue = 0;
      if ((rect1->x+rect1->w) >= rect2->x)
```

```
            if ((rect1->y+rect1->h) >= rect2->y)
                  if (rect1->x <= (rect2->x+rect2->w))
                        if (rect1->y <= (rect2->y+rect2->h))
                              returnedValue = 1;
      return returnedValue;
      }
```

For the first investigation into the effectiveness of the routine, just the checks for enemy bullets against the player will be done. To do the checks, two instances of the RECT structure have to be created in the UpdateWorldGame() function. Declare these at the start:

```
// declare two instances of RECT struture for collision detection
RECT  rect1, rect2;
```

At the end of the same function, add the following code:

```
// only do any of the collision checking between player ship and enemy
// bullets if the player isn't flagged as invulnerable
if (invulnerable == 0)
      {
// setup rect1 with information regarding the player's position
      rect1.x = yourShip.x - 10;
      rect1.y = yourShip.y - 6;
      rect1.w = 20
      rect1.h = 12;
      }
```

Notice that the rectangle is not the full size of the ship sprite (40x23), but a significant amount smaller. This ties in with the philosophy of being 'nice' to the player, giving them a decent amount of lee-way regarding collisions that affect them. Until the game is tested there is no real way of knowing if this is too much or not, but will suffice for now. A quick decision on what to do if the ship does collide with a bullet is needed. A simple answer is move it back to the center of the screen, deduct a life (and check if the number of lives has run out) and set it invulnerable for two seconds again. Directly after the setting of rect1 from previous code (but still in the equality check (invulnerable == 0), add the actual check routine.

```
// every enemy bullet is the same width + height, so these dont change
rect2.w = rect2.h = 4;
// loop to check player against every active enemy bullet
for (counter=0; counter<25; counter++)
      {
// only do overlap checking if bullet is active
      if (enemyBullets[counter].state == 1)
            {
// set up top left coords of rect2 to those of the bullet
            rect2.x = enemyBullets[counter].image.x - 2;
            rect2.y = enemyBullets[counter].image.y - 2;
// if the bullet and player ship do overlap
            if (CheckRectOverlap(&rect1, &rect2))
                  {
// turn off that bullet
                  enemyBullets[counter].state = 0;
// move ship to center of screen
                  yourShip.x = 160; yourShip.y = 120;
// turn on invulnerability
                  invulnerable = 100;
// deduct a life, and if left than 0 lives left, set game flag to quit
                  if (livesLeft-- == 0) returnedValue = 1;
                  }
            }
      }
```

When compiled and run, you now have to dodge the bullets otherwise you turn green and move to the center of the screen, and one of the lives left indicators on the top left of the screen will disappear. When they are all gone and you die again, the game should jump back to the title screen. The game is all one sided at the moment though, you can be hit, but can’t shoot the enemies. Now to add the check for that. This will take the form of two nested loops. The first cycles through each active bullet of the players, and the second will cycle through every active enemy. Again the check will go in UpdateWorldGame(), but first you will need to declare another variable at the start of the function for the second loop:

```
// declare another loop counter variable
u_long counter2;
```

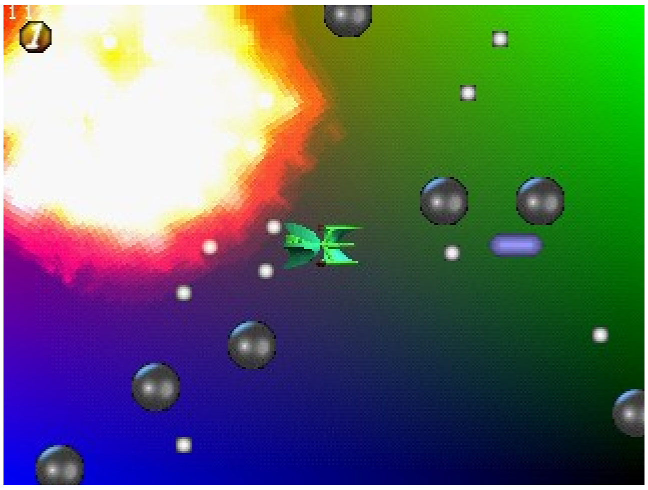
After the previous collision detection routines, add the following:

```
// set up the height + width of both rectangles, as these never change
// during checking. rect1 = player bullets, rect2 = enemy
rect1.w = 24; rect1.h = 12;
rect2.w = rect2.h = 24;

// outer loop cycles through active bullets
for (counter=0; counter<25; counter++)
      {
      if (yourBullets[counter].state == 1)
            {
// set top left corner of rect1 for collision checking
            rect1.x = yourBullets[counter].image.x – 12;
            rect1.y = yourBullets[counter].image.y – 6;

// second loop cycles through active enemies
            for (counter2=0; counter2<25; counter2++)
                  {
                  if (enemyShips[counter2].state > 0)
                        {
// set top left corner of rect2 for collision checking
                        rect2.x = enemyShips[counter2].image.x – 8;
                        rect2.y = enemyShips[counter2].image.y – 8;
// if the two rectangles do overlap, do the following -
                        if (CheckRectOverlap(&rect1, &rect2))
                              {
// turn the bullet off (unless you want them to carry on
                              yourBullets[counter].state = 0;
// turn off the enemy (definitely wanted!)
                              enemyShips[counter2].state = 0;
// set the counter2 so high to break from the second loop
```

```
                    counter2 = 26;
                    }
                }
            }
        }
    }
```

That should now allow you to shoot happily away at the enemies! Of course, there isn't very much that is satisfying about an enemy simply disappearing when shot, so its time to add the eye-candy for the game: the nict explosions and after that a nice scrolling background.

2D STEP 13: Adding Pretty Explosions



Now we can finally justify keeping a movement vector stored for each enemy, rather than just moving it within each of the separate ControlEnemy() functions. The idea of the explosions is that when a ship gets destroyed, then an explosion sprite will take over, and it will fade from full brightness to black, and that it will be drawn semitransparently. So far all that has happened to the explosion sprites is that they have had their texture image applied to them. Now we have to set them all to be drawn using method 1 (100% background + 100% sprite colour). In void main() we wrote a loop that cycled through each of the sprites, setting their information. After the line which sets the texture images for the explosion sprites, add the following:

```
// turn on semitransparent processing for explosion sprites
BitSet(explosions[counter].image.attribute, 1<<30);
// set them to use method 1 (100% + 100%)
BitSet(explosions[counter].image.attribute, 1<<28);
```

Now we need to do the same for enemies and bullets; that is create a function when they are created, and another one to handle the motion of them. The actual creation of an explosion is more like creating a player's bullets, rather than the release of bullets or enemies, because you do not want there to be 25 explosions on screen, and when you shoot a 26th enemy for their to be no explosion. Therefore we will simply cycle through the explosion sprites one at a time, and if by the time sprites are reused they are still active, then tough, they will have to move. When an explosion is created, the function will need to know where to start it, and what the movement vector of the enemy ship was when it was destroyed.

```
// function prototype for creating an explosion
void CreateExplosion(short startx, short starty, short xadd, short yadd);

// function for creating an explosion
void CreateExplosion(short startx, short starty, short xadd, short yadd)
      {
      static u_long explosionIndex = 0;

// set the colour counter to double brightness
```

```
      explosions[explosionIndex].colour = 255;
// set the start position of the explosion
      explosions[explosionIndex].image.x = startx;
      explosions[explosionIndex].image.y = starty;
// set the movement vector of the explosion
      explosions[explosionIndex].xadd = xadd;
      explosions[explosionIndex].yadd = yadd;
// set size to normal
      explosions[explosionIndex].scalex =
            explosions[explosionIndex].scaley = ONE;


// increment the index counter and loop at 25 ready for next explosion
      explosionIndex = (explosionIndex+1) % 25;
      }
```

Now to code the function that will update the movement vector of the explosion. Note that if you just wanted it to carry on moving in a straight line after creation, doing nothing, but a nice effect would be if the explosion got larger and rotated.:

```
// function prototype to alter movement of an explosion
void ControlExplosion(u_long index);

// function for control of explosion
void ControlExplosion(u_long index)
      {
// decrement the colour counter (which at 0 turns it off)
// going from 255 to 0 in steps of 5 means it will take one second for
// the explosion to fade out completely, which sounds ok
      explosions[index].colour -= 5;
// set the brightness regulation of sprite so it fades out to black
      explosions[index].image.r =
            explosions[index].image.r =
            explosions[index].image.r = explosions[index].colour;
// scale the explosion
      explosions[index].image.scalex =
            (explosions[index].image.scaley += 512);
// and why not have it rotating as well, to look nicer?
      explosions[index].image.rotate += 3*ONE;
      }
```

Note that as explosions turn themselves off fairly rapidly, it’s probably not worth having a check to see if they are off screen. Now to add the code to call these functions to the UpdateWorldGame() function. In the checks for a collision between an enemy and a player bullet, there are the lines:

```
// turn off the enemy (definitely wanted!)
enemyShips[counter2].state = 0;
```

After that, add this:

```
// create an explosion at the current enemy location
CreateExplosion(  enemyShips[counter2].image.x,
                  enemyShips[counter2].image.y,
                  enemyShips[counter2].xadd,
                  enemyShips[counter2].yadd);
```

and at the end of the UpdateWorldGame() function, add in the calls to ControlExplosion():

```
// loop to cycle through active explosions and update them
for (counter=0; counter<25; counter++)
      if (explosion[counter].colour > 0) ControlExplosion(counter);
```

If you now recompile the code you should get a nice explosion effect. Finally lets add a large explosion when the player ship gets hit. To do this we will simply use more than one of the explosion sprites moving around to let the player know something dramatically bad has happened. Within the UpdateWorldGame() function there is routine for checking for a player collision with an enemy bullet and it contains the following line:

```
// turn off that bullet
enemyBullet[counter].state = 0;
```

after that, add the following:

```
// create 9 explosions where the ship died
// the first eight move out in a circular pattern
CreateExplosion(yourShip.x, yourShip.y, -3, 0);
CreateExplosion(yourShip.x, yourShip.y, -2, -2);
CreateExplosion(yourShip.x, yourShip.y, 0, -3);
CreateExplosion(yourShip.x, yourShip.y, 2, -2);
CreateExplosion(yourShip.x, yourShip.y, 3, 0);
CreateExplosion(yourShip.x, yourShip.y, 2, 2);
CreateExplosion(yourShip.x, yourShip.y, 0, 3);
CreateExplosion(yourShip.x, yourShip.y, -2, 2);
// the last one stays in the middle of the ship location
CreateExplosion(yourShip.x, yourShip.y, 0, 0);
```

2D STEP 14: Sprite Animation

FASTTRACK

Variety of ways of effecting animation.
Swap one sprite for another one with a different texture image.
Call SetSpriteInfo() again, specifying a different texture image.
Draw all the animation frames into one large tim, tiling them, then using SetSpriteAnim() function to only draw one of the frames. These frames can be tiled either horizontally or vertically or both, but total tim must still not be bigger than 256x256 and only use one CLUT (if used at all)

```
void useinanyfunctionbeforeRenderWorld()
        {
        SetSpriteAnim(&mySprite, 1, 3, 40, 23);
        }
```

SetSpriteAnim()	alter which frame is shown on mySprite. This would use the 2nd frame from the left, and fourth from the top (as top left = 0,0). Each frame is 40x23 in size.

SLOWTRACK

Animation is a very important addition to a game, and there are plenty of different ways of doing it. Alternating the display on screen between frames in an animation is fairly simple, and which method you use depends on what you are trying to achieve.

Displaying different sprites

This is the simplest method. Assume you have 10 tims, each with a different animation frame on. You could use the code:

```
GsSPRITE    playerAnim[10];
u_long      playerAnimFrame = 0;
short       playerXpos = 160;
short       playerYpos = 120;

void main()
      {
      SetSpriteInfo(&playerAnim[0], ANIMONE_TIM, 0, 0);
      SetSpriteInfo(&playerAnim[1], ANIMTWO_TIM, 0, 0);
      .. all the way up to ..
      SetSpriteInfo(&playerAnim[10], ANIMTEN_TIM, 0, 0);
      }

void UpdateWorld()
      {
      playerAnimFrame = (playerAnimFrame+1) % 10;
      playerAnim[playerAnimFrame].x = playerXpos;
      playerAnim[playerAnimFrame].y = playerYpos;
      }

void RenderWorld()
      {
      DrawSprite(&playerAnim[playerAnimFrame], 0);
      }
```

Plus points of this method is each frame can bea different size, have a different CLUT, but you have to remember to update the position of the sprite currently in use. It will also be longer to set up, and requires a lot more memory, but overall it is fairly fast.

Setting one sprite to use a different texture

Here you repeatedly use the function SetSpriteInfo() to associate a different texture image to the sprite each frame, and would go something like this:

```
GsSPRITE    mySprite;
u_long      playerAnimFrame = 0;
short       playerXpos = 160;
short       playerYpos = 120;

void UpdateWorld()
```

```
      {
      playerAnimFrame = (playerAnimFrame+1) % 10;
      switch (playerAnimFrame)
            {
            case 0:     SetSpriteInfo(&mySprite,
                              ANIMONE_TIM, playerXpos, playerYpos); break;
            case 1:     SetSpriteInfo(&mySprite,
                              ANIMTWO_TIM, playerXpos, playerYpos); break;
      .. all the way up to ..
            case 10:    SetSpriteInfo(&mySprite,
                              ANIMTEN_TIM, playerXpos, playerYpos); break;
      }
```

All you have to do is keep drawing the same sprite. This method takes up less memory, is slightly more unwieldy to use than the first, but is also likely to be a bit slower, as you are retreiving information for the sprite each frame from main memory. With this each frame can also be a different size and use a different CLUT.

Putting Frames together into one TIM

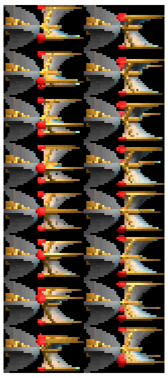
This method requires a bit of preparation, but is the method I will use for the game because it is the fastest method of all three. You have to include all the animation frames into one tim file, and this in itself causes problems, as you immediately become limited by the number of frames you can have, with the problem becoming more of an issue the larger each individual frame is, as the complete tim has to fit in one texture page (256x256). Also, each frame must be the same size, all the frames have to use the same CLUT (if using one at all) and they must be arranged in a grid fashion, either horizontally or vertically or both. To see what I mean, we can swap the old player graphic (one frame only) for the new animated version (16 frames). There is a file called ANIMSHIP.TIM which will replace YOURSHIP.TIM. Copy this file into the data directory, and then rewrite MEMTOOL.DAT to load animship.tim in place of yourship.tim. Run MEMTOOL.exe again, then you can go back into the directory with your code and make the one following change to your void main(). Replace

```
SetSpriteInfo(&yourShip, YOURSHIP_TIM, 160, 120);
```

with

```
SetSpriteInfo(&yourShip, ANIMSHIP_TIM, 160, 120);
```

If you now recompile and run the program, you will see the entire set of animation frames.


You will now create a variable within UpdateWorldGame() to control the animation frame of the ship, and then use a new function SetSpriteAnim() to set which of the possible frames it uses. So in UpdateWorldGame() add the following:

```
// declare a variable to control player ship animation
static u_long    yourShipAnim = 0;

// increment and loop the animation
yourShipAnim = (yourShipAnim+1) %16;
// set which frame to display from the entire texture image
SetSpriteAnim(&yourShip, yourShipAnim/8, yourShipAnim%8, 40, 23);
```

And that is all there is to it. The function first specifies which sprite to alter, then looks at the large texture image, and uses a grid reference to control which frame is displayed. (0,0) being the top left frame, but the extent in either direction purely depends on how you added the frames together. The third and fourth arguments are the width and height one Frame. More generally:

```
SetSpriteAnim(&spriteToAnimate,
                  animationValue / framesAcrossInCombinedTextureImage,
                  animationValue % framesDownInCombinedTextureImage,
                  widthOfOneFrame,
                  heightOfOneFrame);
```

This means all you have to worry about is incrementing the animation value as you see fit. Try running the program again now and you should see the front of the ship rotate.

2D STEP 15: Background Images and Bgedit

FASTTRACK

Create your cells.tim tile graphics file
Create your map.dat map data file using cells.tim and bgedit
Add them both to MEMTOOL.DAT and rerun MEMTOOL

```
void main()
      {
      InitialiseBackgroundMap();
      }
```

```
void UpdateWorld()
      {
      MoveMapTo(20, 20);
      MoveMapBy(16,16);
      }

RenderWorld()
      {
      DrawMap(BACK);
      }
```

InitialiseBackgroundMap()	Sets up the map ready for use
MoveMapTo()	Move top left of the screen to look at (20,20) (in pixel units, not cell units)
MoveMapBy()	Move top left of the screen relative to old position to look at (36,36) (pixel units)
Draw()	Draw the map with priority of BACK

SLOWTRACK

The final thing to add is to have some scrolling background passing by. The automatic background drawing routines are fast and useful for scrolling backgrounds, but can also be used for static screens. The way the background is drawn is by carving a larger texture image into smaller tiles of a regular size, and drawing the screen using these tiles as indicated by some level data. The files associated with the background map are:

- cells.tim. This contains all the graphics you want to make up your background with.
- map.dat. This is the actual map data, made up from an array which indexes the different graphics in cells.tim.

It is advised to create a new directory somewhere, and in it include the bgedit program, and the cells.tim file, and map.dat (if you have got one). Trying to run the program from your data directory will only confuse things. When you have finished your map editing, you can copy map.dat and cells.tim back into your data directory ready to be used by your program.

Creating your tiles

Cells.tim in particular requires greater examination before you can start creating your background. It is a compilation of all the tiles to be used in the background into one large image and it has to fulfil the following criteria:

1. It must be either 16 colour (4 bit CLUT) or 256 colour (16 bit CLUT). 15 bit true-colour is not supported with the background drawing. This means that all your tiles have to use the same palette.
2. Each tile can be only 16x16 pixels in size and lie on a 16 pixel boundary within the large tim file.
3. It must be 256x256 pixels in size. This allows a grid of 16 by 16 tiles, a maximum of 256. If you want to use less tiles than that you must still keep the cells.tim file 256x256 pixels large.
4. As the tim file is as large as the PSX can handle, then it must lie on the start of a texture page (see Step 3: Texture pages and texture image considerations). When converting your

bitmap into tim format, you should align the image horizontally on a multiple of 64, and vertically on a multiple of 256.

The biggest problem with background maps is that you limited to only one palette for all the shapes, but unless you are creating a detailed background then hopefully you can work around the problem quite easily.

Deciding on your map dimensions

It is important early on to decide what dimensions you are going to give to your background. A non-scrolling screen would 20x15 tiles, and for this game we will create a horizontally scrolling level. At the moment the game has no real objective other than to survive indefinitely, so we will create a background to suit that, by making it loop constantly until all three lives are lost. The map will be 15 tiles high and 60 tiles wide. That way you can draw two whole screens worth of background with the last 20 tiles wide being a copy of the first 20 to allow a seamless movement from the end of the map to the start. This is necessary, as the background maps don’t wrap around.

Setting up the background editor bgedit

Now that the map dimensions have been decided upon, there are a few changes to be made to the code of bgedit to accept these dimensions. Open the file main.c into the editor of your choice. Lines 25-27 contain the following:

```
// set the map dimensions here
#define NCELLW    (256)
#define NCELLH    (256)
```

This shows that the defualt map size is 256x256 tiles. Change these to the values above:

```
// set the map dimensions here
#define NCELLW    (60)
#define NCELLH    (15)
```

Recompile the program and you are now ready to start running bgedit.

Running and using bgedit

After making sure that cells.tim is in the same directory as bgedit you have a choice of two ways to run bgedit. The first is to run it and load in a map.dat file, and the other method is for those people who either don’t have a map.dat file, or want to start with a new one. To start with a map.dat file to be loaded, type

```
bgedit
```

at the dos prompt, otherwise type

```
bgedit clean
```

This will then load the editor and the map.dat file if specified. If you loaded the program without the map.dat then the map will be full of rubbish, and need clearing. A full list of controls is below:

Joypad control	Function
Start	Bring up main menu
Select	Toggle information shown in screen.
directional pad	Move the crosshair around.
cross	Fill the current cell under the crosshair with selected tile.
square	Pick up the tile that fills current cell.
circle	Show every tile that you can use, where you can highlight and select a new tile to use.
triangle	Undo the last cell change.
R1 + joypad	Scroll the map keeping crosshair in same cell.
R2 + joypad	Scroll the map keeping crosshair in same screen position.
L1	Forces immediate key repeat (otherwise there is 0.4 sec delay).
L2	Same as above but key repeat speed is 5 times faster.

Controls for using Bgedit

The main menu has several important options, which are highlighted using up and down, and selected using cross:

- Back. This will take you back to the editing screen.
- Save Map. This copies the currently edited map into main memory. You must do this before you quit if you want to be able to save the map to the PC.
- Load Map. Use this if you want to copy the map in main memory over the currently edited map. This can be used together with the Same Map option to create an undo feature.
- Clear Map. This will set all cells in the map to clear. If you started bgedit without loading in a map.dat and there is junk data in the map, use this option immediately.
- Quit. Takes you out of the program, ready for you to transfer the map information from the PSX to the PC. To do this you must have used the Save Map option already.

Once you have messed around with your map and got it looking like you want, then use the Save Map option from the menu screen, and then quit from the program. Never reset the PSX manually, otherwise the map information will be lost! What will happen is the PC will display the size of the file to be saved which will alter every time you change the map dimensions, as shown here:

```
map information size is 364. Press ESCAPE to quit from SIOCONS, then type
SAVEMAP 364
to create or update map.dat.
```

Now all that remains is to transfer the map.dat from the PSX to the PC. To do this, simply type what the PC has suggested. the <escape> to quit siocons, followed by:

```
savemap 364
```

at the Dos prompt and it will either create the file map.dat if it does not exist, or write over the old version if it does. You now have your map.dat file to be included into your own game, so copy it back to data directory with all your other files.

Incorporating the background map into the game

There are a few tricks to setting up the background data to work in the program, but fortunately these can all be placed in general purpose functions that require little knowledge about the make up of the cell arrays. First of all though, you have to load the two required files into the PSX. To do this, edit the MEMTOOL.DAT file in the data directory and add these two lines:

```
cells.tim
map.dat
```

Im assuming at the moment you are yusing the supplied cells.tim and map.dat so in the example below I can assumethe map is 60x15 cells in size. Upon running MEMTOOL again, these will have been included in the loading procedure for your program. In order to use the background map, there are a few odd things you have to do. All you have to do to set up the background is add the following to the start of the void main() function:

```
// prepare the background data arrays for drawing to the screen
// this will only work if cells.tim and map.dat have been uploaded
// to the PSX. Specify the map size (60,15)
InitialiseMap(60,15);
```

As long as you have correctly specified your map size, it is now ready to be used, and all you have to do is draw the map and specify which part of the map you are looking at. In UpdateWorldGame() add the following code:

```
// declare variable to track background position
static short      backgroundPos = 0;

// increment background position and loop
// the map is 60 wide, but the last 20 are copies of the first 20,
// so we wrap at 40 cells
backgroundPos = (backgroundPos+1) % (40*16);
// set the position of the viewing of the map
MoveMapTo(backgroundPos, 0);
```

All you have to add now is the drawing command, whose only argument is the priority of the background map, which usually will be set as BACK. In RenderWorldGame() add:

```
// draw the map
DrawMap(BACK);
```

And that's all there is to it. You now have your scrolling background. Another way of specifying which part of the map to look at is to use MoveMapBy() function, which has two arguments, an x offset and y offset from the current position. So constantly using

```
MoveMapBy(1,0);
```

would scroll the screen from right to left constantly.

More development

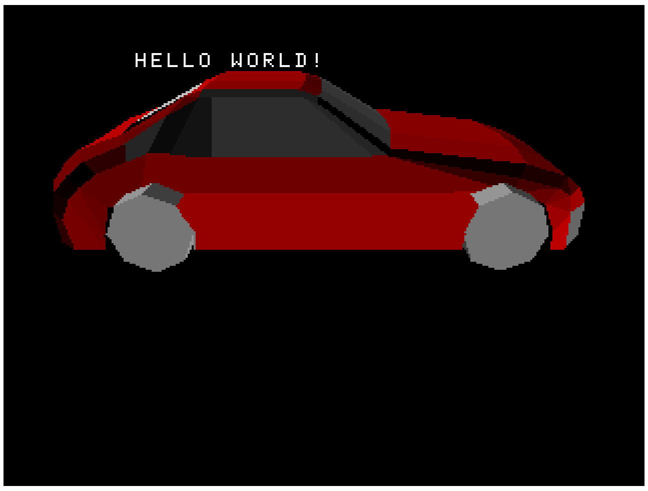
There are a wide variety of things that need changing in the game. Now it is in a fairly complete state, fine tuning is possible. Personally, one of the things I would do is reduce the number of enemy bullets on screen at once, as the game is far too hard! What about power-ups, or a score? Adding a title screen would be nice, and at the moment when you lose your last life the game quickly goes back to the title screen before you even get a chance to watch your own explosion. Different and more intelligent enemies are needed, and a better method for triggering the aliens as well.

With those previous 6 lines of text there are hours and hours of work. How you decide to write your own game will vary, but hopefully an appreciation of the work and designing that goes into even the smallest of games will allow a realistic approach to what is and isn't possible.

3D STEP 3. Load and view a 3D model.

FASTTRACK

In this step we fast forward into the world of 3D graphics and you will get to display a 3D object on the screen. The image below is the final result of this step.



This step covers a lot of information. In order to load and view a 3D model you must

1. Create a TMD model (car01.tmd is provided for this step)
2. Run MEMTOOL to automatically assign addresses.
3. #include addrs.h in your main file to load the address automatically assigned to car01.tmd by MEMTOOL.
4. Create an instance of the struct `PlayerStruct` to hold the PSX variables that are needed to manage the car's position and orientation in the world and link it into the PSX rendering system:

```
PlayerStruct TheCar;
```

5. Call function `InitialisePlayerModel` to initialise the structure giving a starting position to your model.

