I have designed this introductory tutorial as a first tutorial for a 3rd year games course I am running in February 1998. Whilst navigating my way through the PSX documentation, the demo programs, and the news groups I often wished (like I suspect many Yaroze newcomers do!) that I had a tutorial like this which would just tell me the bits I needed to get me programming in 3D from the bottom up rather that struggling with incomprehensible functions and sophisticated but large example programs.

So I hope this will be of use to others of you out there and I only ask that you protect my copyright to this document, and if you use it for teaching that you let me know.

The document is quite long and in a draft state but tells much of what you need to know to get going in 3D. It does not deal with sprites at all, which will be the subject of a further tutorial I may make available here but note that Ira Rainey has already are some good tutorials on 2D which can be found at ~shadow/ftp on the Yaroze server.

I welcome your feedback and (constructive!) criticism so if you wish to contact me, email me at p.passmore@mdx.ac.uk.

UPDATE 1/12/97

#defined functions now bracketed to be ANSI C (& Code warrior compatible) - thanks to US Yarozer Matthew Hulett for pointing this out.

NTSC users note that the video mode should be set to NTSC rather than PAL and the screen width and height set to an appropriate NTSC mode (eg: 320x240) - see page 10.

Step 3 now includes an updated main function in the text, and function prototypes.

UPDATE 9/12/97

Variable matTmp defined as a matrix instead of a pointer to a matrix in functions AdvanceModel and RotateModel. Thanks to Craig Graham and Stefano Provenzano for catching this one.

Tutorial programs have been updated accordingly.

COM3311 Programming Interactive Graphical Systems

Net Yaroze Tutorial

INTRODUCTION

STEP 1. HELLO WORLD ON THE PC.

STEP 2. HELLO WORLD ON THE PSX AND DOUBLE BUFFERING.

STEP 3. LOAD AND VIEW A 3D MODEL.

STEP 4. TRANSFORMING A 3D MODEL.

Making the model move.

Understanding the MATRIX struct.

The specification of angles for rotation.

Rotating an object.

STEP 5. ODDS AND SODS.

A joypad function.

Hierarchical coordinate systems.

Approximating the frame rate.

STEP 6. PRECISION PROBLEMS.

Precision problems already.

A new rotate function.

STEP 7. TRANSLATING IN THE CORRECT DIRECTION.

Making the car move in the direction it is pointing.

STEP 8. TEXTURE MAPPING.

Introduction to texture mapping.

Loading a texture into memory.

Making your own texture files.

STEP 9. 3D MODELLING

Making your own model.

Converting from dxf to rsd.

Assigning texture to the quadrilateral.

Creating a tmd file from an rsd file

STEP 10. BUILDING A WORLD.

Building a road to drive around on.

The world struct.

Creating a world map.

The AddModelToWorld function.

Allow a couple of views.

PROJECT: BUILD THE REST OF THE TRACK AND GIVE THE CAR SOME DYNAMICS.

Tips.

INTRODUCTION

Acknowledgements

Firstly thanks go to Paul Holman and Sarah Bennet at Sony Computer Entertainment Europe (SCEE) for giving us a bunch of Yarozes to play with (erm... I mean study in a rigorous academic manner). Secondly thanks go to Sean Bulter who set up our first games module and developed the original versions of much of the code here before moving on to the games industry. Thanks are also due to a project student of mine Ian Frost who worked on one of the original Yarozes six months before the product was launched in Europe and is now also working in the industry. A year ago nearly all demos came from Japan but Ian managed to struggle through (in spite of ignoring my practical advice to try and find a Japanese girlfriend). Lastly thanks go to Lewis Evans (of SCEE) for reading through this tutorial and clearing up a couple of inaccuracies.

Aim.

The aim of this tutorial is to get you quickly up and running using the PSX and its library functions. The library seems obscure compared to say OpenGL, because it is closely tied to the hardware architecture. However for the most part you can quickly get into 3D graphics programming if you take the basic graphics initialisation, and object manipulation and other functions defined from step 2 onward on faith. Be thankful you haven't had to discover these functions by yourself! It just turns out there are various curious things you just have to do to get the machine to play ball. As you become more competent and curious you should look into how the PSX functions I have shielded you from in these routines really work, and then come back and educate me. I am not an expert PSX programmer and am bound to program in a less that efficient manner, however you will get demonstrable results from this code. Enough said.

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 fils 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 librarys, 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 extention 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.