```
InitialisePlayerModel(&TheCar,  0, -200,0,(u_long*)CAR01_TMD);
```

The parameters we pass are the address of the car, it's initial x,y,z position and a pointer to the actual address of the TMD in memory. MEMTOOL automatically names the address CAR01_TMD.

6. Call the function `InitialiseLights()` to initialise two default lights to light the scene.

7. Create and instance of a PSX viewing struct to store our viewing parameters.

```
GsRVIEW2        ViewPoint;
```

8. Call function `InitialiseView` to set up the view.

```
InitialiseView(&ViewPoint, 250, 0, 1000, -500, 0, 0, 0, 0 );
```

The first parameter is the distance to the screen, the second rotation of the view around the z-axis. The next three parameters are the location of the viewer in space and the last three parameters specify where it is looking at.

9. In RenderWorld call the function DrawPlayer to draw the car on the screen.

```
DrawPlayer(&TheCar);
```

Here's the listing of main.c for step3.

```
/****************************************************************
       main.c
       ======
****************************************************************/

#include <stdio.h>
#include <libps.h>
#include "pad.h"
#include "addrs.h"
#include "step3.h"

// Create an instance of the PlayerStruct to hold our car
PlayerStruct TheCar;
// We need a variable to store the status of the joypad
u_long PADstatus=0;
// We need a viewing system for 3D graphics
GsRVIEW2        ViewPoint;
// This function deals with double buffering and drawing of 3D
// objects
void RenderWorld()
{
            RenderPrepare();
            DrawPlayer(&TheCar);
            //print your elegant message
            FntPrint("Hello World!\n");
            // force text output to the PSX screen
            FntFlush(-1);
            RenderFinish();
}

int main()
{
      // set up print-to-screen font, the parameters are where    // the
font is loaded into the frame buffer
      FntLoad(960, 256);
      //specify where to write on the PSX screen
      FntOpen(-96, -96, 192, 192, 0, 512);
      // initialise the joypad
      PadInit();
      // initialise graphics
      Initialise3DGraphics();
      // Setup view to view the car from the side
  InitialiseView(&ViewPoint, 250, 0, 1000, -500, 0, 0, 0, 0 );
```

```
      // initialise two default lights to light the scene
InitialiseLights();
      // The car's initial xyz is set to 0,-200,0
      InitialisePlayerModel(&TheCar,  0, -200,0,
(u_long*)CAR01_TMD);
      while(TRUE){
            PADstatus=ReadPad();
            if (PADstatus & PADselect) break;
            // render the car and hello world message
            RenderWorld();
            }
      // clean up
      ResetGraph(3);
      return 0;
}
```

SLOWTRACK

Loading a 3D object.

The PSX has it’s own proprietary format 3D object format - the tmd file. The good news is that you can convert standard dxf files which many packages such as 3D Studio and Autocad can export and you’ll also find loads of dxf files on the net (although they usually have much too many polygons to be useful). The not so good news is that its a bit of a tricky process doing the conversion and this is left to a later step. So in the words of every TV chef, here’s one I prepared earlier. In the models subdirectory of the PSX directory on the server are a couple of example dxf files, one of these, a model of a car, I have converted to a tmd and coloured in a rudimentary way. You can find the car01.tmd file in the examples directory on the server.

Assigning your model an address in memory.

The first thing you have to do in using a model is assign it an address in memory in to which it will be loaded. Use MEMTOOL to do this by

1. editing the file MEMTOOL.DAT in your data directory to contain the single line

```
car01.tmd
```

2. Run MEMTOOL, and it will automatically assign an address for you.
3. Check you have the correct makefile for this example from the `tutprogs` tutorial programs.
4. Include “addrs.h” in the program. Eg: your includes should look like:

```
#include <stdio.h>
#include <libps.h>
#include "pad.h"
#include "addrs.h"
```

The manual method is covered below but unnecessary if you used MEMTOOL. The address is specified in hexadecimal and you should start your addresses from hex address 80090000 which is where memory you can use starts from. As this is the first thing we want to load into memory will will assign it address 80090000. We have to edit our batch program ‘auto’ to load the model to this address. So edit your ‘auto’ file to the following:

```
local dload data\car01.tmd  80090000
local load main
go
```

(Note that MEMTOOL does this all for you). This assumes that the model `car01`.tmd is in the subdirectory `data\` of the current directory. You can now rerun your program and note that the feedback on the PC monitor indicates it has been loaded. Amongst the usual output the following text appears:

```
data\car01.tmd  address:80090000-80092393 size:002394  002394:
1sec.
```

Note that this tells you the amount of memory the model has used, which will be handy when you want to load more models (and textures and sounds) later. Having loaded the tmd file we need to indicate in our program what this address is so we can use it. Consequently our program associates a constant with the address:

```
#define CAR01_TMD (0x80090000)
```

(Note that MEMTOOL does this for you and the name used above is what MEMTOOL would call it). We define it as a constant so that if we change its memory address at a later date we only have to change the value of the constant. Starting the number with ‘`0x`’ tells the compiler to think of it as a hex number. If we change the memory address in either our program or the ‘auto’ file we must make sure that we change it in both or certain unhappiness will result.

File organisation.

In order to provide a fast track route through this material we hide as much as possible from the user in the files stepN.c and stepN.h. The idea is that you can use the function calls as described in the fast track section to get quick results without having to understand the underlying functions that are calling the PSX functions. As you advance though you will want to do things that the top level functions don’t cater for so a detailed description of underlying issues is given in the SLOWTRACK section to facilitate understanding and modification of lower level code.

Creating a player struct to hold your model.

Once again you will have to dip into the world of PSX datatypes in order to use your model in a program. In order to get the PSX to insert the model into the ordering tables and eventually draw it we need to associate two PSX datastructures with it. In our program we are going to call the car ‘the player’ because its going to be the bit of our virtual world which the player will manipulate. Thus we need to define a struct for our player (ie: car) which includes the two PSX structs. Our definition is as follows:

```
// Define a structure to hold a 3D object that will be rendered on screen
```

```
typedef struct
{
       GsDOBJ2          Object_Handler;
       GsCOORDINATE2    Object_Coord;
}PlayerStruct;
```

The player will need an instance of the `GsOBJ2` struct so we can handle our object. This structure is important now because we will associate our tmd model with this structure. The second structure `GsCOORDINATE2`, is important because it will contain our car's coordinate system, via this structure we will be able to change the position and orientation of the car in our virtual world - vital stuff huh? However we leave movement for a later step. Having defined our struct we create a (globally defined) instance of it:

```
// Create an instance of the structure to hold our car object
PlayerStruct TheCar;
```

Initialising the player struct.

We define a function called `InitialisePlayerModel` to initialise the car, we call it add model to player because at a later date we may wish to have more than one tmd file associated with an object: for example to produce an animated person walking we may wish to cycle through an number of tmd files with the limbs in different positions to give the impression of walking. The function is as follows:

```
// This function associates a model with our player
// datastructure
void InitialisePlayerModel(PlayerStruct *thePlayer, int nX, int nY, int nZ,
unsigned long *lModelAddress)
{
      //increment the pointer to past the model id.
      lModelAddress++;
      // map tmd data to its actual address
      GsMapModelingData(lModelAddress);
      // initialise the players coordinate system - set to be
      // that of the world
      GsInitCoordinate2(WORLD, &thePlayer->Object_Coord);
      // increment pointer twice more - to point to top of
      // model data
      lModelAddress++;  lModelAddress++;
      // link the model (tmd) with the players object handler
      GsLinkObject4((u_long)lModelAddress,
                          &thePlayer->Object_Handler,0);
      // Assign the coordinates of the object model to the
      // Object Handler
      thePlayer->Object_Handler.coord2 =
                &thePlayer->Object_Coord;
      // Set the initial position of the object
      thePlayer->Object_Coord.coord.t[0]=nX;   // X
      thePlayer->Object_Coord.coord.t[1]=nY;   // Y
      thePlayer->Object_Coord.coord.t[2]=nZ;   // Z
      // setting the players Object_Coord.flg to 0 indicates it
      // is to be
      // drawn
      thePlayer->Object_Coord.flg = 0;
}
```

This function is just a list of the curious stuff we have to do to initialise the PSX structs so our car can get into the PSX world. We pass the function a pointer to our player structure (ie: the car), an x, a y ,and a z which will be set to the cars initial position in the world, and a pointer to the car's address in memory so that the tmd file can be linked to the player. We can ignore the details, you could substitute a different tmd file for the car and this function would faithfully initialise it just the same.

However a short commentary on the function follows for the stout hearted. The first thing that happens is we increment the pointer to point past the ID of the tmd - why? because that the way the PSX likes it. The `GsMapModelingData` function then does some private internal memory mapping, the details of which we can ignore. The call to `GsInitCoordinate2` initialises the car coordinate system to agree with the world coordinate system ie: 0,0,0 is in the same place for both of them. On the next line the tmd memory address pointer is incremented twice to point to the start of the data wanted for the next function call – a bit odd eh! but do not fear as long as we remember to do it (or to use this function which does it!) all will be well. The call to `GsLinkObject4` on the next line links the tmd data to our player struct. We assign the two coordinate systems of the object and the object handler to the same coordinate system on the following line. We now do something we can understand which is on the next three lines set the start position of the car. Note that the origin of the car is the center of it's volume. Finally we set the players `gsObjectCoord.flg` to 0 tells the system that it has moved so that it will be recalculated and drawn. (In fact if you have lots of static objects, you can save some calculation time by never resetting the flags to zero, as the system stores an internal matrix which will be reused rather than having to be recalculated).
Phew. We are now ready to turn on the lights and pick up the camera.

Setting up lighting.

Setting up lighting is pretty straight forward, in fact its so easy we'll have two. A light is defined by the `GsF_LIGHT` struct which defines a light source by the direction in which it is pointing, and it's intensity expressed as a red, green, blue triple. Note these lights can be considered to be infinitely far away, so we don't specify their positions only the direction in which they are emitting light. This direction is represented by a vector whose components can be any size (ie: they are not normalised values). We store our two lights in an array defined as follows:

```
// This is an array of structures for the lights
GsF_LIGHT       LightSource[2];
```

We define a function to initialise all our lights as follows:

```
void InitialiseLights()
{
InitialiseALight(&LightSource[0], 0, -100, -100, -100, 255,255,255);
InitialiseALight(&LightSource[1], 1, 1000, 1000, 1000, 255,255,255);
       GsSetAmbient(0,0,0);
       GsSetLightMode(0);
}
```

The call to `GsSetAmbient` sets the ambient (background) light to zero. The call to `GsSetLightMode` sets the lighting mode to 0 which means no fogging, setting this to 1 would turn fogging on (which you should try later). which makes your world foggy. The first two lines of the function set each light individually by calling `InitialiseALight` as shown below, and passes the light's position and r,g,b intensity value.

```
void InitialiseALight(GsF_LIGHT *LightSource, int nLight, int nX, int nY,
int nZ,
            int nRed, int nGreen, int nBlue)
{
       // Set the direction in which the light travels
       LightSource->vx = nX;LightSource->vy = nY;
      LightSource->vz = nZ;
       // Set the colour
       LightSource->r = nRed; LightSource->g = nGreen;
      LightSource->b = nBlue;
       // Activate light
       GsSetFlatLight(nLight, LightSource);
}
```

Note that the above function also calls `GsSetFlatLight` for each light, which activates ('turns on') the light.

Setting up a viewing system.

Setting up the view is also easy, the PSX struct for defining viewing systems called `GsRVIEW2`, and it has the obvious variables for setting the camera position and look at point. We define a view in our program:

```
GsRVIEW2        ViewPoint;
```

and initialise it with a function `InitialiseView` shown below:

```
void InitialiseView(GsRVIEW2 *view, int nProjDist, int nRZ,
                                      int nVPX, int nVPY, int nVPZ,
                                      int nVRX, int nVRY, int nVRZ)
{
       // This is the distance between the eye
       // and the imaginary projection screen
       GsSetProjection(nProjDist);
       // Set the eye position or center of projection
       view->vpx = nVPX; view->vpy = nVPY;  view->vpz = nVPZ;
       // Set the look at position
       view->vrx = nVRX; view->vry = nVRY; view->vrz = nVRZ;
       // Set which way is up
       view->rz=-nRZ;
  // Set the origin of the coord system in this case the world
       view->super = WORLD; //&TheCar.Object_Coord ;
       // Activate view
      GsSetRefView2(view);
}
```

The first line sets the projection distance, the distance between the viewpoint and the projection plane (the size of the projection plane is set elsewhere by `GsInitGraph()`) and effectively gives the field of view, larger values give a smaller field of view and viva versa. The next few lines of code are obvious until we come to the line:

```
view->super = WORLD;
```

This line indicates that the view system uses the co-ordinate system of the world, ie: 0,0,0 is in the same place for the view system as it is for the world (and incidentally for the car as we set it in `InitialisePlayerModel).` The last line activates the viewing system.

The viewing system is set up in the way that you might imagine with the positive Z axis going into the screen and the X and Y axes aligned in the plane of the screen. Whilst the direction of the positive X axis is to the right as you face the screen the positive Y axis is actually down rather than up! Why? - beats me, but as long as you know what the alignment of the axes is you should have no problem.

I know that the car is oriented lengthways along the positive Z axis, and I also know the car is about 500 units long, and since I have played with the projection distance I know that setting the position of the view (eye position) at 1000, -500, 0 will give a nice side view of the car when we eventually get to see it.

OK, almost there but first we have to look at how to draw the car...

Drawing the player.

Drawing the player is achieved in the function `DrawPlayer` which basically sets up matrices that are going to be needed by the PSX to draw the car on the screen, including one for local screen world and screen co-ordinates and one for lighting calculations. When that's done it is sent to the ordering table using the GsSortObject4 function. We should cover this in greater detail later but note for the moment the same steps or recipe need to be applied whenever you want to have your object drawn on screen and as usual you can treat it as a black box function, you pass it a player struct and an ordering table, and it will draw the player for you. The function looks like this:

```
void DrawPlayer(PlayerStruct *thePlayer)
{
   MATRIX  tmpls, tmplw;
   //Get the local world and screen coordinates, needed for
   //light calculations
   GsGetLws(thePlayer->Object_Handler.coord2, &tmplw, &tmpls);
   // Set the resulting light source matrix
   GsSetLightMatrix(&tmplw);
   // Set the local screen matrix for the GTE (so it works
   // out perspective etc)
   GsSetLsMatrix(&tmpls);
   // Send Object To Ordering Table
   GsSortObject4(&thePlayer->Object_Handler,
      &OTable_Header[CurrentBuffer],4,
      (u_long*)getScratchAddr(0));
}
```

The final step is to actually call the `DrawPlayer` function from the appropriate place inside the `RenderWorld` function (eg: between the existing calls to `RenderPrepare()` and `RenderFinish()`as follows:

```
void RenderWorld()
{
            RenderPrepare();
            DrawPlayer(&TheCar);
            //print your elegant message
            FntPrint("Hello World!\n");
            // force text output to the PSX screen
      FntFlush(-1);
            RenderFinish();
}
```

Update the main function.

You will now need to update your main function to call the new 3D initialisation functions thus:

```
int main()
{
      // set up print-to-screen font, the parameters are where the font is
      // loaded into the frame buffer
      FntLoad(960, 256);
      //specify where to write on the PSX screen
      FntOpen(-96, -96, 192, 192, 0, 512);
      // initialise the joypad
      PadInit();
      // initialise graphics
      Initialise3DGraphics();
      // Setup view to view the car from the side
      InitialiseView(&ViewPoint, 250,0,1000,-500,0,0,0,0);
      InitialiseLights();
      // The car's initial xyz is set to 0,-200,0
      InitialisePlayerModel(&TheCar,  0, -200,0,
            (u_long*)CAR01_TMD);
      while(TRUE){
            PADstatus=ReadPad();
            if (PADstatus & PADselect) break;
            // render the car and hello world message
            RenderWorld();
            }
      // clean up
      ResetGraph(3);
      return 0;
}
```

Ok, thats it, when you run the program you should now see your shiny new red, porche-like car on the screen, lit by your lights and viewed by your viewing system...

You can now try loads of things, see the effect of changing the projection distance, try setting up different views of the car, try loading up a different tmd file or get two on the screen at the
same time, move the lights around etc etc.

3D STEP 4. Transforming a 3D model.

FASTTRACK

In this step we will translate and rotate the car. This step introduces two new functions that both operate on a PSX structure called `GsCOORDINATE2` which defines the objects position and orientation. The `PlayerStruct` has one of these which it calls `Object_Coord`.

If you want to move an object just call function `MoveModel` passing it your objects `Object_Coord` structure and the amount you want it to move by in each of the x,y, and z directions. Eg: the following moves the car 32 units in the z direction:

```
MoveModel(&TheCar.Object_Coord,0,0,32);
```

A first rotation function called RotateModel is introduced. In PSX terms 4096 equals 360 degrees so to rotate by 180 degrees the angle is specified as 2048, for 90 degrees the angle is 1024 etc. To rotate your model call the function passing it your objects and an angle for rotation around each of the x,y and z axes. Eg to rotate around the x axis by 32 units (almost 3 degrees):

```
RotateModel(&TheCar.Object_Coord,32,0,0);
```

It is recommended that you read the ‘Understanding the matrix struct’ and specification of angles for rotation sections in this step.

SLOWTRACK

Making the model move.

Making the car move (translate) is very easy all we have to do is increment the cars position variable. The cars position is is stored in `Object_Coord`.t which is defined as long t[3] - ie one parameter for each of x,y, and z. Define a function:

```
void MoveModel (GsCOORDINATE2 *Object_Coord, int nX, int nY, int nZ)
```

Which passes a pointer to the car and an x,y, and z value for the position to move to. You could call it:

```
MoveModel(&TheCar.Object_Coord,0,0,32);
```

to make the car move to 0,0,32 or 32 steps in the positive Z direction. Note that the last line of this very short function should set the car’s `flg` variable to make sure it is redrawn eg:

```
        Object_Coord ->flg = 0;
```

Make sure your code works to move the car in all the three directions, to any point in space.

Now modify this function so that the car continuously moves in a particular direction when a particular button is pressed.

Right, translation is sorted, but before we can make the player rotate we are going to have to learn a bit more about PSX structs and functions.

Understanding the MATRIX struct.

The players `Object_Coord.coord` variable is actually of type `MATRIX`, which is great because you know all about matrices. However these matrices are different to the ones you are used to, but with good reason. As you are used to using matrices for doing 3D transformations you should immediately expect them to be 4 rows by 4 columns wide. This is not the case, a `MATRIX` is actually defined as:

```
typedef struct {
short m[3][3];
long  t[3];
} MATRIX;
```

This structure is optimised for games, it is big enough to hold the coefficients for rotation or scaling, but cannot hold those for translation (as these occur in the last row or column depending on the convention you use for homogenous matrix representation). The reason is that you typically want to rotations of your object around your objects centre in games, and this is what the PSX implements. Unless you go out of your way to do otherwise all rotations applied to the object are around the object’s centre not the WORLD origin, which is handy. So what about translation? As you should be able to surmise from your last function, translation is specified relative to the WORLD origin, even though rotation is around the objects centre. Furthermore translation is specified by a vector separate from the matrix itself. This formulation seems a little odd at first as the MATRIX actually contains only a 3 x 3 matrix and also contains the t[3] array, but it has obvious advantages - the different transformations are specified relative to different origins which are intuitive for games, and to perform a translation you don’t have to muliply a whole matrix, to name but two.

The specification of angles for rotation.

So what about rotation? As discussed in the accompanying lecture notes, most high performance systems try to use integer arithmetic instead of floating point where ever possible. The PSX is one of these and so often where you would naturally specify a floating point number the PSX wants an integer instead. The specification of angles is one such case. The PSX actually divides 360 degrees of angle into 4096 integer steps. Firstly this implies an obvious translation formula between the two systems: to scale from degrees to PSX steps we apply the formula steps = (4096/360) * degrees. Note that this can lead to slight problems, whilst if we want to rotate say 180 degrees we correctly specify a whole integer result of 2048 steps, if we wanted to move by 1 degree we would calculate we need to move by 11.378 steps - as this is not an integer it would be rounded and we get a slight error, which if it cumulates could be a problem. How can you minimise this?

Rotating an object.

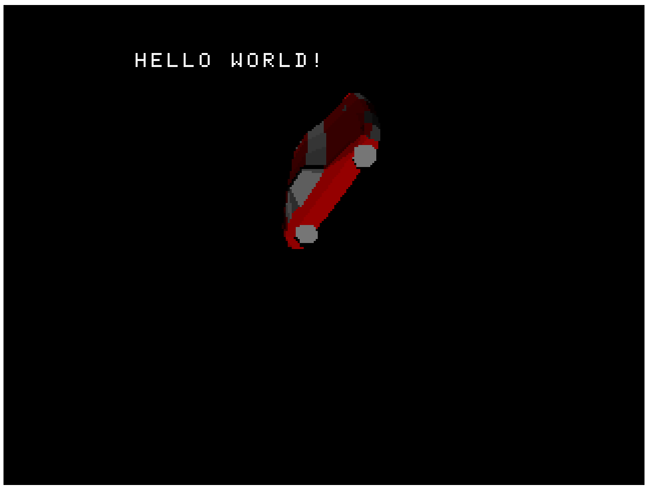
We have identified the the car's `Object_Coord.coord` as being the matrix we need to get the rotation coefficients into but to set up the matrix we have to use the PSX `RotMatrix` function to set a the rotation and the `MulMatrix0` function to concatenate the objects original matrix with the rotation matrix. Note that there is an error in the manual, it specifies the order of arguments to the `RotMatrix` function as `RotMatrix(MATRIX *m ,SVECTOR *)` when it should really be `RotMatrix(SVECTOR *r, MATRIX *m).` We define a function `RotateModel` which we want to call passing a pointer to the car's matrix and rotation amounts (specified in PSX steps rather than degrees, see above) for each of x, y, and z. eg:

```
        RotateModel (&theCar.Object_Coord,  0, -64, 0 );
```

In order to use the rotation and matrix concatenation functions we need to have a local temporary matrix and a local temporary `SVECTOR`. The rotation function look as follows:

```
void RotateModel (GsCOORDINATE2 * Object_Coord, int nRX, int nRY, int nRZ )
{
      MATRIX matTmp;
      SVECTOR svRotate;

      // This is the vector describing the rotation
      svRotate.vx = nRX;
      svRotate.vy = nRY;
      svRotate.vz = nRZ;
      // RotMatrix sets up the matrix coefficients for rotation
      RotMatrix(&svRotate, &matTmp);
      // Concatenate the existing objects matrix with the
     // rotation matrix
      MulMatrix0(&Object_Coord->coord, &matTmp,
           & Object_Coord->coord);
      // set the flag to redraw the object
      Object_Coord->flg = 0;
}
```

3D STEP 5. Precision problems.



FASTTRACK

The `RotateModel` introduced in the last step can produce distortion of your model as shown above because precision errors accumulate on the car's rotation matrix. This step introduces two new versions of `RotateModel`: `RotateModelVector` and `RotateModelMatrix`,. You are recommended to use `RotateModelMatrix` as a general rotation function and `RotateModelVector` for situations where you only need to rotate around one axis eg: for a driving game in a flat world.

Rotation using a vector.

Step5.h defines a new player struct which includes a vector to hold the rotation angles around each of the axes:

```
typedef struct
{   SVECTOR         Rotation;
       GsDOBJ2         Object_Handler;
       GsCOORDINATE2   Object_Coord;
}PlayerStruct1;
```

So if you define `TheCar` to be of type `PlayerStruct1` then when you call `RotateModelVector` you pass the objects vector as well eg:

```
RotateModelVector(&TheCar.Object_Coord,&TheCar.Rotation,32,0,0);
```

To use this new struct you must also use functions `InitialisePlayerModel1` and `DrawPlayer1` which accept the modified struct `PlayerStruct1.`

Rotation using a conditioned matrix.

`RotateModelMatrix` gets around the problem of the rotation matrix becoming distorted by calling a function `AdjustCoordinate2` which makes sure the matrix is kept in good shape and therefore avoids scaling problems. You use it in the same way as the orginal function in your program eg:

```
RotateModelMatrix(&TheCar.Object_Coord,-32,0,0);
```

SLOWTRACK

Precision problems already.

Those of you who rotated your car for a very long time will have noted that it seems to slowly get further away. In fact what happened is that your car has been scaled due to precision problems and the fact the players coord matrix cumulatively stores all your rotational errors. Recall that whatever the direction of rotation, some rotation coefficients fall along the matrix diagonal that is used for specifying scaling. It follows that cumulative precision problems may result in a scaling or shearing after some time which is in fact the case. If you put in very large angles to accentuate the problem you will find that various shearings and scalings up and down can be produced by different numbers. There are various solutions to this problem some of which are discussed later. The solution we will adopt here, suited to a driving simulation on the plane is not to let errors accumulate in the player's rotation matrix. Instead we augment the player struct to include a short vector to hold the rotation parameters:

```
// Define a structure to hold a 3D object that will be rendered on screen
typedef struct
{   SVECTOR         Rotation;
       GsDOBJ2       Object_Handler;
       GsCOORDINATE2 Object_Coord;
}PlayerStruct1;
```

We need to redefine the player initialisation to accept the new struct and initialise the rotation vector:

```
void InitialisePlayer1(PlayerStruct *thePlayer, int nX, int nY, int nZ,
      unsigned long *lModelAddress)
{
      // initialise the players rotation vector to 0
      thePlayer->Rotation.vx=0;thePlayer->Rotation.vy=0;
      thePlayer->Rotation.vz=0;
      // initialise other player variables and link in tmd
      InitialisePlayer(thePlayer,  0, 0,0, CAR01_TMD);
}
```

which we call from the main function:

```
InitialisePlayer1(&theCar,  0, -200,0, (long*)CAR_MEM_ADDR);
```

The RotateModelVector rotate function.

We can now rewrite our RotateModel function to use the players rotation vector. When we call this function we need to reset the players rotation matrix to the identity matrix before setting it to the rotation matrix as calculated for the players rotation vector. In fact there is a PSX library function to do this called GsIDMATRIX. However we wish to keep the players translation vector intact. One way to do this is to copy the translation vector to temporary variables, reset the matrix with GsIDMATRIX, assign the matrix to the rotation matrix, and then finally assign the translation vector to the temporary variables. However it will involve fewer assignments if we insert our own code to reset the players rotation matrix to the identity matrix without interfering with its translation vector. The following two lines of code will do this:

```
Object_Coord->coord.m[0][0]=Object_Coord->coord.m[1][1]=
Object_Coord->coord.m[2][2]=ONE;
Object_Coord->coord.m[0][1]=Object_Coord->coord.m[0][2]=
Object_Coord->coord.m[1][0]=Object_Coord->coord.m[1][2]=
Object_Coord->coord.m[2][0]=Object_Coord->coord.m[2][1]=0;
```

Recall the identity matrix is filled with 0 except along the diagonal which should contain 1s. Given the PSX's use of fixed point of arthmetic 1 in this context turns out to be 4096 - which is the value of the constant ONE. We're now ready to redefine the `RotateModelVector` function:

```
void RotateModelVector (GsCOORDINATE2 *Object_Coord,SVECTOR *rotateVector,
int nRX, int nRY, int nRZ )
{
MATRIX matTmp;

  // reset matrix to the identity matrix
  Object_Coord->coord.m[0][0]=Object_Coord->coord.m[1][1]=
  Object_Coord->coord.m[2][2]=ONE;
  Object_Coord->coord.m[0][1]=Object_Coord->coord.m[0][2]=
  Object_Coord->coord.m[1][0]=Object_Coord->coord.m[1][2]=
  Object_Coord->coord.m[2][0]=Object_Coord->coord.m[2][1]=0;
// Add the new rotation factors into the players rotation
// vector and then set them to the remainder of division by
// ONE (4096)
  rotateVector->vx = (rotateVector->vx+nRX)%ONE;
  rotateVector->vy = (rotateVector->vy+nRY)%ONE;
  rotateVector->vz = (rotateVector->vz+nRZ)%ONE;
  // RotMatrix sets up the matrix coefficients for rotation
  RotMatrix(rotateVector, &matTmp);
  // Concatenate the existing objects matrix with the
  // rotation matrix
  MulMatrix0(&Object_Coord->coord, &matTmp,
        &Object_Coord->coord);
  // set the flag to redraw the object
  Object_Coord->flg = 0;
}
```

Note that when we set the rotation variables we increment them with the new amount and set them to the remainder of division by 4096 as 4096 = 360 degrees to keep each rotation factor below 360 degrees.

If you play with this function you will see that we have got rid of the scaling problems that came with our earlier version, but only works properly for rotation around a single axis. An example of using this function can be found in the step5a directory.

The RotateModelMatrix function

An alternative method to deal with the matrix distortion is to 'condition' the matrix to make sure it doesn't get out of shape. Any row or column in the rotation matrix should always be a unit vector to avoid shearing or scaling.
ie:

```
Sqrt(x*x+y*y+z*z) =1
```

… and row or column vectors should be orthogonal (at 90 degrees) to each other - which you should see is clearly true for the identity matrix:

```
1 0 0
0 1 0
0 0 1
```

… and in fact rotation matrices.

One approach to dealing with the problem is simply to normalise the columns and rows of the matrix. To normalise a vector you divide its components by its magnitude or length ie:

```
Length= sqrt(x*x+y*y+z*z)
```

And then
```
x'=x/length,y'=y/length,z'=z/length
```

The problem is that the sqrt function is extremely slow so you need to use approximation methods or lookup tables. The best route seems to be to use a method from our Japanese friends which I think is based on the Gram-Schmidt method for dealing with matrix drift, the function is called AdjustCoordinate2.

Thus we define `RotateModelMatrix` as follows:

```
void RotateModelMatrix (GsCOORDINATE2 *Object_Coord, int nRX, int nRY, int
nRZ )
{
      MATRIX matTmp;
      SVECTOR svRotate;

      // This is the vector describing the rotation
      svRotate.vx = nRX;
      svRotate.vy = nRY;
      svRotate.vz = nRZ;
      // RotMatrix sets up the matrix coefficients for rotation
      RotMatrix(&svRotate, &matTmp);
      // Concatenate the existing objects matrix with the
     // rotation matrix
      MulMatrix0(&Object_Coord->coord, &matTmp,
                    &Object_Coord->coord);
      // set the flag to redraw the object
      AdjustCoordinate2(Object_Coord);
      Object_Coord->flg = 0;
}
```

An example of using this function can be found in step 5b.

3D STEP 6. Odds and sods

FASTTRACK

This steps covers three miscellaneous topics. It deals with creating a separate function to read the joypad, the concept of hierarchical co-ordinate systems, and how to work out your framerate. There is no slowtrack for this step.

A joypad function.

Create a function to process the joypad input as follows:

```
void ProcessUserInput()
{
      PADstatus=ReadPad();
      if(PADstatus & PADselect)GameRunning=0;
      if(PADstatus & PADLleft){
```

```
                FntPrint( "Left Arrow: Rotate Left\n");
                RotateModelMatrix (&TheCar.Object_Coord, 0, -64, 0 );
        }
        if(PADstatus & PADLright){
                FntPrint ( "Right Arrow: Rotate Right\n");
                RotateModelMatrix (&TheCar.Object_Coord,  0, 64, 0 );
        }
        if(PADstatus & PADstart){
                FntPrint ( "PAD start\n" );
        }
        if(PADstatus & PADcross){
                FntPrint ( "PAD cross: Move Forward\n");
                MoveModel(&TheCar.Object_Coord,0,0,32);
        }
        if (PADstatus & PADsquare){
                FntPrint ( "PAD square Move Backward\n");
                     MoveModel(&TheCar.Object_Coord,0,0,-32);

         }
        if (PADstatus & PADR1){
              FntPrint ( "PAD padR1:STATIC_VIEW\n");
              InitialiseStaticView(&ViewPoint, 250, 0, 0, -500,
                    -1000, 0, 0, 0 );
        }
        if (PADstatus & PADR2)
        {
              FntPrint ( "PAD padR2: TRACKER_VIEW\n");
              InitialiseTrackerView(&ViewPoint, 250, 0, 0, -1000,
                    -1000, 0,0,0,&TheCar.Object_Coord);
    }
}
```

and then change the main function to call your new joypad function:

```
int main()
{
      // set up print-to-screen font, the parameters are where
      // the font is loaded into the frame buffer
      FntLoad(960, 256);
      //specify where to write on the PSX screen
      FntOpen(-96, -96, 192, 192, 0, 512);
      // initialise the joypad
      PadInit();
      // initialise graphics
      Initialise3DGraphics();
      // Setup view to view the car from the side
      InitialiseStaticView(&ViewPoint, 250, 0, 1000, -500, 0,
 0, 0, 0 );
      InitialiseLights();printf("1\n");
      // The car's initial xyz is set to 0,-200,0
      InitialisePlayerModel(&TheCar,  0, -200,0,
 (u_long*)CAR01_TMD);
      while(GameRunning){
            ProcessUserInput();
            // render the car and hello world message
            RenderWorld();
            }
      // clean up
      ResetGraph(3);
      return 0;
}
```

Note that you will have to define the variable `GameRunning` and give it a default value of TRUE.

Hierarchical co-ordinate systems.

A nice feature of the PSX is the implementation of hierarchical co-ordinate system. A co-ordinate system can use any other object's co-ordinate system as its origin (by setting the first objects `super` variable to point to the second object). This means for example that the moon can be made to orbit the earth by referencing the earth as `super`, whilst the earth orbits the sun by referencing the sun as `super,` or a hand moves automatically when the upper arm is moved. We can give a 'in car' view or an 'over the shoulder' view extremely easily by simply changing the `super` variable of the view system to point to that of the car. eg. Id Object_Coord was a pointer to the cars Object_Coord:

```
       ViewPoint->super = Object_Coord;
```

Before playing about with a new view create a function to print out on the PSX console various parameters such as the cars position, and the camera position. Then to produce an over the should view make two versions of the `InitialiseView` function one for the static view you currently have and one for a view tied to car but say 1000 units behind and above the car. Why don't you utilise a couple of those unused buttons to flip between views like this:

```
      if (PADstatus & PADR1){
            FntPrint ( "PAD padR1:STATIC_VIEW\n");
            InitialiseStaticView(&ViewPoint,
                  250, 0, 0, -500,-1000, 0, 0, 0 );
      }
  if (PADstatus & PADR2)
      {
            FntPrint ( "PAD padR2: TRACKER_VIEW\n");
            InitialiseTrackerView(&ViewPoint,
                  250, 0, 0, -1000, -1000, 0,0,0,&TheCar.Object_Coord);
  }
```

To do this the simplest way is to copy your original `InitialiseView` function to make these two new functions and just set the super field appropriately in each. A further issue is that whenever the viewpoint is changed GsSetRefView2(&`ViewPoint`); needs to be called so add this line as the first line of after `RenderPrepare` in `RenderWorld` or the tracker view won't be updated. (This is obviously not ver efficient how would you change it?) Obviously the tracker view can only be detected at the moment by looking at the cameras position variable or switching between views, it gets more interesting when you add a bit of scenery...

Approximating the frame rate

As we add more code and models to our program it will start to slow down so one thing we will want to do is get some estimate of how long it is taking our program to do a cycle of calculations between each frame displayed on the screen. It turns out that the VSync function which waits for vertical synchronisation (ie: until one screen has been drawn) can return the time that has elapsed since the last time it was called. The number returned is in units of horizontal synchronisation (ie: how long it takes to draw a line). If the number returned is less that the horizontal resolution of the screen, for whatever mode we happen to be in, then we are keeping up the maximum frame rate which is great. When the number returned starts to exceed the horizontal resolution the frame rate starts to go down. Whilst some degradation of frame rate can be tolerated it depends on the application as what amount is acceptable, ultimately the graphics get jerky and the game slows down.

To approximate the frame rate we will display the VSync - VSync interval and print it out on the screen, replacing our embarrassing hello world message. We need a variable to store the number:

```
// a variable to track the vsync interval
u_long VerticalSync=0;
```

Note that this is defined as u_long or unsigned long . This is a general PSX programming tip: we should use unsigned longs where ever possible as they happen to be the same width as the register and are processed quickly compared to say a short which would involve the register being padded with 0s to contain the number. We use the `VerticalSync` variable in the RenderWorld function:

```
            DrawPlayer(&TheCar);
            // wait for V_BLANK interrupt
            VerticalSync=VSync(0);
            // print the vSync interval
            FntPrint("VSync Interval: %d.\n",VerticalSync);
            // force text output to the PSX screen
            FntFlush(-1);
```

Note that the vsync interval increases if the model is larger on the screen, during rotation and translation, and even more during simultaneous rotation and translation.

3D STEP 7. Translating in the correct direction

FASTTRACK

In this step you learn how to get the car to move in the direction that it is pointing. To do this the PlayerStruct is augmented with a vector that describes the direction in which it is pointing and this is now set in RotateModelMatrix. A new function is introduced called Advance model that moves the model forward or backwards depending on the sign of the argument passed to it. Eg: to move the car 32 units in the direction it is facing:

```
AdvanceModel(&TheCar.Object_Coord,&TheCar.Direction,32);
```

Making the car move in the direction it is pointing

Our current MoveModel function allows us to move the car to any position in space but what we really want to do is move the car in the direction that its pointing. There are a number of different ways to implement both the rotation of the car and translation in the direction that it's pointing. However we consider only one here.

We know that the car originally points in the direction of the z axis, which we could represent as a unit vector by (0,0,1). We also know all the rotations that the car has been subject to, as these we are storing in the car's matrix. Consequently if we can set up a matrix for rotation and plug in the original orientation of the car we should get the direction it is currently pointing. This then is what we will do, but we need to consider once again the implications of having integer matrix operations. As 360 degrees are represented by 4096 steps we need to consider how we represent the start vector. If we actually used (0,0,1) we would get very poor results due to the integer nature of matrix operations on the PSX. Consequently our unit vector will be expressed as (0,0,4096) in PSX steps.

The effect of this is that we have to divide our final result by 4096 to ensure that the car is translated by the original number of units, and it will be best to do this division as the last step in the computation.

It is useful for a number of reasons to store the objects direction vector. Thus we enhance the `PlayerStruct` by giving it a direction vector:

```
typedef struct
{   VECTOR          Direction;
       GsDOBJ2         Object_Handler;
       GsCOORDINATE2   Object_Coord;
}PlayerStruct;
```

As the rotation function is where the orientation of the car is changed we should set the direction vector in this function. Thus the function changes to:

```
void RotateModelMatrix (GsCOORDINATE2 *Object_Coord,VECTOR
*currentDirection, int nRX, int nRY, int nRZ )
{
      MATRIX matTmp;
      SVECTOR svRotate;
      VECTOR startVector;
      // This is the vector describing the rotation
      svRotate.vx = nRX;
      svRotate.vy = nRY;
      svRotate.vz = nRZ;
      // RotMatrix sets up the matrix coefficients for rotation
      RotMatrix(&svRotate, &matTmp);
      // Concatenate the existing objects matrix with the
        // rotation matrix
      MulMatrix0(&Object_Coord->coord, &matTmp,&Object_Coord->coord);
      // condition the matrix
      AdjustCoordinate2(Object_Coord);
      // set up the vector corresponding to the starting orientation
      startVector.vx = 0;
      startVector.vy = 0;
      startVector.vz = ONE;
      // apply the objects matrix to the start vector to give the
      //current direction
      ApplyMatrixLV(&Object_Coord->coord, &startVector, currentDirection);
      Object_Coord->flg = 0;
}
```

Then we are ready to define the function AdvanceModel:

```
void AdvanceModel (GsCOORDINATE2 *gsObjectCoord, VECTOR *currentDirection,
int nD)
{
       if(nD!=0){ // avoid divide by 0 error
            nD = ONE/nD;

            gsObjectCoord->coord.t[0] += currentDirection->vx/ nD;
            gsObjectCoord->coord.t[1] += currentDirection->vy/ nD;
            gsObjectCoord->coord.t[2] += currentDirection->vz/ nD;
            gsObjectCoord->flg = 0;
       }
}
```

This function introduces us to one new PSX function `ApplyMatrixLV` which takes a `VECTOR` and a `MATRIX,` multiplies them together, and stores the result in a second `LVECTOR`. We can call the function as follows:

```
      if(PADstatus & PADcross){
              FntPrint ( "PAD cross: Move Forward\n");
              AdvanceModel(&TheCar.Object_Coord,&TheCar.Direction,64);
      }
      if (PADstatus & PADsquare){
              FntPrint ( "PAD square Move Backward\n");
              AdvanceModel(&TheCar.Object_Coord,&TheCar.Direction,-64);
       }
```

While we’re about it lets insert some code to reset the car to its original position and orientation:

```
if(PADstatus & PADstart){
              FntPrint ( "PAD start\n" );
              TheCar.Object_Coord.coord=GsIDMATRIX;
              TheCar.Direction.vx=0;TheCar.Direction.vy=0;
  TheCar.Direction.vy=ONE;
              TheCar.Object_Coord.flg=0;
      }
```

Note that a consequence of our setup is that is that we will get strange results if we use very small translation amounts. Try using translation amounts of 1, 2, and 4. What is the explanation for the resulting behaviour? How could you correct it?

Now we can make our car move in the direction it is pointing whatever that direction is, and we can rotate the car in any direction. This of course is not very realistic - the car leaps to a constant speed and can rotate on the spot, which is impossible in real life, to be more realistic we would have to model things like acceleration, friction and gravity. The question we need to answer is: is it good enough for our game? For the moment lets say yeah, and get onto texture mapping.

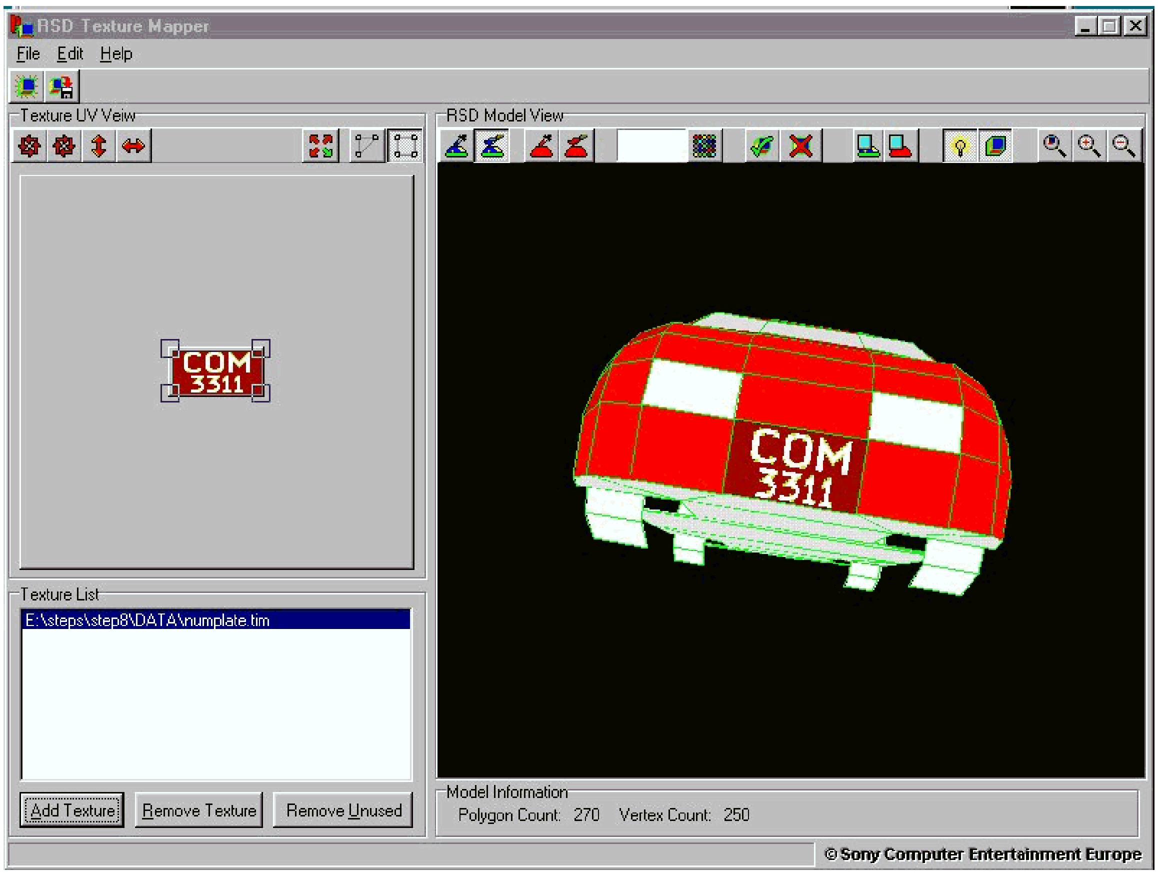
3D STEP 8. RSDtool, texture mapping and colouring

FASTTRACK

In this step we introduce a handy utility from SCEE for colouring and texturing (after a fashion) your 3D models. The basic steps for texturing a 3D model are:

1. Create you TIM file(s).
2. Load your RSD model into RSDtool and colour it, texture map it and save it.
3. Run TIMtool as necessary to position your textures in the framebuffer and save them.
4. Edit MEMTOOL.DAT to add in your new TIMs and TMD files
5. Run MEMTOOL and then RSD2TMD.
6. Make sure that your main function calls `LoadTextures()` in its initialisation section.
7. Use the models in your program.

RSDTOOL



E:\steps\step8\DATA\numplate.tim

Model Information
Polygon Count: 270 Vertex Count: 250

RSDTOOL only runs under Win95/NT.

First load your model into the tool by choosing File|Load RSDmodel The left hand side of the screen deals with textures while the right hand side of the screen shows your model. You can rotate your model by pressing the right mouse button down and moving the mouse. The buttons on the toolbar in the RSD Model View window all tell you what they do when you place the cursor over them, take a moment to learn about each of these buttons does. You paint individual polygons with a colour or apply a texture to them by clicking the appropriate button and left clicking on a polygon.

You can add and remove textures with buttons on the lower left of the screen in the Texture UV View window. The file names of currently used textures are shown above, and above that the currently selected texture. Add the texture numplate.tim from the step8\data directory. Paint the texture onto the object on the appropriate polygons to act as numberplates on the front and back of the model. Paint the back lights of the car from white to orange. Now save your model by choosing File|Save RSD Model.

The next step is to edit MEMTOOL.DAT to:

```
car01.tmd
numplate.tim
```

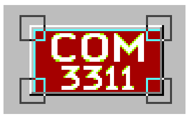
And then run MEMTOOL and RSD2TMD in the data directory.

As this is the first image used in your program you have to call the function `LoadTextures()`, make this function call the first line of your main program.

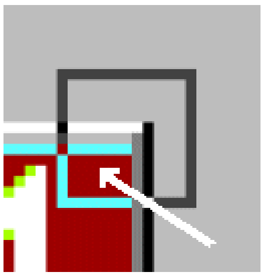
Now just recompile your program by typing `make clean`, and the making it. You should see the texturing and colour changes on your model.

Texturing more than one polygon.

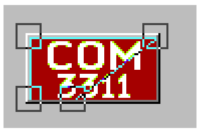
Obviously you can use the same image to texture more than one polygon using the methods above, but the problem is that the whole of the texture With a bit of fiddling around you can use RSDtool to spread the texture over more a few polygons. It's not a very easy feature to use however, and is only manageable for texturinf a small number of polygons with a single texture. Run RSDtool and load the car model again. Click on numplate.tim to display this texture in the texture window. Note that the corners of the texture have grey boxes surrounding them and are joined by a blue line (which is hard to see in black and white):



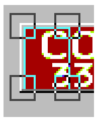
These handles can be used to map a subregion of the texture to the currently selected polygon. These handles can be moved by clicking and dragging with the mouse button held down. However beware, the area that you need to click in is not just anywhere in the grey box marking the handle but in the area that is the intersection of the image and the handles grey box. This is very frustrating if you don't know this information and annoying even if you do because this area can be very small when you have acute angles between the handles. In the image below this region is indicated by the white arrow.



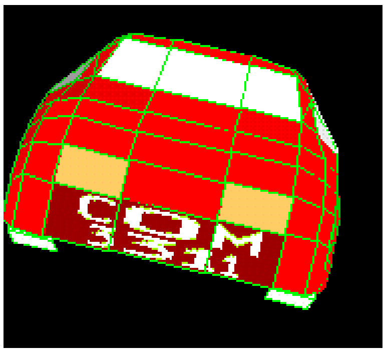
Apply the texture to the polygon to the left of the one already textured, and get it to the correct orientation by rotation if necessary. Now try moving the lower left handle as shown below. You should notice that the area between the four handles is now what is mapped to your polygon.



To try out this technique try to map the texture once over the three polygons at the back of the car. For example the mapping for the polygon to the left of the one originally textured should be something like:

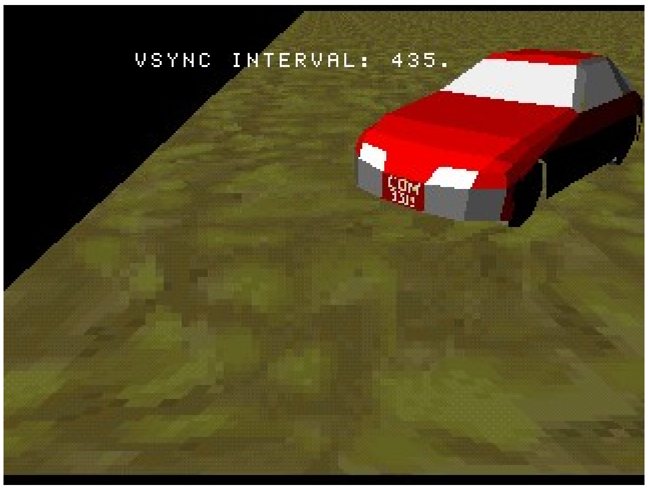


And the final result should be something like this:



3D STEP 9. Building a ground plane to drive on.

FASTTRACK



A lot of behind the scenes changes are made in this step to create a plane to drive on. The plane is built out of an array of squares which are each a bit larger than the car. The world map is specified by an 15x15 array of chars:

```
char worldGroundData[GROUNDMAX_Z][GROUNDMAX_X] ={
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
};
```

Wherever a ‘1’ appears in the array a yellow textured square will be placed. A ‘0’ is reserved to signify that nothing appears at that array position. By mixing ‘1’s and ‘0’s you can create different patterns of squares on the plane. From the fasttrack point of view you anly have to edit the data structure above to create your pattern. A world layer datatype is defined

```
WorldLayerStruct WorldGround;
```

Along with a function to draw it:

```
void DrawWorld(WorldLayerStruct *world, int precision);
```

and this function is called from RenderWorld in order to draw the plane:

```
void RenderWorld()
{
            RenderPrepare();
            DrawPlayer(&TheCar);
            DrawWorld(&WorldGround,WORLD_GROUND_PRECISION);
            // wait for V_BLANK interrupt
            VerticalSync=VSync(0);
            // print the vSync interval
            FntPrint("VSync Interval: %d.\n",VerticalSync);
            // force text output to the PSX screen
            FntFlush(-1);
            RenderFinish();
}
```

Note that `WORLD_GROUND_PRECISION` is predefined in step9.h. Also the program is now very slow because we are drawing so many squares.

SLOWTRACK

Deciding how to represent the world and interaction with it is a design decision that should be justified by what you wish to achieve and is thus application specific. There are a number of commonly used approaches to building a world for a game to take place in. They are application specific and affect your program design in terms of thinking about collision detection and clipping the world and other topics. What approach you take depends on your game and what exactly you are trying to model.

1. The arena approach usually defines one small area for the action to take place in, as typically found for fighting games.
2. The grid approach defines a larger area in which the player can move freely, found in free driving type games.
3. The track approach defines tracks along which the player can move but can’t move off.
4. 3D platform games can incorporate the previous approaches but introduce the idea of different height levels.

Of course all these types of approaches can and often are mixed up in different games, and other more specialised techniques may be needed. You need to decide early on what kind of approach you intend to take.

It’s time to add some ground to drive around on. A simple and general way to do this would is to be make an array of quadrilaterals (four sided polygons) and lay them out as a grid to form a ground plane. In order to do this we will have to develop a datastructure to represent the world and go through all the steps we went through for the player,

- develop 3D models and possibly textures to model world objects
- define a datastructure to model the world objects
- load up their tmd and tims
- initialise the world datastructures
- and draw the world

Developing a ground quadrilateral.

We decide that the size of one quadrilateral will be 1200 by 1200 units. The file square1.dxf is a one sided quad that was modelled in 3D studio. It was converted with a PSX utility called DFX2RSD . Two tools DFX2RSD and DFX2RSDW which have similar functionality run under DOS or Windows respectively and can be found in PSX\bin subdirectory. These tools have many flags and options which include things like reducing the number of polygons in your model, setting transparency, setting whether the model should be triangulated or quadrangulated, are polygons to be two or one sided etc. The dxf format is widely supported by many modelling packages but doesn't contain texture mapping information. If you have access to 3Dstudio then there is a tool called 3DS2RSD (described in this document in the 3Dstudio section) that can convert from the 3DS mesh format to RSD supposedly preserving texture mapping information.

To convert the square1.dxf file we we use she DOS DFX2RSD utility.