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 syncronisation 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 incomprehesible - 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 `InitialiseGraphics` below) we have to set it's length and *org variables, (if fact `InitialiseGraphics` 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 overlaping 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 will discuss solutions to this later). 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 prescision 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        othWorld[2];
```

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

```
GsOT_TAG    otWorld[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 `InitialiseGraphics` 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          out_packet[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 packetArea[2][MAX_NUMBER_PRIMITIVES * SIZEOF_PRIMITIVE]
```

where the maximun size that a primitive can be is 24 (ie: set `SIZEOF_PRIMITIVE` to 24).

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 `PadRead()` function) it and test if the select button has be pressed in which case the following evaluates to true:

```
(PADstatus & PADselect)
```

The program listing follows and is bristling with comments. Most of the PSX specific stuff is hidden in the InitialiseGraphics and RenderWorld functions. You can get by for now with only a cursory understanding of most of these PSX functions. For further information check out the manuals, and over the course of your time using the PSX make a point of noting where they are incomplete or inaccurate, so we can give some feedback to Sony about them.

```
/************************************************************
```

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

#include <stdio.h>
#include <stdlib.h>
#include <libps.h>
#include <string.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 othWorld[2];
// And we need Two Ordering Tables, one for each buffer
GsOT_TAG	otWorld[2][1<<ORDERING_TABLE_LENGTH];
//We also allocate memory used for depth sorting, a block for each buffer
PACKET		out_packet[2][MAX_NO_PACKETS];
// we need a variable to store the status of the joypad
u_long PADstatus=0;

// this function initialises the graphics system
void InitialiseGraphics()
{
	// 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, GsINTER|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
	othWorld[0].length = ORDERING_TABLE_LENGTH;
	othWorld[1].length = ORDERING_TABLE_LENGTH;
	// 2. Associate them with an actual ordering table
	othWorld[0].org = otWorld[0];
	othWorld[1].org = otWorld[1];
	// 3. initialise the World Ordering Table Headers and Arrays
	GsClearOt(0,0,&othWorld[0]);
	GsClearOt(0,0,&othWorld[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*)out_packet[currentBuffer]);
            // clear the ordering table
            GsClearOt(0, 0, &othWorld[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 ordering table to black
            GsSortClear(0, 0, 0,&othWorld[currentBuffer]);
            // register request to draw ordering table
            GsDrawOt(&othWorld[currentBuffer]);
}

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
      InitialiseGraphics();
      while(1){
            // read the joypad
            PADstatus=PadRead();
            // 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;
}
```

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 PAL mode by checking the manual), and see the effect of having a PACKET array length of 0 - have a go...

STEP 3. Load and view a 3D model.

In this step we fast forward into the world of 3D graphics and you will get to display a 3D object on the screen, as well as your text message. This involves getting into more curious

PSX functions and code, however by the end of it you will have the basic essentials to get into games programming and investigate implementation of game details that become less and less PSX specific. Before you get there though, we will have to consider, loading up a 3D object, setting up lights, and setting up a viewing system - which is a lot to cover in one step. Hold on to your hat.

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. 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. 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\car\car01.tmd  80090000
local load main
go
```

This assumes that the model car01.tmd is in the subdirectory data\car\ 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\car\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 CAR_MEM_ADDR  (0x80090000)
```

The naming convention we use is to give some meaningful name followed by `_mem_addr` so we know it refers to the memory address of a tmd model. 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.

Function prototypes.

It is standard practice to define function prototypes before defining the function. Usually these will be stored in an appropriately named header file along with #defined constants, global variables and structs. The approach I have taken here is to define everything (except the pad files) in one program as this is easier to understand for the beginner. However sooner or later you should migrate families of functions and their related header components to appropriately named .c and .h files - there comes a time when your program gets too long to edit easily. For the moment though just remember that the convention we adopt is to define global variables, constants, and structs at the top of our program, then we will define function prototypes immediately afterwards. The function prototypes we need to define for this step are as follows:

```
/************* FUNCTION PROTOTYPES ******************/
void AddModelToPlayer(PlayerStructType0 *thePlayer, int nX, int nY, int nZ,
                  unsigned long *lModelAddress);
void DrawPlayer(PlayerStructType0 *thePlayer, GsOT *othWorld);
void InitialiseLight(GsF_LIGHT *flLight, int nLight, int nX, int nY, int
                        nZ,int nRed, int nGreen, int nBlue);
void InitialiseAllLights();
void InitialiseView(GsRVIEW2 *view, int nProjDist, int nRZ,
                                      int nVPX, int nVPY, int nVPZ,
                                      int nVRX, int nVRY, int nVRZ);
void InitialiseGraphics();
void RenderWorld();
/************************************************/
```

and you should insert these prototypes after your constant, global, and struct definitions and before your function definitions.

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 slap 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         gsObjectHandler;
       GsCOORDINATE2   gsObjectCoord;
}PlayerStructType0;
```

We call the structure PlayerStructType0 so we can enhance the structure later to include more game related variables and develop `PlayerStructType1, PlayerStructType2` etc.

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
PlayerStructType0 theCar;
```

Initialising the player struct.

We define a function called `AddModelToPlayer` 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:

```
void AddModelToPlayer(PlayerStructType0 *thePlayer, int nX, int nY, int nZ,
      unsigned long *lModelAddress)
{
      //increment the pointer to 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, &thePlayer->gsObjectCoord);
      // increment pointer twice more - to point to top of model data
      //(beats me!)
      lModelAddress++;  lModelAddress++;
      // link the model (tmd) with the players object handler
      GsLinkObject4((unsigned long *)lModelAddress,
                        &thePlayer->gsObjectHandler,0);
      // Assign the coordinates of the object model to the Object Handler
      thePlayer->gsObjectHandler.coord2 = &thePlayer->gsObjectCoord;
      // Set the initial position of the object
      thePlayer->gsObjectCoord.coord.t[0]=nX;   // X
      thePlayer->gsObjectCoord.coord.t[1]=nY;   // Y
      thePlayer->gsObjectCoord.coord.t[2]=nZ;   // Z
      // setting the players gsObjectCoord.flg to 0 indicates it is to be
      // drawn
      thePlayer->gsObjectCoord.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. Mostly 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 commemtary 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 and embarrassing internal memory mapping, the details of which we will 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 - this is really odd! 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. Suddenly 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 normaliased values). We store our two lights in an array defined as follows:

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

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

```
void InitialiseAllLights()
{
       InitialiseLight(&flLights[0], 0, -1, -1, -1, 255,255,255);
       InitialiseLight(&flLights[1], 1, 1, 1, 1, 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 `InitialiseLight` as shown below, and passes the light's position and r,g,b intensity value.

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

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        view;
```

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 car
       view->super = WORLD; //&theCar.gsObjectCoord ;
       // 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 coordinate 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 `AddModelToPlayer)`. 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 acheived 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 coordinates and one for lighting calculations. When thats 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(PlayerStructType0 *thePlayer, GsOT *othWorld)
{
       MATRIX  tmpls, tmplw;
       //Get the local world and screen coordinates, needed for light
       // calculations
       GsGetLws(thePlayer->gsObjectHandler.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->gsObjectHandler,4,
                        (u_long*)getScratchAddr(0));
}
```

The final step is to actually call the `DrawPlayer` function from the appropriate place inside the `RenderWorld` function which is as follows

```
...
            GsClearOt(0, 0, &othWorld[currentBuffer]);
            // draw the player
            DrawPlayer(&theCar, &othWorld[currentBuffer]);
```

```
            //print your elegant message
            FntPrint("Hello World!\n");
...
```

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
      InitialiseGraphics();
      // Setup view to view the car from the side
      InitialiseView(&view, 250, 0, 1000, -500, 0, 0, 0, 0 );
      InitialiseAllLights();
      // The car's initial xyz is set to 0,-200,0
      AddModelToPlayer(&theCar,  0, -200,0, (long*)CAR_MEM_ADDR);
      while(1){
            PADstatus=PadRead();
            if (PADstatus & PADselect) break;
            // render the car and hello world message
            RenderWorld(&theCar);
            }
      // 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.

STEP 4. Transforming a 3D model.

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 gsObjectCoord.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 *gsObjectCoord, 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.gsObjectCoord,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:

```
gsObjectCoord->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 `gsObjectCoord`.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 center 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 center. 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 `gsObjectCoord->coord->m` 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.gsObjectCoord,  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 *gsObjectCoord, 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(&gsObjectCoord->coord, &matTmp, &gsObjectCoord->coord);
      // set the flag to redraw the object
      gsObjectCoord->flg = 0;
}
```

Ok now you're happening, lets give joypad reading its own function.

STEP 5. Odds and sods.

A joypad function.

Create a function to process the joypad input as follows:

```
int ProcessUserInput()
{
      PADstatus=PadRead();
      if(PADstatus & PADselect)PLAYING=0;
      if(PADstatus & PADLleft){
              FntPrint( "Left Arrow: Rotate Left\n");
              RotateModel (&theCar.gsObjectCoord,  0, -64, 0 );
      }
      if(PADstatus & PADLright){
              FntPrint ( "Right Arrow: Rotate Right\n");
              RotateModel (&theCar.gsObjectCoord,  0, 64, 0 );
      }
      if(PADstatus & PADstart){
              FntPrint ( "PAD start\n" );
      }
      if(PADstatus & PADcross){
              FntPrint ( "PAD cross: Move Forward\n");
              MoveModel(&theCar.gsObjectCoord,0,0,32);
      }
      if (PADstatus & PADsquare){
              FntPrint ( "PAD square Move Backward\n");
                   MoveModel(&theCar.gsObjectCoord,0,0,-32);

       }
}
```