```
dxf2rsd -o square1.rsd -quad2 20 -back  square1.dxf
```

Following some experimentation it was found that the above converts the file in the way wanted. The -o square1.rsd part of the line specifies that the output filename should be square1.rsd. The -quad2 20 specifies that adjacent pairs of triangles should be amalgamated into quadrilaterals if the angle between their surface normals is less that 20 degrees which turns our quadrilateral defined by two triangles into a single quadrilateral.

Note that there are 21 different flags that can be passed to DFX2RSD, and we will not go through them all here. Your best approach in dealing with converting other models (if you don't use 3Dstudio and 3DS2RSD) is to follow the steps to get the model onto the screen and play about with the various parameters, whilst consulting the User Guide which documents this program, until you get the results you want. Sometimes you have to revert to the modeller to resolve problems (eg: once when using a cone in a model it always resulted in a tmd with a point from the cone's apex way off the scene, the solution was to model the cone with a tapered box instead).

Now use RSDTool to texture the quad with the image ground.tim found in the data subdirectory.

Building a plane to drive around on.

The quadrilateral defined above with its associated texture mapping constitutes a building block to build a road to drive around on. If we can replicate the quad around the scene we should be able to build a ground plane.

The model struct.

The definition of the world is different from that of the player in that the world will hold many objects rather than just one. However there is a degree of commonality between a world object and the player object. They both need object handlers and coordinate systems so we will abstract out the common bits into a new `ModelStruct` structure.

```
typedef struct
{
  u_long            Subdivision;
  GsDOBJ2           Object_Handler;
  GsCOORDINATE2     Object_Coord;
} ModelStruct;
```

This means the definition of PlayerStruct changes:

```
 typedef struct
{      ModelStruct            Model;
       VECTOR           Direction;
}PlayerStruct;
```

Making these changes means that we have to slightly modify the functions `InitialisePlayerModel` and `DrawPlayer` and change the calls to the advance and rotate functions. (Look at the file step9.c and step9.h to see the exact changes).

We can now define a structure for the ground. The simple design adopted here is to define layers for the world, we decide that we will have a ground layer to define the ground, and an overlay layer that we use in a later step to place objects on the ground. For a layer we will need a map to say what objects are where in the world, which is defined in the next section. The definition of the a world layer is different from that of the player in that the world will hold many objects rather than just one. Consequently we define arrays of relevant data structures. As the ground will lie in the XZ plane we define constants (in step9.h) in each direction that will be used to specify the array sizes:

```
#define GROUNDMAX_X 15
#define GROUNDMAX_Z 15
```

Thus we can define (in step9.h) a `WorldLayerStruct` structure:

```
typedef struct
{
       ModelStruct            Model[GROUNDMAX_Z][GROUNDMAX_X];
       char                         ModelMap[GROUNDMAX_Z][GROUNDMAX_X];
} WorldLayerStruct;
```

Creating a world map.

In the `WorldLayerStruct` we defined an array of characters `ModelMap` to represent this world. If we interpret each character as a different world object that gives us 255 different world objects in any layer, which should be ample. If you were designing serious game with many levels then it makes sense to build a level or layer editor to graphically lay out levels.

The approach adopted here is to define an char array that can be initialised at definition, and then copy this into the `ModelMap` array. This approach means the array is fairly easy to edit by hand. Thus we define:

```
char worldGroundData[GROUNDMAX_Z][GROUNDMAX_X] ={
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
};
```

We plan that wherever there is the character 1 in the array we will place a square.tmd, so our road will be a square one, right around the perimeter of the grid. We define a function to copy this map:

```
void InitialiseMaps()
{int tmpz,tmpx;
   for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ){
      for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
       WorldGround.ModelMap[tmpz][tmpx]=worldGroundData[tmpz][tmpx];
      }
   }
}
```

Initialising and drawing the world.

We now define a generic function, for initialising models that can be used by any struct that contains a variable of type `ModelStruct`. The following function is just a generalisation of `InitialisePlayer` function that copes with the new structure for models.

```
void InitialiseModel(ModelStruct *theModel, int nX, int nY, int nZ,
      unsigned long *lModelAddress)
{

      //increment the pointer to move past the model id. (weird huh?)
      lModelAddress++;
      // map tmd data to its actual address
      GsMapModelingData(lModelAddress);
      // Initialise the players coordinate system - set to be that of
      // the world
      GsInitCoordinate2(WORLD, &theModel->Object_Coord);
      // increment pointer twice more - to point to top of model data
      lModelAddress++;  lModelAddress++;
      // link the model (tmd) with the players object handler
      GsLinkObject4((u_long)lModelAddress,
            &theModel->Object_Handler,0);
      // Assign the coordinates of the object model to the Object
      // Handler
      theModel->Object_Handler.coord2 = &theModel->Object_Coord;
      // Set polygon subdivision flag to lowest possible value
      theModel->Object_Handler.attribute = (1<<9);
      // Set the Initialiseial position of the object
      theModel->Object_Coord.coord.t[0]=nX;   // X
      theModel->Object_Coord.coord.t[1]=nY;   // Y
      theModel->Object_Coord.coord.t[2]=nZ;   // Z
      // setting the models flg to 0 to indicate it is to be drawn
      theModel->Object_Coord.flg = 0;
}
```

A function is now needed to interpret the world map and link and build up the grid of squares into a plane. In order to go from array addresses to working out the actual z and x values to use to place squares in the world we need to know the size of a square so we define (in step9.h) a variable that is both the width and height of a square:

```
#define GROUNDSIZE  1200
```

The world can then be initialised by stepping through the array and putting a square wherever a ‘1’ is found

```
void InitialiseWorld ()
void InitialiseWorld ()
{
int tmpx,tmpz;
char c;
   // initialise world ground maps
    InitialiseMaps();
// then for each element of the worldGroundData array if we find a 1
// then place an instance of square.tmd at the appropriate position
// in the world.
   for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ){
      for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
         c= WorldGround.ModelMap[tmpz][tmpx];
         if(c == '1')  {
            InitialiseModel(&WorldGround.Model[tmpz][tmpx],
             (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                (u_long *)SQUARE_TMD);
          }
      }
   }
}
```

Next we define a generic function to draw model based on the ModelStruct that is a simple generalisation of drawplayer. The function is different though in that we pass an extra parameter to it called `precision`. This variable is used in the call to `GsSortObject4` and it specifies the precision with which models are inserted into the ordering tables. If we give a value of 1 for the ground and 2 for the car the objects look the same but their polygons will not intersect correctly in the ordering table. This is a trick used to stop the ground polygons overlapping with the car polygon, an effect that can be see in some PSX demos such as survival – your object is just translating over the ground when part of it is obscured by a ground polygon. Presumably this occurs when the polygons share the same location in the ordering table. Thus we define:

```
void DrawModel(ModelStruct *theModel, GsOT *OTable_Header, int precision)
{
       MATRIX  tmpls, tmplw;
       //Get the local world and screen coordinates, needed for light
       //calculations
       GsGetLws(theModel->Object_Handler.coord2, &tmplw, &tmpls);
       // Set the resulting light source matrix
       GsSetLightMatrix(&tmplw);
       // Set the local screen matrix for the GTE (so it works out
       //perspective etc)
       GsSetLsMatrix(&tmpls);
       // Send Object To Ordering Table
       GsSortObject4( &theModel->Object_Handler,
         OTable_Header,precision,(u_long *)getScratchAddr(0));
}
```

Then we need another function to loop through the world arrays drawing drawing the squares as necessary. Note that we reserve the symbol '0' to signify that there is no object at a ModelMap array address.

```
void DrawWorld(WorldLayerStruct *world,   int precision)
{
int tmpz,tmpx;

   for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ) {
      for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
         if(world->ModelMap[tmpz][tmpx]!='0'){
            DrawModel(&world->Model[tmpz][tmpx],
                    &OTable_Header[CurrentBuffer],precision);
         }
      }
   }
}
```

Finally if we call the above function from render world we will get our ground plane to drive around on.

```
void RenderWorld()
{
            RenderPrepare();
            DrawPlayer(&TheCar);
            DrawWorld(&WorldGround,WORLD_GROUND_PRECISION);
            // wait for V_BLANK interrupt
            VerticalSync=VSync(0);
            // print the vSync interval
            FntPrint("VSync Interval: %d.\n",VerticalSync);
            // force text output to the PSX screen
            FntFlush(-1);
            RenderFinish();
}
```

Note that the precision variables are defined in step9.h:

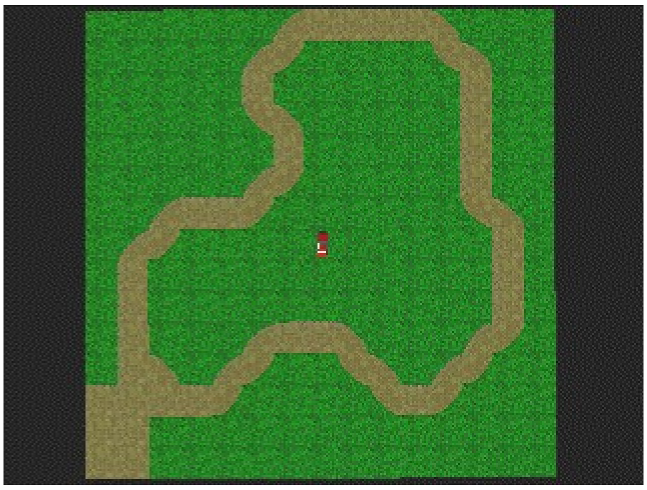
```
#define PLAYER_PRECISION 2
#define WORLD_GROUND_PRECISION   1
```

You can now drive your car around the ground, but you will notice that the vsync interval is over 300 and the action is now slowed down. This is due to the fact that the program is drawing the whole world even though you can't see it, which is a problem dealt with in the next step.

3D STEP 10 Creating a track.

FASTTRACK

This section involves creating new textures in order to create a track to drive around on as the distant view from above shows in the image below. The fasttrack approach is just to be able to edit the worldGroundData array to change the track .



In this example there are six types of textured square, which are shown at the beginning of the SLOWTRACK section overleaf. In the worldGroundArray they are signified by different letters. Following the order of the images in the picture overleaf the letters are '5','g','1','2','3', and '4'.

SLOWTRACK.

The stages you go through in this procedure are:

1. Make the texture images for the extra squares you are going to use for your track.

2. Open a new project in TIMtool, import all the images in, position them and their CLUTS and save the project and the modified TIM image files.

3. Copy square1.dxf to the new dxf filename you want (eg:grass.dxf) for each new square you want in the scene.
4. For each new square convert to rsd with DXF2RSD eg:

```
dxf2rsd -o ground3.rsd -quad2 20 -back  ground3.dxf
```

5. For each new square convert from RSD to TMD eg:

```
Rsdlink –v –o grass.tmd   grass.rsd
```

6. Run RSDtool and texture each new square with its texture.

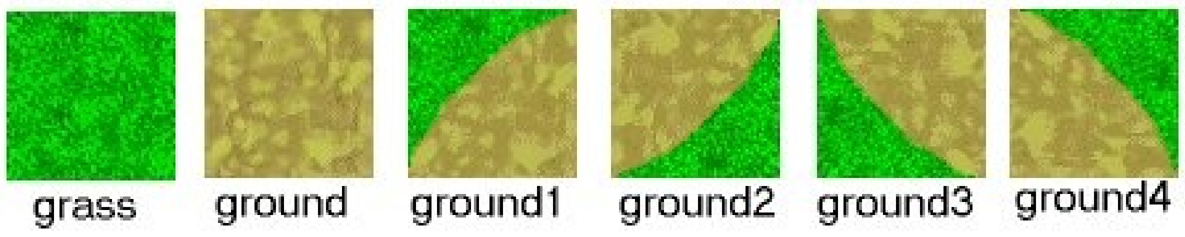
7. Edit memtool.dat to include all your new textures and tmd files. Then run MEMTOOL and the RSD2TMD batch file.

8. Assign characters that are to signify each of the new squares in your program and insert an initialise statement in to the loop in InitialiseWorld function to initialise each new square eg:

```
if(c == '5')  {
   InitialiseModel(&WorldGround.Model[tmpz][tmpx],
      (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),(u_long *)GRASS_TMD);
}
```

9. Edit the worldGroundData array inserting the appropriate numbers to get the track layout you want.

The slow track involves going through some or all the stages of this step. So far you only



have the one square (`square1.tmd`) which is textured by `ground.tim`. If you made the five new TIM files shown with ground.tim below then you will have sufficient textures to make the track shown at the start of this step.

Note that the textures ground2-4 are just rotations of ground1.

Run TIMtool and first add the existing textures you are using (`ground.tim` and `numplate.tim`). Then import all the new texture files and arrange them into the same texture page. Then autosort the CLUTS and save you project. Don't forget to also save the modified TIM files via the menu choice: `TIM Options|Save Modified TIM's`. Your textures are now ready to use.

In order to display these textures they each have to be associated with their own copy of the original square1 file. To do this you need to copy the original `square1.dxf` file for each new square as named above and convert them with DXF2RSD. Note the result of running DXF2RSD on a file is that four new files are created: eg: `grass.grp, grass.mat`, `grass.ply` and `grass.rsd`. These four files together specify the object in the RSD format, and as they make internal references to each other you cannot just individually copy these files to a new root file name, you have to copy the DXF file instead and then convert it to RSD . We use the same flags to DXF2RSD as used for the original conversion. Eg to convert `ground3.dxf` we use:

```
dxf2rsd -o ground3.rsd -quad2 20 -back  ground3.dxf
```

Now you have to texture each of your new squares using RSDtool remembering to save each one after texturing.

Your files are now read to be convert finally to TMD format. (MEMTOOL should really do this for you but doesn't at the time of writing. Eg: to covert all five new files:

```
rsdlink -v -o  ground1.tmd ground1.rsd
rsdlink -v -o ground2.tmd ground2.rsd
rsdlink -v -o  ground3.tmd ground3.rsd
rsdlink -v -o  ground4.tmd ground4.rsd
rsdlink -v -o grass.tmd grass.rsd
```

The next step is to get MEMTOOL assigning memory addresses for etc by editing the memtool.dat file in the data directory to:

```
car01.tmd
numplate.tim
square.tmd
ground.tim
ground1.tmd
ground1.tim
ground2.tmd
ground2.tim
ground3.tmd
ground3.tim
ground4.tmd
ground4.tim
grass.tmd
grass.tim
```

At this point you start editing your program to use the new square. You have to edit the InitilaiseWorld function to initilaise each new square in the world array. Thus you need to asssign a character to each square to be able to specify it in the map. The following assignment was used in this example:

```
void InitialiseWorld ()
{
int tmpx,tmpz;
      char c;
      // initialise world ground maps
       InitialiseMaps();
      // then  place the appropriate square for each element of the map
   for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ){
      for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
         c= WorldGround.ModelMap[tmpz][tmpx];
            if(c == 'g')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                  (u_long *)SQUARE_TMD);
            }
            if(c == '1')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                   (u_long *)GROUND1_TMD);
            }
            if(c == '2')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                  (u_long *)GROUND2_TMD);
            }
            if(c == '3')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                   (u_long *)GROUND3_TMD);
            }
            if(c == '4')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                   (u_long *)GROUND4_TMD);
            }
            if(c == '5')  {
               InitialiseModel(&WorldGround.Model[tmpz][tmpx],
                  (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
                  (u_long *)GRASS_TMD);
            }
         }
      }
}
```

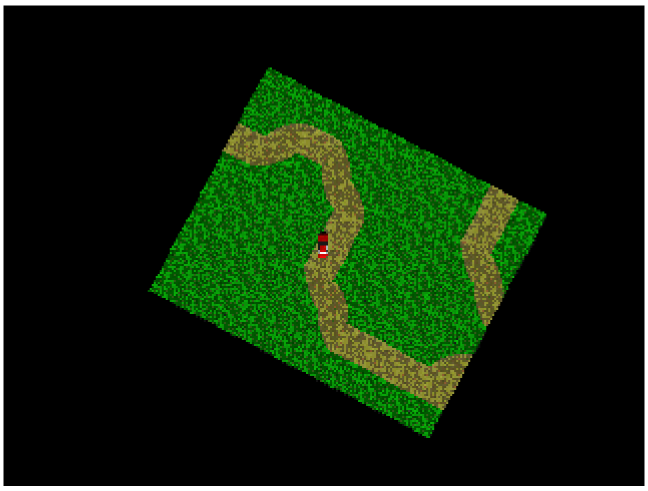
Finally you can now specify your track by changing the entries in the map array `WorldGroundData` . The size of the array is pretty small only 15x15 squares, which is a pretty boring size. However you will soon discover that in terms of editing it’s big enough as there are of course 225 entries that may need changing. This will probably cause you to consider that it would be e good idea to create a nice GUI map map editor, which is generally the case for programs like this, as a large amount of time can be spent in level design.

The worldGroundData array is shown with the valuse used for this example. Note that the orientation is rotated by 90 degrees in as compared to the 'over the shoulder view' you get when starting the program. The car is oriented in the first square looking to the left. This orientation is also indicated by the rectangle of square1's which are shown in the image as being in the bottom left hand corner of the grid, but are shown in the arrary below rotated ny 90 degree clockwise to appear in the upper left on the grid.

```
char worldGroundData[GROUNDMAX_Z][GROUNDMAX_X] ={
{'g','g','g','5','5','5','5','5','5','5','5','5','5','5','5'},
{'g','g','g','g','g','g','g','1','5','5','5','5','5','5','5'},
{'5','5','g','4','5','5','5','2','1','5','5','5','5','5','5'},
{'5','5','g','5','5','5','5','5','g','5','5','5','5','5','5'},
{'5','5','2','1','5','5','5','5','g','5','5','5','5','5','5'},
{'5','5','5','2','1','5','5','5','2','1','5','3','g','1','5'},
{'5','5','5','5','g','5','5','5','5','2','g','4','5','2','1'},
{'5','5','5','5','g','5','5','5','5','5','5','5','5','5','g'},
{'5','5','5','3','4','5','5','5','5','5','5','5','5','5','g'},
{'5','5','3','4','5','5','5','5','5','5','5','5','5','5','g'},
{'5','5','g','5','5','5','5','5','5','5','5','5','5','5','g'},
{'5','5','2','1','5','5','5','5','5','5','5','5','5','3','4'},
{'5','5','5','2','1','5','5','5','3','g','g','g','g','4','5'},
{'5','5','5','5','2','g','g','g','4','5','5','5','5','5','5'},
{'5','5','5','5','5','5','5','5','5','5','5','5','5','5','5'},
};
```

3D STEP 11. Clipping the world.

FASTTRACK



The rendering speed we have now is way above the recommended VSync interval of 256 and the biggest contributor to this is the amount of the world we are processing and/or drawing, ie: all of it. To reduce this and increase the speed we have to draw less of the world and we are really interested to only draw the part that we can see.

The approach taken is simple and inefficient, we will draw a rectangle of textured squares surrounding the car. The clipping widths for the box are defined in step11.h as:

```
#define CLIPXWIDTH 5
#define CLIPZWIDTH 5
```

and then to perform clipping around the car all you have to do is call a new clipping function in the main loop before you render the world:

```
…
while(GameRunning){
      ProcessUserInput();
      ClipWorld(TheCar.Model.Object_Coord.coord.t[0],
            TheCar.Model.Object_Coord.coord.t[2]);
      RenderWorld();
}
…
```

SLOWTRACK

Clipping considerations.

We need to control what parts of the world are drawn by clipping out the bits that we can't see so than no processing is associated with these bits. In natural viewing systems the visible world is essentially a infinite cone whose apex lies in the eye so we would like to have a (wide angle) conical viewing system. As the world modelled here is a plane we only need a triangular view ie: we want to only deal with those squares that lie in a equilateral triangle whose apex is the viewing point. The problem is how do we calculate it quickly as it changes as the viewpoint changes. One cell based (room like) approach is to render all of the current (small enough) 'room' and change to the next 'room' when the player goes over some threshold (eg: door) and then just display that 'room'.

In a track based game clipping simplifies to just drawing as many as possible of the upcoming track segments.

In the free roving game we are left with some approximation of the conical viewing system. Calculating which world elements fall in this view is likely to be expensive although there are no doubt many game specific solutions to this such as storing masks of array entries that need to be drawn for principal directions or even lookup tables for individual array positions. As usual this is all very game specific, and within a game obviously view specific.

A related technique for decreasing the amount of time it takes to process the current view is Level Of Detail (LOD), which involves either storing a number of copies of either models or textures at different levels of detail, or dynamically calculating the detail that needs to be displayed based on the distance of objects/images from the viewpoint. Such techniques may not be needed in games with near world interaction but become important in games like flying games where long views are essential to the game. (Actually this is one area Yarozers don't seem to have explored much at the time of writing).

It’s also worth mentioning that judicious use of background textures and sprites can give the appearance that a view to the horizon is being rendered when it isn’t. Robert Swan’s RACE2 demo is an example of this.

As with so many gaming aspects you have a trade off between computational time and use of memory. Ultimately the criteria for all such issues must favour playability above anything else. (As always playability is hard to put a finger on).

The general result of restricting the part of the world that is drawn is popup – as you move forward you see the world popping up in front of you which is often lessened by judicious use of fogging, but can lead to gloomy worlds.

Clipping the world to a square.

After all that we are just going to implement the simplest form of clipping possible by merely clipping the world to a uniform square of world squares surrounding the car. This is a simple approach but may be OK for starters, you can optimise this later based on constraints in the world, constraints on how the view point is allowed to move, and constraints on time and space. This clipping will also be useful when we get to collision detection.

Basically we just want to draw those parts of the world that lie in a square surrounding the car. This is rather wasteful as some squares behind the car will be processed, but the procedure has very little processing overhead. To implement the scheme we start by defining the length of the square in x and z:

```
#define CLIPXWIDTH 5
#define CLIPZWIDTH 5
```

… which you can see is not very much! We then define some global variables to define the start and end points for clipping in each direction:

```
long ClipXStart,ClipXEnd, ClipZStart, ClipZEnd;
```

Next we need a function to assign values to these variables whenever the view changes. A little complication is that we need to check that the start and end points are within the array bounds which is done via the MAX and MIN macros as follows.

```
void ClipWorld(int x,  int z){
      // convert car z and z position to array indices
      x=(x+GROUNDSIZEDIV2)/GROUNDSIZE; z=(z+GROUNDSIZEDIV2)/GROUNDSIZE;
      ClipZStart=MAX(x-CLIPXWIDTH,0);
      ClipZEnd=MIN(x+CLIPXWIDTH,GROUNDMAX_X);
      ClipXStart=MAX(z-CLIPZWIDTH,0);
      ClipXEnd =MIN(z+CLIPZWIDTH,GROUNDMAX_Z);
}
```

We convert the car’s position in the world to an array position by dividing by `GROUNDSIZE` the width or height of an individual square. How ever we need to add `GROUNDSIZE/2` to each of the x and z values before division to take into account the fact that the car starts off in the middle of a ground square. GROUNDSIZEDIV2 is defined in step11.h:

```
#define GROUNDSIZEDIV2   (GROUNDSIZE/2)
```

That being done we must call the function to update these values. Assuming that the view is likely to be changing most frames we stick it unconditionally into the main loop before rendering the world:

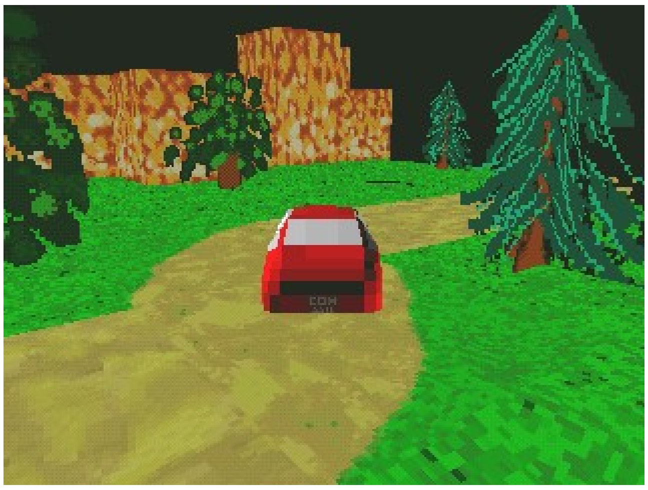
```
…
while(GameRunning){
      ProcessUserInput();
      ClipWorld(TheCar.Model.Object_Coord.coord.t[0],
            TheCar.Model.Object_Coord.coord.t[2]);
      RenderWorld();
}
…
```

Finally we can now implement the clipping by using the calculated start and end values in drawing the world:

```
void DrawWorld(WorldLayerStruct *world,   int precision)
{
int tmpz,tmpx;

   for (tmpz = ClipZStart; tmpz < ClipZEnd; tmpz++ ) {
      for (tmpx = ClipXStart; tmpx < ClipXEnd; tmpx++ ){
         if(world->ModelMap[tmpz][tmpx]!='0'){
            DrawModel(&world->Model[tmpz][tmpx],
                  &OTable_Header[CurrentBuffer],precision);
         }
      }
   }
}
```

3D STEP12. Adding a bit of scenery

FASTTRACK



To add some scenery to the world we need a world overlay map as follows:

```
char WorldOverlayData[GROUNDMAX_Z][GROUNDMAX_X] ={
{'0','0','0','0','1','0','0','0','0','0','0','0','0','0','0'},
{'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
{'2','2','0','0','0','0','0','0','0','0','0','0','2','0','0'},
{'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
{'1','0','0','0','2','0','0','0','0','0','0','0','0','1','0'},
{'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
{'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
{'0','1','0','0','0','0','0','0','0','0','0','0','4','0','0'},
{'0','0','0','0','0','0','0','0','3','0','3','4','3','2','0'},
{'0','0','0','0','0','0','0','0','0','3','0','0','0','0','0'},
{'0','0','0','0','0','0','0','0','0','1','0','0','2','0','0'},
{'0','1','0','0','0','0','0','0','0','0','0','0','0','0','0'},
{'4','0','0','0','0','0','0','1','0','0','0','0','0','0','0'},
{'0','3','0','0','0','0','0','0','0','2','0','0','0','0','0'},
{'0','0','0','0','4','0','0','0','0','0','0','0','0','0','0'}
};
```

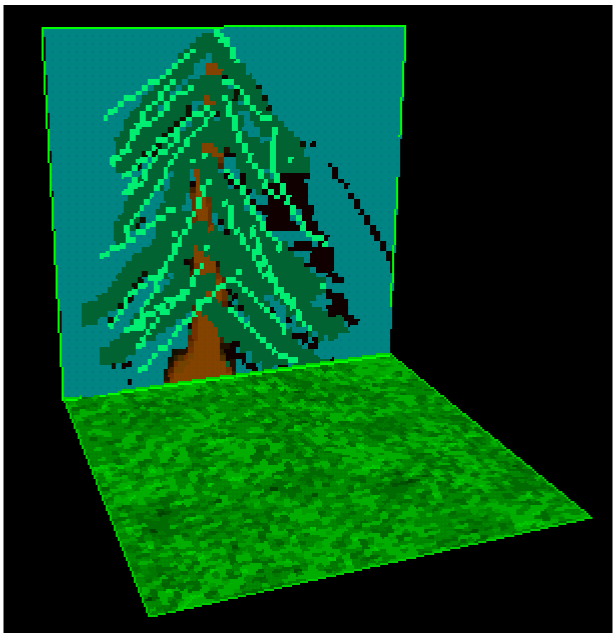
In this array ‘1’ signifies and oak-type tree as shown above to the left of the car and infrom of the blocks; ‘2’ signifies a fir tree as shown to the left of the car, ‘3’ denotes a short block; and ‘4’ denotes a tall block. The fast track approach is to edit this array to organise your own scenery. Note that all scenery has a processing overhead. The second step in the fasttrack is to put your own textures onto the tree objects:

- Create your own textures
- use RSDtool to texture the trees and save the modified RSD’s
- place your textures in screen memory using TIMtool
- run rsd2tmd in the data directory to update TMD texture information.

SLOWTRACK

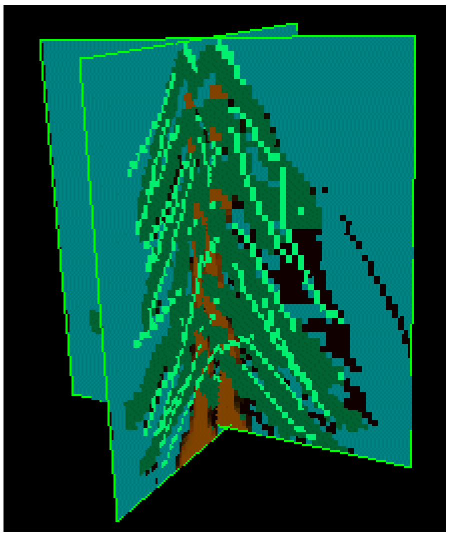
In this step you get to add a bit of scenery to your world.

Cheap trees.



The cheapest way to build a tree is as part of the ground plane the tree shown above in RSDtool has been created in 3Dstudio by adding an extra square at 90 degrees to the ground square. The tree image, fir.tim, has been converted in TIMtool with black set to transparent (ie: the top box checked and the bottom box unchecked in the TIMtool Import Image window Semi Transparent Information section). The black background is shown in blue in RSDtool. In the step12 program the model shown above called ctree.tmd is used to line the part of the road immediately in front of the car. Single sided quadrilaterals were used in construction so these trees can only be seen from one side. Drive along the track through the trees and turn around and drive back, the trees disappear on the return journey.

Note that another problem is that ctree.tmd is being drawn with the precision used for the ground which means that the car doesn't go through a tree correctly. Try driving through a tree to the point where you can see the tree but are just about through it and you will discover that the rendering is incorrect because looking at the scene from above by pressing R1 or R2 clearly shows that the car has already passed through the tree. To make the car correctly intercept objects in the world they have to be inserted into the ordering table with the same precision.



A more expensive way to make a tree is from two double sided quadrilaterals is shown above (in RSDtool).

Adding a world overlay layer.

The easiest way to deal with the above situation is to create a world overlay layer.

```
WorldLayerStruct WorldOverlay;
```

A map is needed as defined in the fast track section to specify what objects should be placed where in the overlay layer. Once we have this we have to initialise the map so function `InitiliaseMaps()` gets an extra line to deal with the overlay layer:

```
void InitialiseMaps()
{int tmpz,tmpx;
      for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ){
            for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
WorldGround.ModelMap[tmpz][tmpx]=worldGroundData[tmpz][tmpx];
WorldOverlay.ModelMap[tmpz][tmpx]=WorldOverlayData[tmpz][tmpx];
            }
      }
}
```

We specify the world over lay precision to be the same as that of the car to overcome the problem described in the first section of this step.

```
#define WORLD_OVERLAY_PRECISION  2
```

The `RenderWorld()` function needs an extra line to draw the overlay layer:

```
void RenderWorld()
{
            RenderPrepare();
            DrawPlayer(&TheCar);
            DrawWorld(&WorldGround,WORLD_GROUND_PRECISION);
            DrawWorld(&WorldOverlay, WORLD_OVERLAY_PRECISION);
            // wait for V_BLANK interrupt
            VerticalSync=VSync(0);
            // print the vSync interval
            FntPrint("VSync Interval: %d.\n",VerticalSync);
            // force text output to the PSX screen
            FntFlush(-1);
            RenderFinish();
}
```

The data directory contains oak.tmd and oak.tim, fir.tmd and fir.tim, block.tmd and block.tim, block1.tmd (a double height block), and side.tmd (the ground square that includes an upright, one sided tree). You can of course make your own objects to insert into the world or change the texturing of these objects as outlined for the fasttrack section. If you wish to make your own tree using tree.dxf then copy this file to a new filename and convert it using the following to make the quadrangles double sided:

```
dxf2rsd -o newtree.rsd -quad2 20 -back newtree.dxf
```

To get the objects into the scene you have to edit memtool.dat to include the tmds and tims and then run memtool and rsd2tmd.bat. You also have to edit the `InitialiseWorld` function to add the new objects to the world, ie in the main loop after the ground squares are dealt with insert:

```
c= WorldOverlay.ModelMap[tmpz][tmpx];
if(c == '1')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
       (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE), (u_long *)OAK_TMD);
}
if(c == '2')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
       (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE), (u_long *)FIR_TMD);
}
if(c == '3')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
       (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
       (u_long *)BLOCK_TMD);
}
if(c == '4')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
       (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
       (u_long *)BLOCK1_TMD);
}
```

3D STEP 13. Collision detection.

FASTTRACK

In this step collision detection between the car and world overlay objects is developed. To use it, in the function `AdvanceModel` a bounding box for the car must be specified by calling func and passing the centre of the object and minimum and maximum extents of the object in each of the x,y, and z directions:

```
SetBoundingBox(TheCar.Range,
            TheCar.Model.Object_Coord.coord.t[0],
            TheCar.Model.Object_Coord.coord.t[1],
            TheCar.Model.Object_Coord.coord.t[2],
            -205,205,-158,158,-500,500);
```

Then we go through all overlay objects in the current clipped view checking for collision. If collision is detected then the car bounces back.

```
      for (tmpz = ClipZStart; tmpz < ClipZEnd; tmpz++ ) {
         for (tmpx = ClipXStart; tmpx < ClipXEnd; tmpx++ ){
            if(!(WorldOverlay.ModelMap[tmpz][tmpx]=='0')&&
               CollisionDetect(TheCar.Range,
                WorldOverlay.Range[tmpz][tmpx])){
                  gsObjectCoord->coord.t[0] -=
                     (currentDirection->vx/ nD)*2;//64;
                  gsObjectCoord->coord.t[1] -=
                     (currentDirection->vy/ nD)*2;
                  gsObjectCoord->coord.t[2] -=
                     (currentDirection->vz/ nD)*2;//64;
                  }
            }
        }
```

SLOWTRACK

Collision detection is obviously very important for most games and can take many forms. The most commonly used collision detection used in 3D games is to use a simple bounding shape for your objects and see whether the bounding shapes collide. The bounding shapes are usually boxes, spheres, or cylinders. However note that there are may other collisions you may wish to detect in your games such as: the intersection of two lines in a plane or in space, the intersection of a line and a plane, the intersection of two planes, the intersection of a plane and a 3D shape like a box , the intersection of individual or sets of polygons. Also some schemes may use hierarchical collision detection. All these schemes can be found searching the web.

However we will take the simplest case which is the intersection of two boxes. The basic test applied to each of the x ,y, and z values is:

If the minimum value for Box1 is greater than the maximum value for Box2 or the minimum value for Box2 is greater than the maximum value for Box1 then there is clearly no intersection. This is applied for each of x,y, and z:

Box1.Xmin>Box2.Xmax OR Box2.Xmin>Box1.Xmax OR
Box1.Ymin>Box2.Ymax OR Box2.Ymin>Box1.Ymax OR
Box1.Zmin>Box2.Zmax OR Box2.Zmin>Box1.Zmax

The bounding boxes to define an objects 3D range is stored in a `VECTOR` array variable called `Range` where the first vector is the minimum values and the second vector the maximum values, thus the structs for both the player and the world have such a variable:

```
typedef struct
{   ModelStruct              Model;
       VECTOR                Direction;
       VECTOR                Range[2];
}PlayerStruct;

typedef struct
{
       ModelStruct           Model[GROUNDMAX_Z][GROUNDMAX_X];
       char                  ModelMap[GROUNDMAX_Z][GROUNDMAX_X];
       VECTOR                Range[GROUNDMAX_Z][GROUNDMAX_X][2];
} WorldLayerStruct;
```

In order to set these values a `SetBoundingBox` function is defined which takes the vector array in which values are to be stored, the values of the centre of the object, and minimum and maximum values from the centre for each of x,y, and z. The information about the extents of an object can be found by using the DOS RSDFORM utility. RSDFORM can be used for scaling rotating or translating an rsd model, but if just called with the –v flag and no transformation parameters it just gives statistics. Here is and example of calling this utility for block1:

```
E:\steps\step12\DATA>rsdform -v block1.rsd
(C) 1994-1996 Sony Computer Entertainment Inc. All Rights Reserved.
rsdform Version 1.81 Mon Apr  1 12:00:00 1996

        Input  RSD   : block1.rsd
        Output RSD   : a

        Scale        :     1,     1,     1
        Rotation     :     0,     0,     0  (degree)
        Translation  :     0,     0,     0
        Range      x :    -601.2 ...    +611.2 (center:     +5.0)
                   y :   -2008.0 ...      +1.0 (         -1003.5)
                   z :    -600.0 ...    +600.0 (            +0.0)

          block1.ply ->           a.ply
          block1.mat ->           a.mat
          block1.grp ->           a.grp
          block1.rsd ->           a.rsd

        New Range  x :    -601.2 ...    +611.2 (center:     +5.0)
                   y :   -2008.0 ...      +1.0 (         -1003.5)
                   z :    -600.0 ...    +600.0 (            +0.0)
```

Thus we can see that the range for both x and z around the centre of the object is approximately –600 to +600 and for y from –2000 to 0. The `SetBoundingBox` function is as follows:

```
void SetBoundingBox(VECTOR Range[2],int mx,int my, int mz, int xmin, int
xmax,int ymin, int ymax,int zmin,int zmax)
 {
 Range[0].vx=xmin+mx; Range[0].vy=ymin+my; Range[0].vz=zmin+mz;
 Range[1].vx=xmax+mx; Range[1].vy=ymax+my; Range[1].vz=zmax+mz;
}
```

The collision detection function is then defined as follows:

```
int CollisionDetect(VECTOR v1[2], VECTOR v2[2])
{
      if(   v1[0].vz>v2[1].vz || v2[0].vz>v1[1].vz ||
            v1[0].vy>v2[1].vy || v2[0].vy>v1[1].vy ||
            v1[0].vx>v2[1].vx || v2[0].vx>v1[1].vx)
            return(FALSE);
      else  return(TRUE);
}
```

Obviously we have to set the bounding box parameters for all objects in the overlay world that we may collide with which we do in `InitialiseWorld`:

```
c= WorldOverlay.ModelMap[tmpz][tmpx];
   if(c == '1')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
         (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
         (u_long *)OAK_TMD);
      SetBoundingBox(WorldOverlay.Range[tmpz][tmpx],
          tmpz * GROUNDSIZE,0,tmpx * GROUNDSIZE,
         -100,100,-300,0,-100,100 );
   }
   if(c == '2')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
         (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
         (u_long *)FIR_TMD);
      SetBoundingBox(WorldOverlay.Range[tmpz][tmpx],
         tmpz * GROUNDSIZE,0,tmpx * GROUNDSIZE,
         -100,100,-300,0,-100,100 );
   }
   if(c == '3')  {
      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
         (tmpz * GROUNDSIZE),(0), (tmpx * GROUNDSIZE),
         (u_long *)BLOCK_TMD);
      SetBoundingBox(WorldOverlay.Range[tmpz][tmpx],
            tmpz * GROUNDSIZE,0,tmpx * GROUNDSIZE,
            -600,600,-600,0,-600,600 );
   }
   if(c == '4')  {      InitialiseModel(&WorldOverlay.Model[tmpz][tmpx],
         (tmpz * GROUNDSIZE),(0),(tmpx * GROUNDSIZE),
         (u_long *)BLOCK1_TMD);
      SetBoundingBox(WorldOverlay.Range[tmpz][tmpx],
         tmpz * GROUNDSIZE,0,tmpx * GROUNDSIZE,
         -600,600,-2000,0,-600,600 );
   }
```

We also need to set the bounding box for the car, which needs to be updated when the car moves. When the bounding box is updated we also need to detect whether a collision has occurred and decide what to do about it. A common approach to responding to a collision for a car is to make it bounce back in the opposite direction it is travelling which is what we implement. The `AdvanceModel` function now looks like this:

```
void AdvanceModel (GsCOORDINATE2 *gsObjectCoord, VECTOR *currentDirection,
int nD)
{
       int tmpz,tmpx;
   if(nD!=0){ // avoid divide by 0 error
      nD = ONE/nD;
      gsObjectCoord->coord.t[0] += currentDirection->vx/ nD;
      gsObjectCoord->coord.t[1] += currentDirection->vy/ nD;
      gsObjectCoord->coord.t[2] += currentDirection->vz/ nD;
      gsObjectCoord->flg = 0;
      SetBoundingBox(TheCar.Range,
            TheCar.Model.Object_Coord.coord.t[0],
            TheCar.Model.Object_Coord.coord.t[1],
            TheCar.Model.Object_Coord.coord.t[2],
            -205,205,-158,158,-500,500);
      for (tmpz = ClipZStart; tmpz < ClipZEnd; tmpz++ ) {
         for (tmpx = ClipXStart; tmpx < ClipXEnd; tmpx++ ){
            if(!(WorldOverlay.ModelMap[tmpz][tmpx]=='0')&&
               CollisionDetect(TheCar.Range,
                WorldOverlay.Range[tmpz][tmpx])){
                  gsObjectCoord->coord.t[0] -=
                     (currentDirection->vx/ nD)*2;
                  gsObjectCoord->coord.t[1] -=
                     (currentDirection->vy/ nD)*2;
                  gsObjectCoord->coord.t[2] -=
                     (currentDirection->vz/ nD)*2;
                  }
            }
        }
   }
}
```

Note that the collision detection is done for all world overlay objects which are within the clipping range. For the current example the clipping range provides an area for collision detection which is greater than we need. As the car moves so slowly it really only needs to check for collisions with its immediately adjacent squares. However if you had objects that were moving very fast the clipping area may be too small. Also we only really need to consider collisions in the direction we are moving.

The approach taken here is rather crude and doesn’t take into account the orientation of the car, so if you collide with a block along the z axis the point of collision is correct whereas if you collide along the x axis it is incorrect – the front of the car actually goes into the block some way before collision is detected. Ideally, collision detection would take orientation in to account and would be applied on rotation as well as translation of the object.

3D STEP 14. Shooting.

FASTTRACK

In this step we allow the car to shoot any world overlay object.

SLOWTRACK

To create a bullet we need a structure for it similar to the Player and world structs:

```
typedef struct
{long speed;
      ModelStruct       Model;
      VECTOR            Range[2];
      long              active;
      VECTOR            Direction;
}BulletStruct;

BulletStruct TheBullet;
```

We decide to allow the car only one bullet. We initialise it by initilaising its model struct and setting it’s active flag to 0:

```
void InitialiseBullet(BulletStruct *theBullet, int nX, int nY, int nZ,
      unsigned long *lModelAddress)
{

    InitialiseModel(&theBullet->Model,nX,nY,nZ,(u_long *)ModelAddress);
    theBullet->Model.Object_Handler.attribute=
      theBullet->Model.Object_Handler.attribute|(1<<31);
    theBullet->active=0;
}
```

We need a function to draw the bullet:

```
void DrawBullet(BulletStruct *theBullet,int precision)
{
      DrawModel(&theBullet->Model,
        &OTable_Header[CurrentBuffer],precision);
}
```

We also need a function to set the direction in which the bullet will travel, it starts from the centre of the car and will move in the direction the car is facing:

```
void SetBulletVector(GsCOORDINATE2 *gsObjectCoord,VECTOR currentDirection,
BulletStruct *theBullet)
{
       if(theBullet->active==1){
       theBullet->Direction.vx= currentDirection.vx;
       theBullet->Direction.vy= currentDirection.vy;
       theBullet->Direction.vz= currentDirection.vz;
       theBullet->Model.Object_Coord.coord.t[0] =
            gsObjectCoord->coord.t[0];
       theBullet->Model.Object_Coord.coord.t[1] =
            gsObjectCoord->coord.t[1];
       theBullet->Model.Object_Coord.coord.t[2] =
            gsObjectCoord->coord.t[2];
       theBullet->active=1;
       }
}
```

The function that does most of the work for the bullet is func. This function advance the bullet if it is active and check for collisions with overlay objects. If a collision is detected the object handler attribute bit 31 is set for both the overlay object and the bullet, which means they will not be drawn in the future. If the bullet does not collide with an object repeated calls to this function decrease the static variable life. When life == 0 the bullet is set to inactive and not drawn.

```
void AdvanceBullet(BulletStruct *theBullet)
{int nD=256;
static int steps=(ONE)/64;
static int life=steps;
int tmpx,tmpz;
nD = 4096/nD;
// if the bullet is active move it forward
      if(theBullet->active==1){
       theBullet->Model.Object_Coord.coord.t[0] +=
            theBullet->Direction.vx / nD;
       theBullet->Model.Object_Coord.coord.t[1] +=
            theBullet->Direction.vy / nD;
       theBullet->Model.Object_Coord.coord.t[2] +=
            theBullet->Direction.vz / nD;
       SetBoundingBox(theBullet->Range,
            theBullet->Model.Object_Coord.coord.t[0],
            theBullet->Model.Object_Coord.coord.t[1],
            theBullet->Model.Object_Coord.coord.t[2],
            -25,25,-25,25,-25,25 );
       life--;
// if there is an object and it is being drawn and there is a collision
       for (tmpz = ClipZStart; tmpz < ClipZEnd; tmpz++ ) {
            for (tmpx = ClipXStart; tmpx < ClipXEnd; tmpx++ ){
                  if(!(WorldOverlay.ModelMap[tmpz][tmpx]=='0')&&
 (!(WorldOverlay.Model[tmpz][tmpx].Object_Handler.attribute&(1<<31)))&&
                   CollisionDetect(theBullet->Range,
                  WorldOverlay.Range[tmpz][tmpx])){
                    // then set the bullet to be inactive
                    theBullet->active=0;
                    // set attribute bit 31 to 1 so bullet wont be
                    // drawn again
                  theBullet->Model.Object_Handler.attribute=
                  theBullet->Model.Object_Handler.attribute|(1<<31);
                    // set attribute bit 31 to 1 so overlay object wont
                    //be drawn again
            WorldOverlay.Model[tmpz][tmpx].Object_Handler.attribute =
      WorldOverlay.Model[tmpz][tmpx].Object_Handler.attribute|(1<<31);
                        }
                  }
            }
```

```
        // if bullet life is over set it inactive and set attribute bit
        // to stop it being drawn
        if(life==0){
             theBullet->active=0;
             life=steps;
             theBullet->Model.Object_Handler.attribute=
             theBullet->Model.Object_Handler.attribute|(1<<31);
             }
        theBullet->Model.Object_Coord.flg = 0;
   }
}
```

The bullet is fired by pressing the R1 button:

```
if (PADstatus & PADR1){
     FntPrint ( "PAD padR1:Shoot\n");
      if (TheBullet.active==0){
            TheBullet.Model.Object_Handler.attribute=
               TheBullet.Model.Object_Handler.attribute&(0<<31);
            TheBullet.active=1;
            SetBulletVector(&TheCar.Model.Object_Coord,
               TheCar.Direction,&TheBullet);
            AdvanceBullet(&TheBullet);
      }
}
```

To reset the demo the start button is pressed. The car is moved to it’s starting position and all world overlay objects have their object handler attribute bit 31 set to 0 so they will be drawn.

```
if(PADstatus & PADstart){
      FntPrint ( "PAD start\n" );
      TheCar.Model.Object_Coord.coord=GsIDMATRIX;
      TheCar.Model.Object_Coord.coord.t[0]=0;
      TheCar.Model.Object_Coord.coord.t[1]=-200;
      TheCar.Model.Object_Coord.coord.t[2]=0;
      TheCar.Direction.vx=0;TheCar.Direction.vy=0;
      TheCar.Direction.vz=ONE;
      TheCar.Model.Object_Coord.flg=0;
      for (tmpz = 0; tmpz < GROUNDMAX_Z; tmpz++ ){
         for (tmpx = 0; tmpx < GROUNDMAX_X; tmpx++ ){
            if ((!(WorldGround.ModelMap[tmpz][tmpx]=='0'))&&
(WorldOverlay.Model[tmpz][tmpx].Object_Handler.attribute&(1<<31)))
                {
WorldOverlay.Model[tmpz][tmpx].Object_Handler.attribute ^=(1<<31);
                  WorldOverlay.Model[tmpz][tmpx].Object_Coord.flg=0;
              }
          }
        }
}
```

To draw the bullet we need to insert a line into the RenderWorld function:

DrawBullet(&TheBullet,WORLD_OVERLAY_PRECISION);

Finally in the main function we need to initialise the bullet and call advance bullet inside the main game loop:

```
…
      InitialiseBullet(&TheBullet,0,0,0,(long*)BALL_TMD);
      while(GameRunning){
            ProcessUserInput();
            ClipWorld(TheCar.Model.Object_Coord.coord.t[0],
                  TheCar.Model.Object_Coord.coord.t[2]);
            AdvanceBullet(&TheBullet);
            RenderWorld();
            }
…
```

Implementing Sound Effects

FASTTRACK

Include the .vh and .vb files into memtool.dat and run MEMTOOL to get them loaded

```
static u_short BankId;

void main()
        {
        BankId = InitialiseSound(SOUND_VH, SOUND_VB);
        }

void anyfunctionyouwant()
        {
        PlaySound(BankId, 5, 0, 48);
        StopSound();
        }
```

BankId Variable to hold sound bank identification number. Allows use of more than one sound bank.
InitialiseSound Macro to copy .vb data into sound ram, and assign .vh data location to SPU
PlaySound Macro to play sound number 5 from BankId sound bank, on sound channel 0 (0-15 channels are valid), with pitch 48 (48 = middle C, +/- 12 raise/lower by an octave)
StopSound Macro to turn all sounds off.

SLOWTRACK

The PSX uses its own format for sound effects, which consists of two associated files, a header file (.vh) containing information about how to play 'programs' of sound. A program consists of telling the SPU what samples to play and how. E.g program 5 could specify playing sample 5 at half speed and samply 7 only on the left channel. On the course, we will use programs at their simplest; program 0 specifies playing back sample 0 etc.. The sound bank file (.vb) contains all the samples, and this must be copied to the sound ram at the program initialisation stage. Because you can use more than one sound bank per program, a variable needs to be declared to hold the bank identification number decided upon the by PSX.

```
// declare variable to hold sound bank id number
static u_short BankId;
```

In void main(), initialise sound, and pass the names of your .vh and .vb files with it:

```
// initialise sound and return bank id number
BankId = InitialiseSound(SOUND_VH, SOUND_VB);
```

Now all you have to do to play a sound is call the appropriate macro, PlaySound() in your program:

```
// play program 0, which consists just of sample 0
PlaySound(BankId, 0, 0, 48);
```

The first argument is the bank id number (found during initialisation), then the program number, which in this case just plays the corresponding sound effect, followed by the channel number. This is a value of 0-15. Each channel can play one program at a time and usually more, but if you start playing too many sounds on one channel you may find some turning on and off at random so unless you want to play a lot of sounds, it is best to give each a different channel number. The last number dictates what pitch (and hence speed) the sample should be played back. 48 is middle C, equivalent to playback at the recorded speed. The number is specified in semi-tones, so giving a value of 60 plays the sample back at twice the recorded speed, as 60-48 = 12, and their are 12 semi-tones to an octave. There is an included macro to turn off all sounds:

```
// turn off all sound effects
StopSound();
```

Converting .wav files for the PSX

FASTTRACK

Make sure all samples are 16bit straight or monoaural.
Run DEFTOOL from dos specifiying how many samples you want to use; ie DEFTOOL <#> to create .def file
Convert each .wav file to .vag format using dos prorgam AIFF2VAG <.wav file>
Convert .def file and .vags into one .vab file using dos program MKVAB -f <.def file> -o <output .vab> <vag file 1> ... <vag file #>
Convert .vab file into .vh and .vb playstation files using dos program VABSPLIT <.vab file>

SLOWTRACK

Creating the sound files for the playstation involves quite a few steps, but providing that you have your .wav files in the correct format to begin with the process is fairly painless. The correct format for your .wavs are as 16bit straight or monaural uncompressed data. The PSX uses a .def file to compile all the programs for the spu to use, but as our programs are going to be the simplest possible, we can use DEFTOOL to create a .def file automatically. If you are going to want 6 samples in the sound bank, you would use the following line from the dos prompt:

DEFTOOL 6

Which creates the file ******.def ready to be used. Next to convert all the .wavs into .vags for the PSX. Another dos tool does this called aiff2vag. Simply specify one or more .wav files on the command line and it will convert them, as:

AIFF2VAG SWOOSH.WAV BOOM.WAV

which will convert two .wavs. These .vags and the .def file are combined into one large file (.vab) now, using the dos mkvab utility. At this point, an error will occur if you specify a different amount of .vags to include in the .vab to the number you specified using DEFTOOL.

MKVAB –f ******.def –o SOUND.VAB SWOOSH.VAG BOOM.VAG

This creates the .vab file called sound.vab, including two .vags and using the created ******.def file. Now all that remains is to split to the .vab file into the header (which remains in main memory on the PSX) , and sound sample sections (which move to the sound RAM). This is the least painful part, and is done from dos by using:

VABSPLIT SOUND.VAB

which splits the .vab into two appropriately named parts, sound.vh and sound.vb, which are ready to be included into your program.

Fogging

FASTTRACK

```
GsFOGPARAM  Fogging;

void main()
      {
      Fogging.dqa = -15000;
      Fogging.dqb = 5120*5120*1.0;
      Fogging.rfc = Fogging.gfc = Fogging.bfc = 0;
      GsSetLightMode(1);
      GsSetFogParam(&Fogging);
      }
```

GsFOGPARAM	Create instance of fogging structure
Fogging.dqa	Measure of distance at which start to fade out. It is not directly related to world units. More -ve = start to fade out further in front
Fogging.dqb	Measure of the spread of fading. A value of 5120*5120*2 = very abrupt fading, 5120*5120*0.7 = smooth fading
Fogging.rfc	RGB colour to fade out to

GsSetLightMode(1)
Set lighting model to use fogging
GsSetFogParam(&Fogging)
Specify fogging structure to use in the lighting model

SLOWTRACK

Fogging is a process that makes 3d objects fade more and more to a specified colour the further that object is from the viewpoint. It is extremely useful as it can give a natural look to your scene (such as fading objects to black in a night scene, or giving a blue tint to objects further away, like in natural light, or fading to white to give a mist effect), and it can also help reduce object pop-up in scenes that are clipped. Rather than have objects suddenly jump into the view as you move around your world, you can have them fade in, which when used against complimentary background makes them appear in a less obvious manner. To use fogging, you have to use the fogging structure GsFOGPARAM:

```
// create instance of fogging structure
GsFOGPARAM  myFogging;
```

The parameters that need to be set include the rfc, gfc and bfc member variables, and these determine the colour that you want the objects to completely fade out to in the distance. These are passed as values from 0 to 255, and it must be noted that these are scalar operations applied to colours on objects, meaning that if you set them all to 255 (which should in theory scale everything to a white colour) and there is an object with no red component in its pigment colour, then there is no red to be scaled up by the fogging routine, and the brightest colour it would reach would be cyan (full green and blue brightness). If you are using fogging to any colour other than black, it may be worth including even a minimal amount of each of the RGB components in it's pigment to avoid that effect. The two main member variables are dqa and dqb, which are attenuation components. The values in this are based on what works for the processing of fogging, and as such the values are pretty meaningless to the user so trial and error is really what is required to get the effect looking how you want. Nice default values need to be set in void main() during initialisation, but after you have set up all the lights in the scene otherwise it will not be registered:

```
myFogging.dqa = -15000;
myFogging.dqb = 5120*5120*1.0;
GsSetLightMode(1);
GsSetFogParam(&Fogging);
```

The dqa value indicates at what z ditance from the viewpoint objects will start to fade to the specified colour. If you make the dqa value less negative, then this distance decreases and objects fade closer to the viewpoint. If you make it more negative, this distance increases. In fact it is possible to mess around with it so much that objects fade to the colour you specified the closer they are to the player! The dqb variable dictates over how large a distance the change from normal brightness to faded colour occurs. Altering it to 5120*5120*2 makes the change very abrupt (and hardly worth it), whereas a value like 5120*5120*0.5 spreads the change out. Beware again though, you can make the change so gradual as to not be noticeable. The best you can do is play around a few times with the values.

Dynamically created TMDs

NOTE:
If you are using a background map AND dynamically created Tmds, then setup the background map first, and when you create your first tmd, do not pass it FREE_MEM as the location to start from as this is where the map data is. As it is, you can simply start by passing 0 as the argument of where to start from, as you would for your second or subsequent models. For more information, see below.

About dynamically created TMDs

Dynamically created TMDs allow you to make up 3d models from within your program. While they are not as intuitive to use as a full 3d model generation program like 3D Studio Max and complex models are unwieldy and almost impossible to make, there are a variety of reasons why you would want to create certain models by the computer at run-time:

- dynamic models are specified using world coordinates, unlike model packages which require imprecise scaling to the PSX
- to edit dynamic models requires only recompiling. Editing within a package requires conversion.
- dynamic models can be created from mathematical formula to allow precise creation of models hard to create within a modelling package.
- dynamic models allow you more control over semitransparency of polygons with a model
- dynamic models give access to different primitives that modellers can not access.
- They don't require downloading to the PSX, meaning your program will start running much quicker

Creating a simple cube model

Creating a model in memory requires a small amount of setting up in the program. You need to know where in main memory you can start creating the models, and also how to go about defining the vertices and polygons that make up the model. Fortunately, the MEMTOOL utility does the first bit for you. When you run MEMTOOL, it not only creates a list of #defines that give names for all your textures and models, but also creates one extra #define, called FREE_MEM. This is an address in main memory after all loaded files, so that you can happily create your models without fear of writing over already loaded models or textures. The model is going to be created in a function called createCube() surprisingly enough. Create the function prototype at the start of your program as follows:

```
// prototype to create a cube model in memory
void createCube();
```

and add a call to it at the start of void main():

```
// call the function that creates the cube model
createCube();
```

Now for the actual function to create the model. A cube has 8 vertices and 6 faces, so assuming we are going to create a cube with each face a different colour, then we create an array to hold this information. Fortunately there are already two types of structure defined to hold exactly this sort of information, VERTEX and RGB.

```
// dynamic TMD creation
void createCube()
      {
      VERTEX      vert[8];
      RGB         cols[6];
```

the next section is where we have to set these. There are two functions written to speed up this process. TMD_setVERTEX() allows you to set the x, y and z positions of a VERTEX in one line. The first argument is the address of the VERTEX to edit, and the other three are the x, y and z positions respectively for that vertex.

```
// set the positions of the vertices
      TMD_setVERTEX(&vert[0], -200, -200, -200)
      TMD_setVERTEX(&vert[1], -200, -200, 200)
      TMD_setVERTEX(&vert[2], 200, -200, 200)
      TMD_setVERTEX(&vert[3], 200, -200, -200)
      TMD_setVERTEX(&vert[4], -200, 200, -200)
      TMD_setVERTEX(&vert[5], -200, 200, 200)
      TMD_setVERTEX(&vert[6], 200, 200, 200)
      TMD_setVERTEX(&vert[7], 200, 200, -200)
```

If you were to plot those points on a piece of paper you would see they form the corners of a cube. Remember that for the PSX, the positive Y direction is down. Creating models requires planning, and preferably a sketch of what you want to create, as order of vertices becomes very important when defining the faces of the cube. For now, though we specify the six colours using the TMD_setRGB() function as follows:

```
// set the colours to be used in the shape
      TMD_setRGB(&cols[0], 255, 255, 255);
      TMD_setRGB(&cols[1], 255, 255, 0);
      TMD_setRGB(&cols[2], 255, 0, 255);
      TMD_setRGB(&cols[3], 0, 255, 255);
      TMD_setRGB(&cols[4], 0, 255, 0);
      TMD_setRGB(&cols[5], 0, 0, 255);
```

The arguments are the address of the RGB to edit, followed by the red, green and blue elements of that colour. Now to actually use this information to create a cube. Every time you create a model, you must have a call to the TMD_prepare() function as follows.

```
// prepare for creating the model
      TMD_prepare(FREE_MEM, &vert[0], &cols[0], 0);
```

What that does is pass the address of where to start creating the model in memory. It also passes the start addresses of the vertex and colour arrays. If this is not done then when you create the polygons it will take information from some unpredictable part of the memory. Next you define each of the faces, one at a time using (in this example) the TMD_face4_NS_FP() function. For a list of other types of polygons you can create, see the second following section. For now, all you need to know is that you are creating a 4 sided polygon, with no shading (meaning it won't react to any light sources) and with a single colour (flat pigment). The last argument is actually a pointer to the start of a texture array, but as we are not using them we might as well send a useless value.

To use the function, you must specify which vertices (as indices to the vertex array) you want to make up the four corners of the polygon. To understand which order you need to specify them in, hold up a piece of paper in front of you. You would specify the corners of the piece of paper in a 'Z' fashion, i.e. the top left corner first, then the top right, then the bottom left, followed by the bottom right. This would create a polygon facing you. However this only would create a one sided polygon, which you might or might not want. Imagine turning the piece of paper to face the other way and that you see nothing, just a see-through back! For solid models you could never see the reverse side of a polygon, but in some models you may need to specify two polygons, one for front and back as with a piece of paper.

The next argument simply specifies which colour to use (as an index to the colour array). The last argument is either true or false, and indicates whether you want that polygon to be semitransparent. Note that if you specify it semitransparent here, there is no way to make it solid later! The reverse isn't true though, so be aware of what you want before you start.

So lets have a look at the code to create the first face:

```
// create the faces of the cube
      TMD_face4_NS_FP(0, 1, 3, 2, 0, FALSE);
```

the polygon will use vertices 0, 1, 2 and 3 correctly listed as 0, 1, 3 and 2 owing to the Z order. The next 0 means it will use colour zero (which was white) and that it will be solid to begin with. The following five faces are as follows:

```
      TMD_face4_NS_FP(2, 3, 7, 6, 1, FALSE);
      TMD_face4_NS_FP(0, 3, 4, 7, 2, FALSE);
      TMD_face4_NS_FP(1, 0, 5, 4, 3, FALSE);
      TMD_face4_NS_FP(2, 1, 6, 5, 4, FALSE);
      TMD_face4_NS_FP(5, 4, 7, 6, 5, FALSE);
```

Now to wrap up the model and end the function. You use the TMD_finish() function at the end as follows:

```
// finish creating the model
      TMD_finish();
      }
```

And that's all there is to it. Your finished createCube() function should look like this.

```
// dynamic TMD creation
void createCube()
      {
      VERTEX      vert[8];
      RGB         cols[6];

// set the positions of the vertices
```

```
      TMD_setVERTEX(&vert[0], -200, -200, -200)
      TMD_setVERTEX(&vert[1], -200, -200, 200)
      TMD_setVERTEX(&vert[2], 200, -200, 200)
      TMD_setVERTEX(&vert[3], 200, -200, -200)
      TMD_setVERTEX(&vert[4], -200, 200, -200)
      TMD_setVERTEX(&vert[5], -200, 200, 200)
      TMD_setVERTEX(&vert[6], 200, 200, 200)
      TMD_setVERTEX(&vert[7], 200, 200, -200)

// set the colours to be used in the shape
      TMD_setRGB(&cols[0], 255, 255, 255);
      TMD_setRGB(&cols[1], 255, 255, 0);
      TMD_setRGB(&cols[2], 255, 0, 255);
      TMD_setRGB(&cols[3], 0, 255, 255);
      TMD_setRGB(&cols[4], 0, 255, 0);
      TMD_setRGB(&cols[5], 0, 0, 255);

// prepare for creating the model
      TMD_prepare(DYNAMIC_TMD_START, &vert[0], &cols[0], 0);

// create the faces of the cube
      TMD_face4_NS_FP(0, 1, 3, 2, 0, FALSE);
      TMD_face4_NS_FP(2, 3, 7, 6, 1, FALSE);
      TMD_face4_NS_FP(0, 3, 4, 7, 2, FALSE);
      TMD_face4_NS_FP(1, 0, 5, 4, 3, FALSE);
      TMD_face4_NS_FP(2, 1, 6, 5, 4, FALSE);
      TMD_face4_NS_FP(5, 4, 7, 6, 5, FALSE);

// finish creating the model
      TMD_finish();
      }
```

Of course, that doesn’t explain how to use the model. Its simple though. Now it has been created, FREE_MEM points to a valid model in memory, and can be substituted for any of the other #defined names when associating a model to a model handler. You could add the code above to a program right now, and replace a car with your cube.

Multiple dynamic models

If you want to create more than one model in memory then there a couple of things you need to do. Lets assume you want to create two cubes in memory, and have already created two functions for them, called createCubeOne() and createCubeTwo() which at the moment are both identical to the function above. FREE_MEM only points to the start of the first model, so you need a way of holding the addresses of the two models. To do this, create an array of two to hold these two start addresses:

```
// declare an array to hold dynamic model memory locations
u_long            *DYNAMIC_TMD[2];
```

There were two things that weren’t mentioned before. The first is to do with the TMD_prepare() function. In the example above the first argument passed was FREE_MEM and that specified where to start creating the model in memory. Obviously, if the second model is created at that location as well, it will overwrite the other one. Fortunately, all you need to do for creating the second and subsequent models is to pass a zero as the first argument. This tells the function to use the next piece of free memory after the previously created model. In this example, createCubeTwo() would have the line altered to:

```
TMD_prepare(0, &vert[0], &cols[0], 0);
```

Lastly, to find out where in memory the model is, TMD_finish() returns a pointer to the memory, which is entered into the DYNAMIC_TMD[] array. In the first example this return value was not used, as there was only the one model and we already knew where it started. To get these values, each of the functions will have something similar to the following to replace the old line:

```
DYNAMIC_TMD[0] = TMD_finish();
```

As long as each of the model creation routines uses a different array element then you have your method of finding the start of each model. This array of model start addresses would then be used in the same way as any other model address when linking a model to a model handler.

Understanding the different primitive types and using them

There are a variety of different primitive types that can be used. Note that lines and 3d sprites (billboarding) are not among them. A key to understanding the terms is at the bottom.

Function name	Details
`TMD_face3_NS_FP()`	`3 sided, no shading, flat pigment`
`TMD_face3_NS_GP()`	`3 sided, no shading, gradient pigment`
`TMD_face3_NS_FP_TX()`	`3 sided, no shading, flat pigment, textured`
`TMD_face3_NS_GP_TX()`	`3 sided, no shading, gradient pigment, textured`
`TMD_face3_FS_FP`	`3 sided, flat shading, flat pigment`
`TMD_face3_FS_GP`	`3 sided, flat shading, gradient pigment`
`TMD_face3_FS_NP_TX()`	`3 sided, flat shading, no pigment, textured`
`TMD_face4_NS_FP()`	`4 sided, no shading, flat pigment`
`TMD_face4_NS_GP()`	`4 sided, no shading, gradient pigment`
`TMD_face4_NS_FP_TX()`	`4 sided, no shading, flat pigment, textured`
`TMD_face4_NS_GP_TX()`	`4 sided, no shading, gradient pigment, textured`
`TMD_face4_FS_FP`	`4 sided, flat shading, flat pigment`
`TMD_face4_FS_GP`	`4 sided, flat shading, gradient pigment`
`TMD_face4_FS_NP_TX()`	`4 sided, flat shading, no pigment, textured`

no shading	Polygon will not be affected by light sources at all
flat shading	Polygon is affected by light sources
no pigment	Polygon displays texture as is (textured only)
flat pigment	Polygon has one flat colour across if
gradient pigment	Polygon has a different colour at each vertex and blends in between them
textured	Polygon has a texture mapped to it

Table to show available types of polygons in dynamic TMDs and key to phrases

The arguments passed to each of these functions is different, but is calculated by the following rules:

1. Vertices are always passed first (either three or four arguments) as indices to the vertex array.
2. Colour information is passed next. one argument for flat pigment or three/four arguments for gradient pigment depending on number of vertices. Passed as indices to the colour array. For no pigment, no colour argument is passed.
3. Texture information is passed next (if applicable) as an index to the texture array.
4. Finally, an argument to state whether the polygon is initially semitransparent.

For example, a four sided, flat shading, gradient pigment polygon has the following function:

```
// example of 4 sided, flat shading, gradient pigment polygon
// TMD_face4_FS_GP(0, 1, 3, 2, 4, 12, 5, 7);
// polygon is made up of vertices 0, 1, 2 and 3
// colours of these vertices are 4, 12, 7 and 5
```

What hasn't been shown is how to apply a texture, and the arguments required to do so. To do so, you would create an array of texture information in the same way as colour and vertex information, using the structure TEXTURE:
slowly.

```
TEXTURE     txt[1];
```

What is required next involves a bit of thinking, and remembering where you put your textures into the video RAM with timtoll or TIMutil. As an example:

```
TMD_setTEXTURE(&txt[0], 320, 256, 383, 256, 256, 320, 319, 383, 319, 1, 0,
481);
```

Quite a lot of arguments! The first is the address of the texture array, with the next two being the coordinates in video RAM of the top left of whichever texture page the image you want is on. What follows next are four sets of coordinate, each referring to a location in video RAM, and it is these locations that specify exactly where the texture map is taken from. Like before, the points are specified in the Z order, so make sure you don't get them the wrong way around. If you are going to use a three sided polygon, simply send the coordinates for the fourth point as anything you like; they wont be used. The third to last argument will be either a 0, 1 or 2, and depends on the colour depth of the texture you are trying to map. 0 is used for 4bit 16 colour textures, 1 for 8bit 256 colour textures, and 2 for 16bit true colour mode. The last two are fairly simple, and are the coordinates in video RAM of the CLUT. If there is no CLUT (15bit true colour mode) , send any values you want, they wont make a difference.

To use textures, you then have to send the start address of the texture array in TMD_prepare() function as in the following example:

```
TMD_prepare(0, &vert[0], &cols[0], &txt[0]);
```

You can now use the textures in primitives that support them. The mechanism for setting textures is unwieldy, and an easier method is being investigated.

3D Studio MAX – Essentials for Yaroze modelling.

MAX is a very powerful 3D modelling and animation tool, which can be used for production quality work. It aims to allow you to make things that look great, but in doing this it usually generates thousands of polygons. The requirement for 3D modelling for games is different as your task is to use the least amount of polygons possible so you have to constantly take care to reduce the number of polygons.

Another problem to note that the centre of the object is always the centre of the world in MAX. This means you should place your object in MAX so that the centre of the world is your objects centre.

We have 5 copies of MAX 1.2 and 10 copies of MAX R2 and 1 copy of Release 4. From the door of the lab the 15 PCs on the right are the MAX machines, the first 10 run MAX R2 and the last 5 MAX 1.2. The very last machine also runs 3Dstudio R4. There is a plugin for MAX to export RSD files and TOD (animation files) but this is only available to licensed PSX developers. However there is a utility 3DS2RSD.EXE that will convert .3DS files to RSD files and this is the tool of choice but as usual involves a bit of fiddling about to get it working.

If you create your model as a set of different components you have two choices

1. you can export your model as a dxf, and use DXF2RSD to convert your file, but you will
   - lose the texture mapping options of 3dStudio (in rsdtool it is difficult to texture a set of polygons)
   - probably have some complications in conversion to .rsd and .tmd (some polygons may disappear or render untextured when you don't expect it, this can usually be fixed but there are lots of flags to set for dxf2rsd …)

2. you can export your model as .3DS convert it with 3DS2RSD and
   - the whole object can be texture mapped (but for complex objects this sometimes screws up a bit…)
   - the conversion is usually geometrically correct (but not always).

Using either of these approaches above you can produce 3D models for the PSX and can also produce 3D animation by either:

… creating a number of models of your object in different poses and cycling through them in your program.

… or you can get real fancy, create an articulated object, do some fancy animation on it, export it to R4 and save the model as a set of RSD's and save the animation as a .tod file – and end up with fancy animation eg: for a fighting game. I won't cover this here, but I can help you with it if you are interested.

Using the 3DS2RSD utility.

This DOS utility is supports conversion of .3DS mesh format files to RSD format, so when you save your models from MAX you have to export them as . 3DS format by choosing File|Export. To run 3DS2RSD you have to pretend that you have R4 on the machine you are running. To do this you have to create a dummy 3DS.SET file which sets a pathname to the directory where your .BMP texture files are ie: Create a file called 3DS.SET which set the variable map-path to *your* directory (in this example I used E:\3DBOOK\BULLET\DATA):

```
MAP-PATH = " E:\3DBOOK\BULLET\DATA "
END
```

You also have to a DOS environment variable to tell 3DS2RSD where the file you just created is. This should be the same directory. Ie: at the DOS prompt type:

```
SET 3DDIR=E:\3DBOOK\BULLET\DATA
```

To use MAX's texture mapping facility your texture map *must* be a true colour BMP file, which whilst being an annoying restriction is not a problem as you after using 3DS2RSD to do the conversion you can convert the BMP to TIM with a lower colour depth. NOTE: the texture mapping is often not quite correct but is an advance on using RSDTOOL.

Note that this software does not generate a .grp file which means you can't use it with tools like RSDFORM – however all the operations performed by RSDFORM, like scaling and rotation, can be done in MAX.

Summary of using 3DS2RSD.

1. In DOS set the variable to your data directory eg:
```
      SET 3DDIR=E:\3DBOOK\BULLET\DATA
```

2. Create a file called 3ds.set which sets the variable map-path to your directory:
```
      MAP-PATH = "E:\3dbook\bullet\DATA"
      END
```

3. If you are using texture mapping your texture file should be a true colour BMP.

4. Export your file from MAX using File|Export.

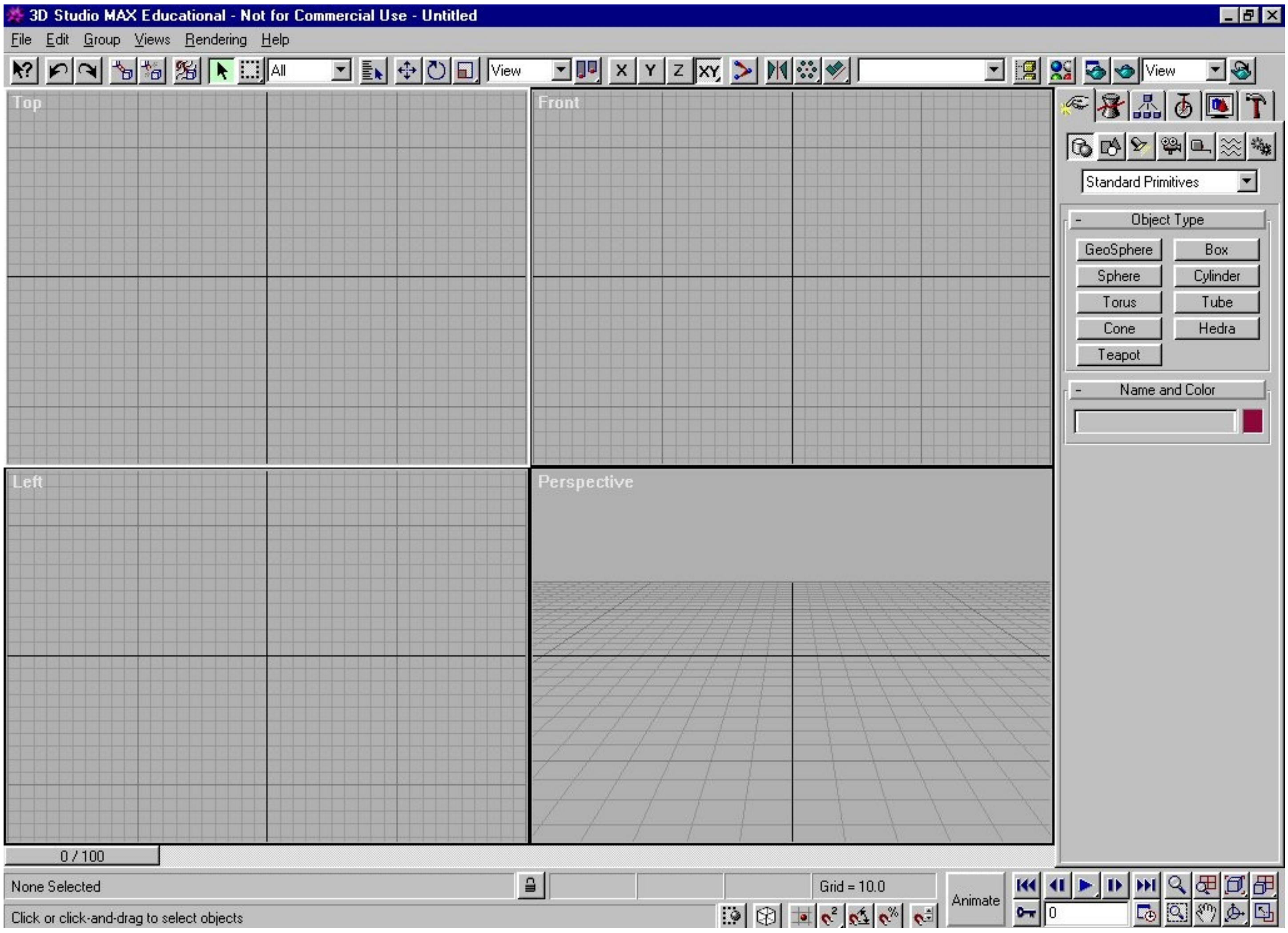
5. Convert your file to RSD by running 3DS2RSD eg:
```
      3DS2RSD -v myfile.3ds
```

6. If you are using texture mapping create a TIM file from your BMP reducing the colour depth if you want to.

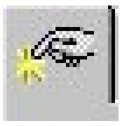
7. Use the model in your program, remembering to load texture files if appropriate.

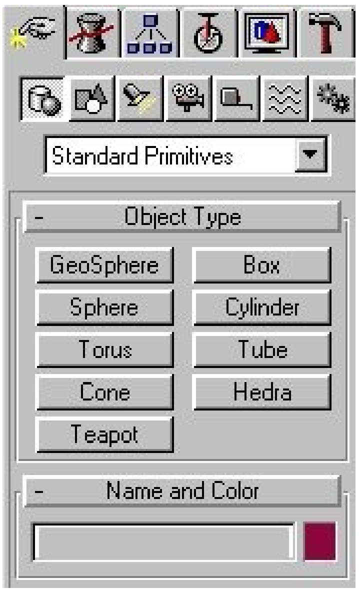
Start MAX.

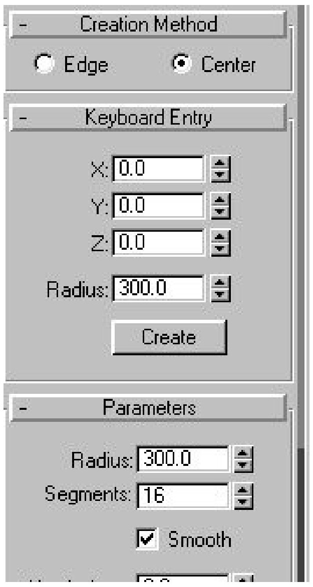
MAX is very sophisticated and complex and we will try to only cover the basic subset of features needed to get it going. The default screen layout is to give the four views shown.



Create a sphere.

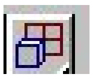






To create a sphere first click on the hand symbol on the upper right of the screen which then shows you a menu of possible shapes below under the heading 'Object Type'.

Next click on the Sphere button. Note that below there is a button called Keyboard Entry which has a + sign next to it. This is a roll down menu, if you click on it the + changes to a - and some options appear as shown above right. Note that the other menus like Creation Method and Parameters can be rolled up or down by clicking on them. Set the Radius in Keyboard Entry to 300.0 and press the create button. Some wire frame grids appear in some of the view windows. In order to see the view of the sphere properly click on the Zoom Extents button on the lower right of the screen which looks like this:



You should now see a wire-frame view of the sphere in all windows. In the Perspective window right click on the word Perspective and choose Smooth + Highlight. You should see the sphere rendered in this window using some default colour chosen by MAX.

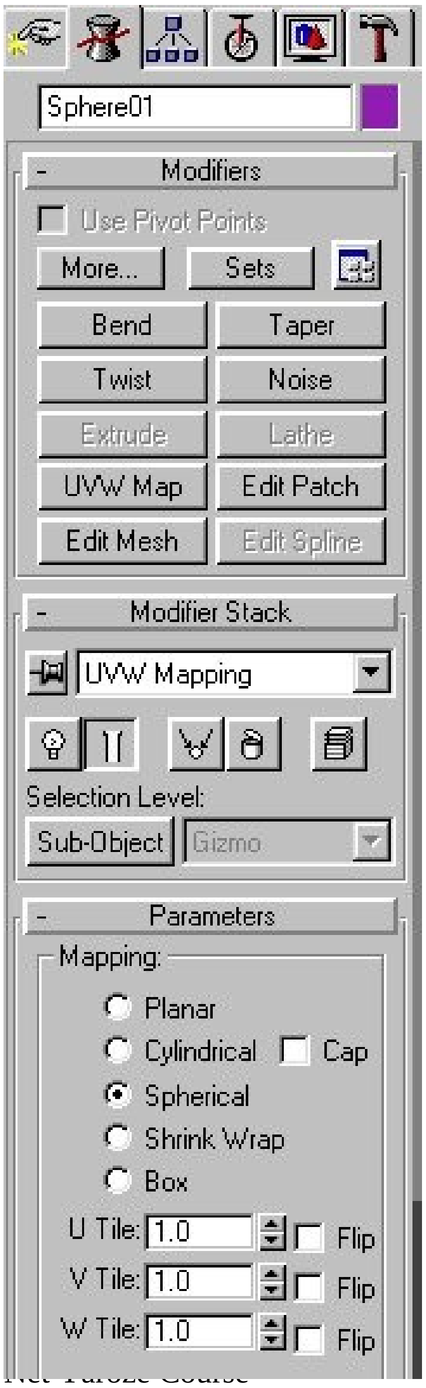
Texture the sphere with a bitmap

Create the bitmap.

For texturing on the PSX the maximum size of bitmap is 256x256 pixels per texture, but typically you will use smaller ones such as 16x16 with 4bit-colour resolution. Due to vagaries of MAX you will need to create two copies of your bitmap, one with true colour resolution and one with whatever resolution you are going to use eg: 4 bits per pixel – they should be the same size however. The reason is that MAX seems to only like true or high colour bitmaps, and won't display your texture using 4-bit colour. However using a true colour bitmap of the same size allows you to se how the texturing will turn out and generates the proper texture mapping coordinates. Put both copies on your floppy.

Set the UVW mapping co-ordinates.
In MAX select the modify icon which is next the create icon you already selected on the top right hand side of the screen and looks like this:

This will change the menu buttons available on the right of the screen and below this button. You need to set the texture mapping co-ordinates (ie: the UVW mapping) by clicking on the UVW Map button in the Modifiers section. As shown below:



Sphere01

Modifiers

[ ] Use Pivot Points

More... | Sets

Bend | Taper
Twist | Noise
Extrude | Lathe
UVW Map | Edit Patch
Edit Mesh | Edit Spline

Modifier Stack

UVW Mapping

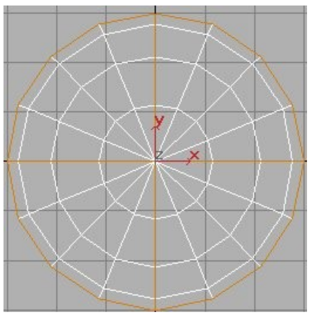
Selection Level:
Sub-Object | Gizmo

Parameters

Mapping:
( ) Planar
( ) Cylindrical [ ] Cap
(●) Spherical
( ) Shrink Wrap
( ) Box

U Tile: 1.0 [ ] Flip
V Tile: 1.0 [ ] Flip
W Tile: 1.0 [ ] Flip

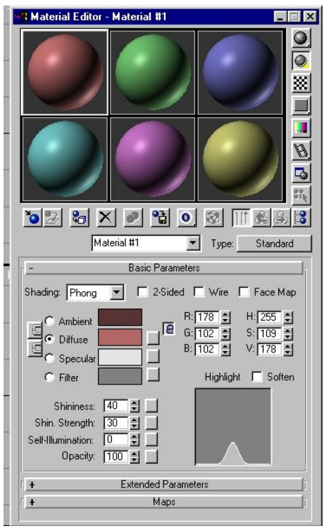
Under the parameters section click on Spherical, and you will now see a couple of orange circles drawn around your sphere. Notice that you cannot see the options at the bottom of the screen properly in the Alignment area. Also note that if the cursor is on a clear grey piece of background in the area below the Create and Modify buttons it changes shape from the usual cursor to a hand. Whilst the cursor is displayed as a hand if you press and hold down the left mouse button when you move the mouse up and down the whole area scrolls so that you can get to reveal the options under the Alignment area. Note that you could have also revealed the Alignment options by minimising the Modifiers menu by clicking on the Modifiers button. Either way, make the Alignment area of the Parameters section visible, and click on the Fit button. Note that the orange circles now follow the contours of the sphere closely. Below is the top view:



Map your texture.

To map the texture you have to activate the Material Editor by clicking on the button on the upper right of the screen that contains four coloured spheres and looks like this:

… which brings up the materials editing window which looks like this:


Material Editor - Material #1

Material #1 Type: Standard

- Basic Parameters

Shading: Phong ☐ 2-Sided ☐ Wire ☐ Face Map

○ Ambient
◉ Diffuse
○ Specular
○ Filter

R: 178 H: 255
G: 102 S: 109
B: 102 V: 178

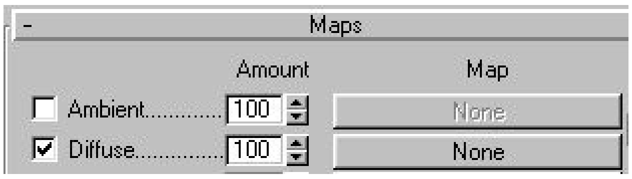
Highlight ☐ Soften

Shininess: 40
Shin. Strength: 30
Self-Illumination: 0
Opacity: 100

+ Extended Parameters

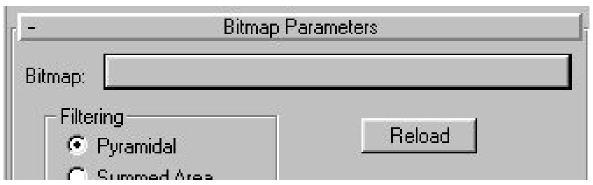
+ Maps

In this window click on the leftmost top sphere, and note that its border is now surrounded by a white border. Below the spheres you can set things like highlights etc. which are irrelevant for our purposes – we need to map our image to the sphere. Click on the Maps button at the bottom of this window and you will now see a new range of options. Click on the diffuse radio button so that it is checked as shown below:

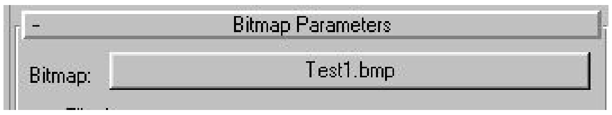


Then click on the None button, shown on the right in the same row as Diffuse under the Map column. This will popup a window which allows you to choose the kind of texture mapping to use. In this window double click on the Bitmap option which looks like:

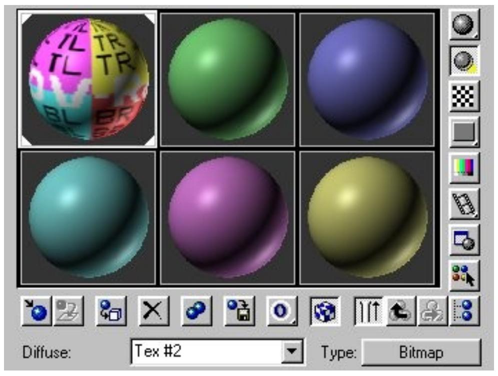
The popup window will disappear and you will now see bitmap parameters in the Material Editor window:



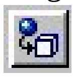
You now need to click on the grey untitled button next to Bitmap and a popup file sector window will appear. Choose the truecolour version of your texture file from your floppy and the blank button will now be labelled by your bitmap name:



… and the sphere on the upper left is now texture mapped with your bitmap :

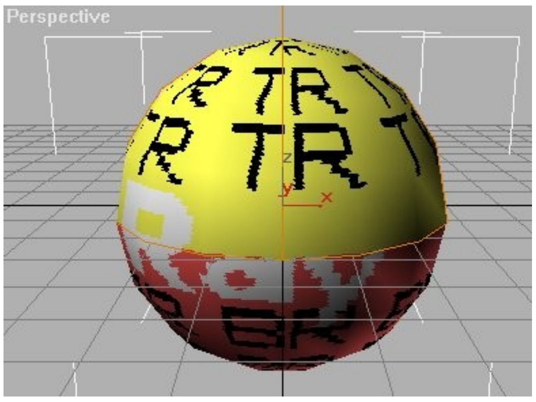


The next step is to assign the material you have specified to the selection (i.e. your sphere) by pressing the third button from the left under the spheres that looks like this:



… and finally to see how the texture looks on you model press the Show Map in Viewport button which is five buttons further on from the last and looks like this:

At last the sphere in the viewport should be shown appropriately textured:



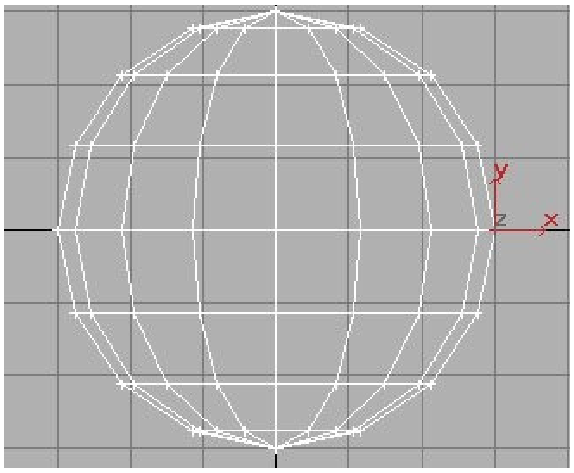
You could have a look at the different rendered views of the object by right clicking on the title of any view (eg: top or left) and choosing Smooth + Highlight instead of Wireframe.

It's a good idea to save your file in MAX format using File|Save, but to load it into R4 for final conversion to PSX format you need to export it to your floppy by choosing File|Export giving your file an appropriate name. The convert your 3DS file to RSD using 3DS2RSD. Don't forget to sort out your TIM file and run rsdlink on the RSD file to convert it to TMD format.

Tweak your sphere.

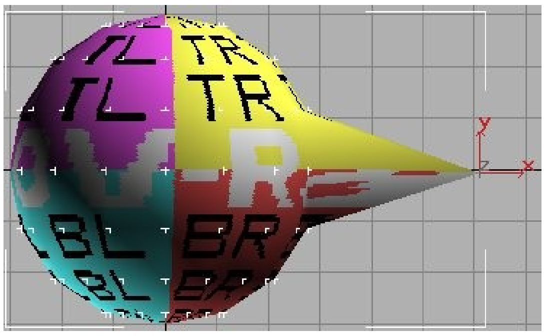
Now you will change the shape of your sphere by moving vertices. Select Modify

And then select the Edit Mesh button. The vertices of the sphere are now shown as little

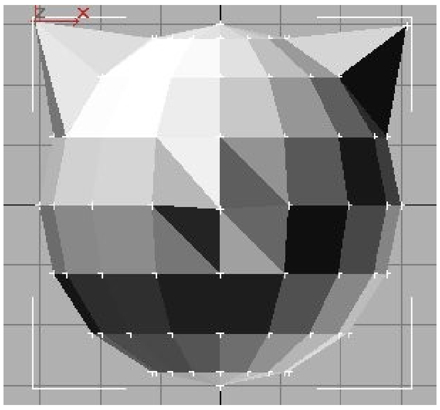


crosses. Click on the right most equatorial vertex in the Left view:

… and note that it shows axes in red at this vertex. Right click on the mouse button and select move. The cursor changes to a cross and if you press the left button down and move it to the right you pull out that vertex. Pull it out about 3 square to the right. You will have to do this by pulling it out a bit, clicking Zoom Extents on the lower right of the screen and pulling out the vertex more. If you right click on the word Left in the Left viewport and choose Smooth + Shading the image should look like this:



That was a nose, now lets add some ears. Click on the Front Viewport to select it, and set the Front viewport to Faceted + Highlight by right-clicking on the title Front, and pull out some

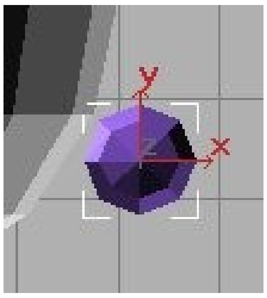


ears so that you model looks as above.

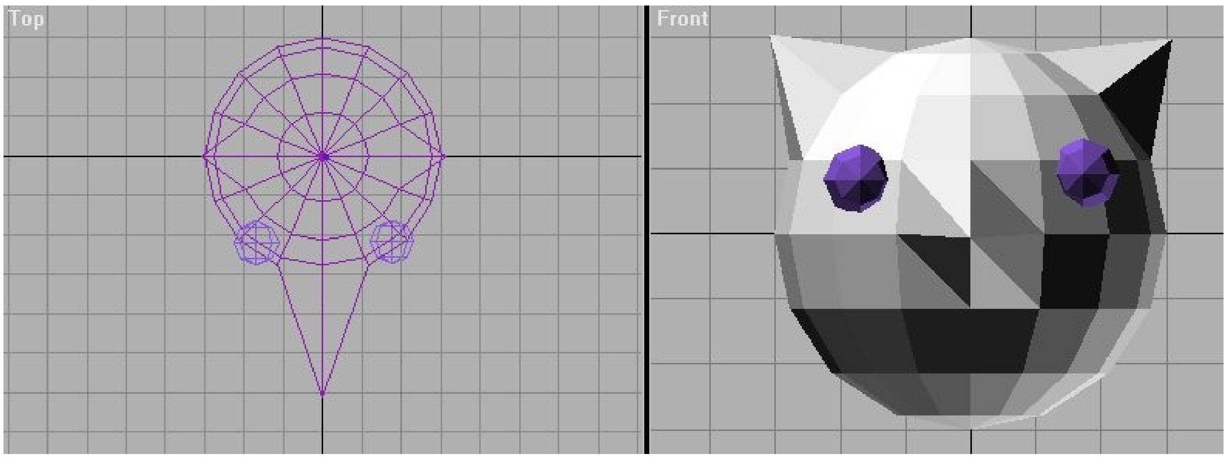
Note that under Selection Level you can choose to operate on faces and edges instead of vertices. Also under Modifiers there are other interesting options like edit patch and taper etc. Try experimenting with these at your leisure. Also note that sometimes wireframe views seem to omit lines – if your model look like this has happened in a particular view the set the view to Smooth + Highlights so see how it really looks. To get these ears more cat-like move the vertices forward (ie: towards the bottom of the screen) in the Top viewport.

Constructive solid geometry.

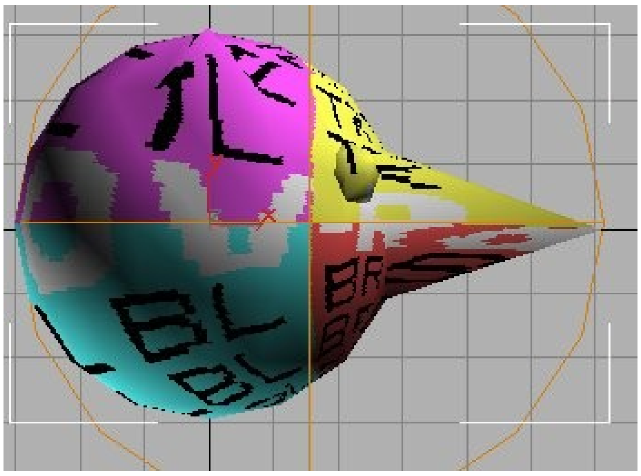
CSG is generally best avoided as it tends to generate loads of tiny polygons, however it’s such a standard technique so we’ll cover it.To add some eyes to our head we are going to use CSG. Click on create and click on sphere. Click on Parameters and set Segments to 8 – you will constantly be wanting to reduce the number of polygons in your model, this is a good way to do it, at the time of creation. This time create a sphere by pressing down the left mouse button in the Front window and dragging it a little way to size a sphere. MAX will give some arbitrary colour eg:



Before we place this eye lets copy it by choosing Edit|Clone (from the menu on the upper left of the screen) and clicking OK in the resultant popup window. Now if you left click on the eye and keep the button pressed down you will move the new copy around the screen. In the Front view place, an eye at the right position on the face, though note that the eye actually disappears into the head as it is constrained to move in the x plane. When you have it where you want move it down in the Top view until it half protrudes from the face, and do the same with the other eye until you have something like:



Now you are ready to merge the eyes with the face by creating a CSG union. Note that in any view you can click on an object to select it – it will be shown in white in wireframe view or with lines describing it's bounding box in other views. Select the head by clicking on it. Then click on the selection box on the upper left of the screen that currently says Standard Objects and choose Compound Objects instead. When you do this new buttons will appear below. Press the Boolean button in the Object Type section, and then click the Union radio button in the Parameters section. You are now ready to create a union object by clicking the Pick Operand B button in the Pick Boolean section and the click on one of the eyes to merge it with the head. Repeat the procedure for the other eye. The face has now lost its texture mapping to reinstate, this you need to perform UVW mapping again by clicking on the Modify Button, clicking on UVW Map, and choosing a radio button (eg: spherical) in the Parameters section below. Your Left view should now look something like this:



If you export your object to R4 and then to the PSX, it should save one RSD model.

Note that in general it is a good idea to save versions of your objects without CSG as once you've merged them you can't manipulate them later.

Building an articulated object.

An animated articulated object.

There are two basic ways to produce animation of articulated structures on the PSX. The sophisticated way, appropriate for eg: fighting games, is to create an number of .rsd models that correspond to the different parts of the object, set their co-ordinate systems appropriately to point to their parent structures, calculate and implement pivot points for rotation, and then animate them in real-time in your program either by calculation in your program or by using data generated by MAX or by using motion capture data. This approach generally uses less memory and is more flexible, but is a bit convoluted and not covered here.

Animation by model cycling.

An easier way, eg: for models that just have to walk around etc., is to produce a number of models for your character that correspond to key postures eg: left leg up, right leg up, sword thrust out etc., and have your program draw the appropriate model as required.

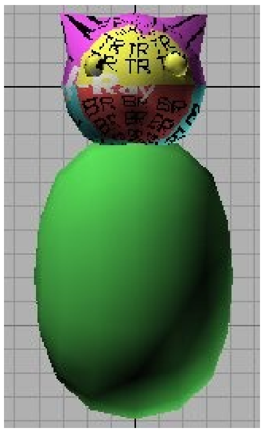
Building articulated structures in MAX and changing poses could assist this model-cycling approach, the 3D version of sprite cycling, although this approach is more expensive on memory. To build an articulated structure in MAX you build individual components, where each component is an atomic structure that may need to move/rotate individually from other components. For example the head we built in the previous section is atomic, because we built the head as a CSG object we can't later move the ears without re-modelling the object.

Build the articulated structure.

We need to build the head, body, arms and legs of the object and then link them together as an articulated structure. This time we are going to be as economic as possible with the amount of polygons we create.

Build the torso Activate the Front view by clicking on it. Then create a sphere of 500 radius, offset by –300 along the x axis, with segments set to 16. To make this more ellipsoidal select Modify and click on the More button to get a list of more modifiers, then choose Stretch. Under the Parameters windows for stretch set the stretch radio button for axis toY, and set Stretch value to 0.2. You should transform the sphere to an ellipsoid.

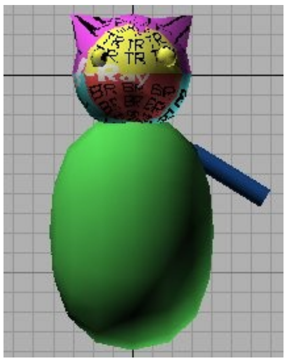
Then move your ellipsoid by clicking on the object (and make sure that the mode is set to Move by right clicking on the object first and selecting Move) and drag it into position below the head. Make sure that the torso is in an appropriate position by activating both the Top and Left views and dragging the object into position. The perspective view should look something like:



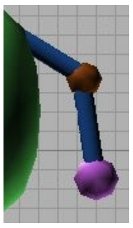
Build an arm.

Create a cylinder to make the upper arm segment by clicking on the create icon and then cylinder. In the Top view press the mouse button down and drag it a little way to define the radius of the cylinder. When you lift the button up, when you move the mouse the length of the cylinder changes. When you have the length you want left click once to set it. To get out of create mode click on the Select Object button on the upper left of the screen which looks like:

Select the cylinder by clicking on it and then right click and select rotate from the popup menu. By switching between the Top and Front views and right clicking to select Move or Rotate you should be able to get the arm into the correct position:



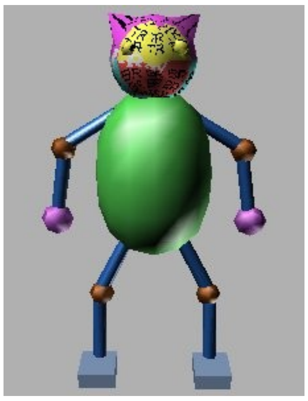
Create a sphere for the elbow and set its segments to 8 and place it in the correct position. To create the lower arm copy the cylinder of the upper arm by: selecting the cylinder choosing Edit|Clone from the menu of the upper left and clicking OK in the popup window that appears. The copy is at the same position as the original so if you select the upper arm and move it you actually move the copy and the upper arm remains. Use Move and Rotate to position the lower arm. Then add a sphere for the hand so your arm looks something like:



Build another arm.
To build the other arm we will just copy the first arm. We need to select all components of the arm by pressing down the Ctrl key and clicking on the objects in turn. They should all have white lines around them to show they are selected. Then select Edit|Clone from the menu of the upper left and click on OK in the popup window that appears. Now move your copy of the arm from the right of the body to the left. To get the new arm the right way round move and rotate it in the appropriate views.

Build the legs.

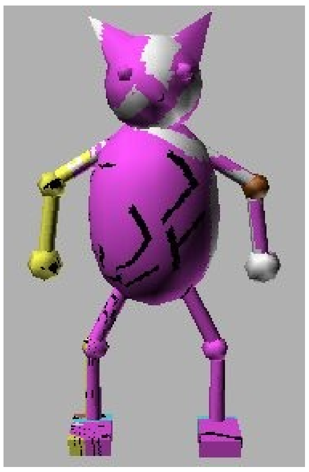
Build a leg by copying an arm (but not the hand) and adjusting the components until it looks OK. Create a box for the foot, then copy and mirror the whole leg to finish your object, and

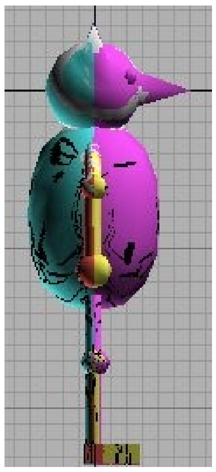


save it to disk.

Play about.

You can texture the whole object by selecting all elements (ie: hold down control and click on each object), choosing Modify and UVW map, then activate the material editor and apply the texture to get:





(Actually I also tweaked the ears here…)

Linking your articulated structure.

You now know enough to be able to create and texture models and manipulate them by moving and rotating components to produce different poses for animation. However it can be tricky manipulating your object without linking the articulated structure. The easiest way to manipulate your object is to link the components together and then specify where the pivot points for rotation should be for each components.

Linking.

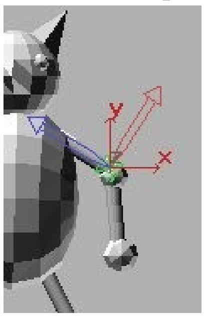
Links consist of specifying parents and children. If a parent is moved or rotated then so will all its children, however a child can be manipulated separately from a parent ie. If you rotate the torso the connected arms and legs should also rotate, if you rotate the upper arm you expect the connected elbow, lower arm and hand to rotate also. You will link your objects by starting with the torso and going down the limbs. Click on the link button found on the upper left toolbar underneath the Group menu:

Now click on the upper left arm to select it and the cursor will change shape to the icon shown above. To link the upper arm to the torso left click and drag the cursor over the torso and let go. White lines breifly flash around the torso to indicate that a link has been made. To confirm the link has been made select the torso, set rotate and rotate the torso. As it rotates the upper arm should rotate with it. Select Edit|Undo a couple of times to undo your test rotation. Now work your way down the arm by linking the elbow to the upper arm, the lower arm to the elbow, and the hand to the lower arm. Repeat the same process for the other arm and the legs. Test your linkage by rotation and undo the rotation afterwards.

Setting pivot points.

Depending on how you created the cylinders used for limb segments the centre of rotation for individual segments may not be in the correct place. You can set this by placing the pivot points yourself by first clicking on the Hierarchy button which is on the upper right immediately to the right of the modify button:

Then click in the Affect Pivot Only button under Adjust Pivot. You can now select a component to show it's pivot point which is shown as a green wireframe box with red and blue arrows pointing out of it.



In the example above the pivot for the upper arm is shown to be at the elbow end which is wrong so we would need to move the pivot point to the shoulder end which will then make sure that rotation is around the shoulder. Set the pivot points for all cylinder segments and the feet, and get out of the Hierarchy editing mode by clicking back on the Create button.

You are now ready to manipulate your object and the general rule is to apply rotations starting from the highest parent in the hierarchy (ie the torso) and then work down the branches to the lowest children (eg hands and feet).

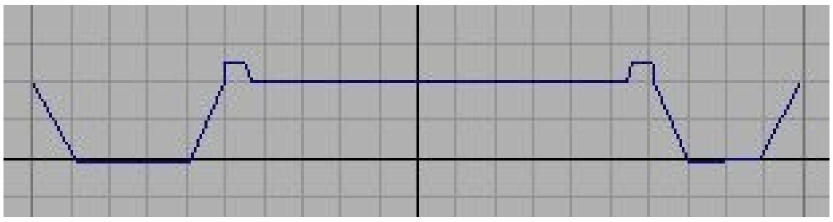
It’s a good idea to save a separate copy of your object in this format. If you are going to export your object to R4 you will have to use CSG to merge all the components of the object in a particular pose.

Using spline objects and extrusion.

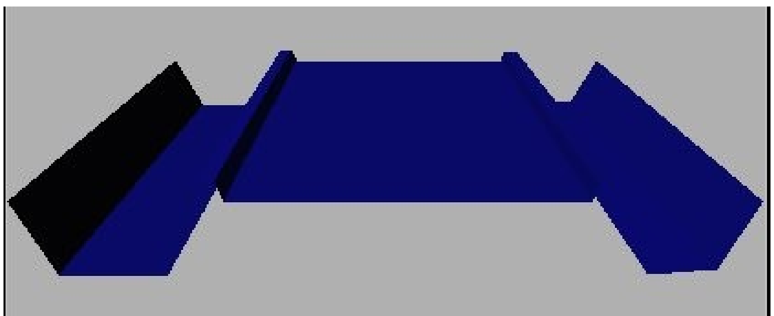
It is easier to make some objects (eg: landscape) using line drawing and extrusion rather than 3d primitives and transformations. We will briefly look at making components for a track like Robert Swan’s one in his race demo. Start by clicking on the Create button and then click on the Shapes button which is underneath the Modify button and looks like this:

Before drawing your shape set the scale of you window correctly. You have decided that the width of the track is going to be 2000 units wide. Move the mouse ten units along the grid in the x direction and look at the numbers at the bottom of the screen that change as you move the cursor around the screen. You will need to scale the window in or out using the Scale All button on the lower right of the screen:

Scale the window until the position of ten units from the origin along the x-axis is about 1000 units. The buttons below the Shapes button now allow you to create 2d primitives like lines and circles etc. Select Line and draw the horizontal profile of the track by moving the mouse and clicking to create lines, right click to end line construction:



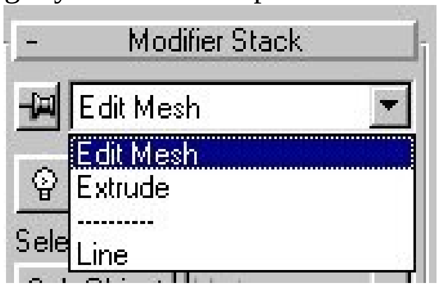
Now click Modify and then click Extrude. Set the Amount under Parameters to 2000 and your shape is extruded out for 2000 units. However the normals are facing the wrong way so you don’t see much in the perspective window. To flip the normals Choose edit Mesh under Modify and set the selection on the right below Modifier Stack to Faces rather than Vertex. Then select all the faces in the object by pressing the left button down and creating a bounding box that contains the whole object. Then scroll all the way to the bottom in the



button area to the right and click Flip Normals. The perspective viewport should look something like this:
You could save this as a segment of track for straight road.

Making corners.

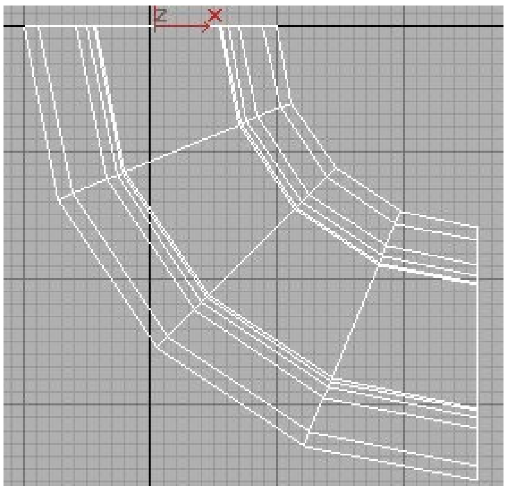
To make a corner segment from your track you could start by making it longer. All the transformations that are applied to an object are shown in the Modifier Stack under Modify. At the moment the your selection list says Edit Mesh but if you click on the arrow to the right you can set the previous transformations in the history of the object:



You could go back to the extrude transformation or even the original line transformation and change parameters by selecting either of these. However if you do that it may conflict with subsequent operations. In this case we will delete the edit mesh transformation by selecting Edit Mesh from the selection list and the clicking on the Remove Modifier icon below it:

Now Extrude is shown and we are back to the Extrude parameters. Set the extrusion Amount to 4000, and the segments amount to 4 under the Parameters. The Top or Bottom view of the track as a wireframe should show that you have divided the track into four segments:

Now you are ready to make the curve by selecting the Bend button under Modify, and setting the Angle under Bend Angle to 90 degrees. Now flip the normals as described above and you have a 90 degree bend:



Getting data from the PSX to the PC.

Getting a screenshot from the PSX (method 1)

First you need to include (or copy and paste) the following function in your program:

```
// function prototype
void StoreScreen (void);
  // stores screen as bona fide TIM file;
  // use siocons Dsave command to transfer directly to PC file
 // This stolen directly from sample\japanese\sscreen

void StoreScreen (void)
{
//#if STORING_SCREEN
      u_long* destination;
      int x, y, w, h;
      RECT rect;

      destination = (u_long *) 0x80090000;

      x = y = 0;        // top left of frame buffer

      w = SCREEN_WIDTH;
      h = SCREEN_HEIGHT;

      *(destination+0) = 0x00000010;            /* ID */
      *(destination+1) = 0x00000002;
/* FLAG(15bit Direct,No Clut) */
      *(destination+2) = (w*h/2+3)*4;           /* pixel bnum */
      *(destination+3) = ((0 & 0xffff) << 16) | (640 & 0xffff);
                                    /* pixel DX,DY: at 640, 0 */
      *(destination+4) = ((h & 0xffff) << 16) | (w & 0xffff);
                                    /* pixel W,H */

      // NO CLUT since 16-bit mode used

      rect.x = x;
      rect.y = y;
      rect.w = w;
      rect.h = h;
      DrawSync(0);
      StoreImage(&rect, destination+5);
      printf("\n\nPress [F10][F4] for dsave, to get screen picture\n");
      printf("Dsave[0]: filename %08x %x\n\n\n", destination, (w*h/2+5)*4);
      DrawSync(0);
      VSync(0);
//#endif
}
```

Note that `SCREEN_WIDTH and SCREEN_HEIGHT` must be defined in your program. Then you need to call StoreScreen from an appropriate point in your program and exit your program. This function copies the screen buffer from video memory to conventional memory starting at address 80090000 overwriting your original data, so you have to exit immediately afterwards calling or your program will crash. Eg: call it:

```
      if (PADstatus & PADtriangle){DrawSync();StoreScreen();exit(0);}
```

As shown above it’s advisable to call DrawSync() to make sure drawing has ended. This then prints out the starting address and size of your image on the PC monitor and restores the siocons prompt:

```
Press [F10][F4] for dsave, to get screen picture
Dsave[0]: filename 80090000 25814


ResetGraph:jtb=8004829c,env=800482e4
ResetGraph:jtb=8004829c,env=800482e4
PS-X Control PAD Driver  Ver 3.0
addr=8004d724

Connected CIP ver2.0
Communication baud rate 115200 bps
OK
>>
>>
```

At which point you need to press F10 and then F4, which brings up the dsave prompt where you type in your filename (ending in .tim) followed by the starting address in memory and the file size. Feedback is given on the screen whilst saving and you are returned to the siocons prompt on completion:

```
>>
Dsave[8]: dump.tim 80090000 25814
DSAVE dump.tim 80090000 0/25814
DSAVE dump.tim 80094000 4000/25814
DSAVE dump.tim 80098000 8000/25814
DSAVE dump.tim 8009c000 c000/25814
DSAVE dump.tim 800a0000 10000/25814
DSAVE dump.tim 800a4000 14000/25814
DSAVE dump.tim 800a8000 18000/25814
DSAVE dump.tim 800ac000 1c000/25814
DSAVE dump.tim 800b0000 20000/25814
DSAVE dump.tim 800b4000 24000/25814
26(sec)
>>
>>
```

The final step is to convert your .tim file to a .bmp file so that it can be loaded by web browsers etc, which you do by loading your .tim into TIMutil and saving it as a .bmp.

Storing all your data in one file

If you want to distribute your work without giving away your source and artwork (bitmaps, models, sounds), you can use the above technique.

What you need to do is work out the size of memory that your models, textures, and sounds are loaded into which you can find out by looking at the PC screen to see the last address to which your data is loaded. In the example below the last address used for data is 800cc817 – for convenience we'll round it up to 800d0000. Just to be on the safe side, use the calculator to calculate the size of memory taken up and make sure you put the calculator into hex mode before typing anything in. The calculation is 800d0000 – 80090000 which is 40000 in hex.

```
.\textures\sky.tim  address:800bc5f8-800cc817 size:010220  010220:   26sec.
main [ .text] address:80140000-80143cff size:003d00  003d00:   13sec.
```

As soon as you have loaded your program, exit it, and save the memory to file to disk using dsave. Eg: Press F10 followed by F4 which gets you to the dsave prompt then give the filename address and size, in this example:

```
Dsave[9]: data 80090000 40000
```

When that's finished it remains to edit the auto file to load up this memory dump in one go, which should look like this:

```
local dload data     80090000
local load main
go
```

Therefore to distribute your program so that its playable but doesn't give away models, sounds, or bitmaps, you only have to have:

The auto file as above
The compiled version of your main program (not the main.c)
The memory data file (called 'data' above).

Getting a screenshot from the PSX (method 2)

Odd cases occur when the data transfer between the PSX and the PC becomes corrupt for one reason or another, and you may find that the picture either won't convert from the .log file to the .raw file or appears jumbled when viewed on the PC file after conversion. If this happens, just attempt the procedure from scratch. This method is more prone to corruption that method 1.

Preparing for grabbing screenshots

To do this, copy the supplied files scrngrab.h and scrngrab.c into the same directory as the rest of your code. These files will be supplied to you. You have to edit your makefile to get these files to compile. Simply add scrngrab.o to the fifth line as follows:

```
OBJS    =    loadtex.o main.o pad.o
```

Would become:

```
OBJS    =    scrngrab.o loadtex.o main.o pad.o
```

You have to include the header file scrngrab.h into your main code, so at the start add:

```
#include "scrngrab.h"
```

now all you have to do in your code itself is add a trigger to call the dumpscreen() function. This could be a function of time, but the most useful trigger is to attach it to a joypad button. For example, in ProcessJoyPad():

```
// grab screen when you press L2+R2 together during game
if ((PAD& PADL2) && (PAD& PADR2)) dumpScreen(0, 240, 319, 479);
```

As you might be able to guess, the four arguments of dumpscreen() are top left corner co-ordinates of area you want to grab, and bottom right co-ordinates. In this case it will grab the contents of the bottom buffer one. What will happen is when triggered, the dumpscreen() function outputs the screen contents to the PC display. To get a copy of this output, add the following line to your auto file just before the last line, local load main:

```
local log grab
```

What this line does is record every bit of output from the PSX to the PC in a file called grab, which will include the screen information. Your program is now ready and you can recompile and run it.

Grabbing the screen and converting it to a bitmap

After running the program, at whatever point you want to grab the screen press whatever it was that you designated as the trigger (in the example above, L2 and R2 together). The program will appear to freeze on the PSX, and loads of gibberish will appear on the PC. This is normal, and after about 20 seconds the game will restart as normal. You can now stop your program and exit Siocons on the PC with the ESCAPE key. The contents of the screen grab are in the log file (in this case 'grab'). There is a utility to extract that screen info from the log file and it is called log2raw and is used as follows from DOS:

```
Log2raw grab raw.raw
```

The first argument is the log file name, and the second argument is the .raw graphics file you want to output it is. Note that this only will convert the first lot of graphics information it comes across in the log file, so if you grab more than one image from the game, you will only be able to extract the first. You now have your graphics file you can view on a variety of art packages. What follows is a way to convert it into a bitmap using Paint Shop Pro.

1. Load Paint Shop Pro,
2. Select Open from the File menu,
3. Select 'all types' in the format type menu selector.,
4. Select your file raw.raw,
5. A new dialog box will appear. Set the width of the image to 320, and the height to 240 and click OK,
6. Your game image should now appear on screen (barring corruption as mentioned earlier),
7. Save the image in whatever file format you want.

To prepare for grabbing another image, delete the log file so that next time you run your game it doesn't append any new screen grabs to the end of the old one, and proceed as before.

To remove the screen grabbing routine, delete the scrngrab.* files from your directory, remove the #include and dumpscreen() trigger from your code, remove scrngrab.o from your makefile, and finally delete the local log grab line from your auto file.

----- FIN -----

COM3311 Programming Interactive Graphical Systems

Rotation – revisited.

The original formulation.

In the 3D tutorial the first RotateMatrix function worked by concatenating all rotations applied to the player into the player's matrix. The advantage of this formulation is that when you perform a series of rotations around the x,y, and z axes in whatever order, the rotation is always around the players appropriate local axis. The disadvantage is that over time precision errors accumulate and eventually shear or scale the player. Note that it is not the model co-ordinates that are corrupted only the matrix.

The driving game formulation.

The second formulation in the 3D tutorial (used for cswk 1) stores a vector for the player's rotation and simply adds up successive rotations in each of the x, y, and z, to the vector. This only works for rotation in the plane (ie around one axis) and is fine for a simple driving game, but doesn't work for situations where you want to have arbitary rotation around more than one axis (eg: a flying game). The reason for this is because the order of rotation counts (eg: roty(90), rotx (90) ,rotz(90) != rotx (90) ,rotz(90), roty(90)), the same set of rotations in a different order produce a different final orientation).

Conditioning the player's matrix.

There are a number of possible approaches to dealing with this problem. The most obvious one is to use the original formulation above and 'condition' or normalise the matrix to remove the accumulated errors.

By definition any row or column in a rotation matrix should be a unit vector (ie: that sqrt(x*x+y*y+z*z)= 1 or ONE in Yaroze terms), and the three rows or columns should be orthogonal to each other (ie: all forming a 90 degree angle to each other) – both these conditions are clearly true for the identity matrix.

Thus you can simply normalise a matrix row or column by finding the length of the vector and dividing each component by the length, eg:

```
length=sqrt(x*x+y*y+z*z)

x=x/length; y=y/length; z=z/length;
```

Unfortunately using the floating point sqrt function is very expensive, eg: if you normalise all 3 rows and 3 columns on every frame your vsync value goes from say 120
to 1600.

Using a lookup table.

You can probably imagine a number of ways to deal with the slow sqrt problem and the method you use will be strongly influenced by ‘what needs to be rotated and how’ in your game. A good general approach (in terms of speed and precision) is to use an integer lookup table that contains the numbers 0 – 4096 squared. The file squared.h contains this table and look like:

```
int squared[] =
 {
 0,
 1,
 4,
 9,
 16,
etc.
```

We can then define our own square root function to use this table:

```
int mysqrt(int length)
{
int i,j;

j=4096;
if (length >squared[j]) return j+1;
else while(length < squared[j]){
          j--;
     }
     return j;
}
```

This function will return 4097 if length is greater than 4096 squared, otherwise it just goes down through the table until it finds the first value that is smaller than length, and returns its index. This is like a floor function, for example if you asked for mysqrt(35) or mysqrt(25) it should return 5…

The use of the lookup table also gives us an added benefit in that instead of having to square terms we can just look them up which is quicker. We have to remember though that the values in the rotation matrix can be positive or negative so we have to apply the ABS macro to force them to be positive (remember that -n*- n = n*n). One final refinement is that we don’t have to normalise both rows and columns on every frame so in the example a flag is used to set the normalisation of either rows or columns on alternate frames.

This formulation is generally fairly stable and only costs 2-3 vsyncs per frame when applied to the car in step10 (but there will be some situations where it is still not satisfactory…)

Getting a screen dump from the PSX.

First you need to include (copy and paste) the following function in your program:

```
// function prototype
void StoreScreen (void);
     // stores screen as bona fide TIM file;
     // use siocons Dsave command to transfer directly to PC
file
     // This stolen directly from sample\japanese\sscreen

void StoreScreen (void)
{
//#if STORING_SCREEN
     u_long* destination;
     int x, y, w, h;
     RECT rect;

     destination = (u_long *) 0x80090000;

     x = y = 0;          // top left of frame buffer

     w = SCREEN_WIDTH;
     h = SCREEN_HEIGHT;

     *(destination+0) = 0x00000010;          /* ID */
     *(destination+1) = 0x00000002;
          /* FLAG(15bit Direct,No Clut) */
     *(destination+2) = (w*h/2+3)*4;         /* pixel bnum */
     *(destination+3) = ((0 & 0xffff) << 16) | (640 & 0xffff);
                              /* pixel DX,DY: at 640, 0 */
     *(destination+4) = ((h & 0xffff) << 16) | (w & 0xffff);
                              /* pixel W,H */

     // NO CLUT since 16-bit mode used

     rect.x = x;
     rect.y = y;
     rect.w = w;
     rect.h = h;
     DrawSync(0);
     StoreImage(&rect, destination+5);
     printf("\n\nPress [F10][F4] for dsave, to get screen
picture\n");
     printf("Dsave[0]: filename %08x %x\n\n\n", destination,
(w*h/2+5)*4);
```

```
     DrawSync(0);
     VSync(0);
//#endif
}
```

Note that `SCREEN_WIDTH and SCREEN_HEIGHT` must be defined in your program. Then you need to call StoreScreen from an appropriate point in your program and exit your program. This function copies the screen buffer from video memory to conventional memory starting at address 80090000 overwriting your original data, so you have to exit immediately afterwards calling or your program will crash. Eg: call it:

```
     if (PADstatus & PADtriangle){StoreScreen();exit(0);}
```

This then prints out the starting address and size of your image on the PC monitor and restores the siocons prompt:

```
Press [F10][F4] for dsave, to get screen picture
Dsave[0]: filename 80090000 25814


ResetGraph:jtb=8004829c,env=800482e4
ResetGraph:jtb=8004829c,env=800482e4
PS-X Control PAD Driver  Ver 3.0
addr=8004d724

Connected CIP ver2.0
Communication baud rate 115200 bps
OK
>>
>>
```

At which point you need to press F10 and then F4, which brings up the dsave prompt where you type in your filename (ending in .tim) followed by the starting address in memory and the file size. Feedback is given on the screen whilst saving and your are returned to the siocons prompt on completion:

```
>>
Dsave[8]: dump.tim 80090000 25814
DSAVE dump.tim 80090000 0/25814
DSAVE dump.tim 80094000 4000/25814
DSAVE dump.tim 80098000 8000/25814
DSAVE dump.tim 8009c000 c000/25814
DSAVE dump.tim 800a0000 10000/25814
DSAVE dump.tim 800a4000 14000/25814
DSAVE dump.tim 800a8000 18000/25814
DSAVE dump.tim 800ac000 1c000/25814
DSAVE dump.tim 800b0000 20000/25814
DSAVE dump.tim 800b4000 24000/25814
```

```
26(sec)
>>
>>
```

The final step is to convert your .tim file to a .bmp file so that it can be loaded by web browsers etc, which you do by loading your .tim into timutil and saving it as a .bmp.

Protecting your artwork.

If you want to distribute your work without giving away your source and artwork (bitmaps, models, sounds), as I have done with most of the student work to be found on my Yaroze site, you can use the above technique (well it worked in 5 out of 7 cases but not for Casper's or Olly's programs – I don't know why, let me know if you crack it).

What you need to do is work out the size of memory that your models, textures, and sounds are loaded into which you can find out by looking at the PC screen to see the last address to which your data is loaded. In the example below the last address used for data is 800cc817 – for convenience we'll round it up to 800d0000. Just to be on the safe side, use the calculator to calculate the size of memory taken up and make sure you put the calculator into hex mode before typing anything in. The calculation is 800d0000 – 80090000 which is 40000 in hex.

```
.\textures\sky.tim  address:800bc5f8-800cc817 size:010220  010220:   26sec.
main [ .text] address:80140000-80143cff size:003d00  003d00:   13sec.
```

As soon as you have loaded your program, exit it, and save the memory to file to disk using dsave. Eg: Press F10 followed by F4 which gets you to the dsave prompt then give the filename address and size, in this example:

```
Dsave[9]: data 80090000 40000
```

When that's finished it remains to edit the auto file to load up this memory dump in one go, which should look like this:

```
local dload data     80090000
local load main
go
```

Therefore to distribute your program so that its playable but doesn't give away models, sounds, or bitmaps, you only have to have:

The auto file as above
The compiled version of your main program (not the main.c)
The memory data file (called 'data' above).

Mdxlib

Eg: step15

Edit makefile (in windows) rplace 2dlib.o with mdxlib.o

InistialiseGraphics3d

Initiaiseworld and Initialise maps are game dependant – eg ground_tim etc

Render prepare is d dependant move render prepare to render prepare3d

For 2d graphics screen is set up with 0.0 at upper lrft
For 3d graphics screen is set up with 0.0 as centre!

You need to use Mdxlib to integrate 2d and 3d graphics. The example merges 2Dstep04 and 3dstep13