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

```
int main()
{
      // set up print-to-screen font
      FntLoad(960, 256);
      FntOpen(-96, -96, 192, 192, 0, 512);
      // set up the controller pad
      PadInit();
      InitialiseGraphics();
      InitialiseStaticView(&view, 250, 0, 0, -500,-1000, 0, 0, 0 );
      InitialiseAllLights();
      AddModelToPlayer(&theCar,  0, -200,0, (long*)CAR_MEM_ADDR);
      while(PLAYING){
            ProcessUserInput();
            currentBuffer=GsGetActiveBuff();
            RenderWorld(&theCar);
            FntPrint("Hello World!\n");
            FntFlush(-1);
            }
      // clean up
      ResetGraph(0);
      return 0;
```

```
}
```

Note that you will have to define the variable PLAYING and give it a default value of 0.

Hierarchical coordinate systems.

A nice feature of the PSX is the implementation of hierarchical coordinate system. A coordinate system can use any other object's coordinate 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.

```
view->super = &theCar.gsObjectCoord ;
```

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 & PADL1){
      FntPrint ( "PAD padL1:STATIC_VIEW\n");
      InitialiseStaticView(&view, 250, 0, 0, -500,-1000, 0, 0, 0 );
}
if (PADstatus & PADR1)
{
      FntPrint ( "PAD padR1: TRACKER_VIEW\n");
      InitialiseTrackerView(&view, 250, 0, 0, -1000, -1000, 0,0,0);
 }
```

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 vsyncInterval=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 VSync variable in the RenderWorld function:

```
...
      // wait for end of drawing
      DrawSync(0);
      // wait for V_BLANK interrupt
      vsyncInterval=VSync(0);
      // print the vSync interval
      FntPrint("VSync Interval: %d.\n",vsyncInterval);
      // force text output
      FntFlush(-1);
      // Swap The Buffers
      GsSwapDispBuff();
...
```

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.

So what next?

Try to get the car to move in the direction that it is pointing
Try and get some scenery into the scene to drive around.

STEP 6. Precision problems.

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 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 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 scalings up and down can be produced by different numbers. The solution to this problem is not to let errors

accumulate in the players 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         gsObjectHandler;
       GsCOORDINATE2   gsObjectCoord;
}PlayerStructType1;
```

Having defined the rotation vector we should initialise it. We decide that player initialisation will probably need it's own function so we define:

```
void InitialisePlayer(PlayerStructType1 *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
      AddModelToPlayer(thePlayer,  0, 0,0, CAR_MEM_ADDR); //-200
}
```

which we call from the main function instead of `AddModelToPlayer`:

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

A new 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 write our own ResetMatrix function which will reset the players rotation matrix to the identity matrix without interfering with its translation vector. We define it as follows:

```
void ResetMatrix(short m[3][3])
{
m[0][0]=m[1][1]=m[2][2]=ONE;
m[0][1]=m[0][2]=m[1][0]=m[1][2]=m[2][0]=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 `RotateModel` function:

```
void RotateModel (GsCOORDINATE2 *gsObjectCoord,SVECTOR *rotateVector , int
nRX, int nRY, int nRZ )
{
      MATRIX matTmp;
      // the reset the players coord system to the identity matrix
      ResetMatrix(gsObjectCoord->coord.m );
      // 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(&gsObjectCoord->coord, &matTmp, &gsObjectCoord->coord);
      // set the flag to redraw the object
      gsObjectCoord->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.

STEP 7. Translating in the correct direction.

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 rotation vector. 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. Thus we have:

```
// move the model nD steps in the direction it is pointing
void AdvanceModel (GsCOORDINATE2 *gsObjectCoord, SVECTOR *rotateVector,
```

```
                        int nD)
{
      // Moves the model nD units in the direction of its rotation vector
      MATRIX matTmp;
      SVECTOR startVector;
      SVECTOR currentDirection;
      // if nD = 0 there is no movement and we need to avoid the
      // main body of the function which will cause a divide by zero error
      if(nD!=0){
            // set up original vector, pointing down the positive z axis
            startVector.vx = 0; startVector.vy = 0;   startVector.vz = ONE;
            // RotMatrix sets up the matrix coefficients for rotation
            RotMatrix(rotateVector, &matTmp);
            // multiply startVector by mattmp and put the result in
            // currentDirection which is the vector defining the direction
            // the player is pointing
            ApplyMatrixSV(&matTmp, &startVector, &currentDirection);
            // set the translation based on the currentDirection, note
            // currentDirection components have been scaled by 4096 so we
            // divide by 4096 to scale them back
            gsObjectCoord->coord.t[0] +=(currentDirection.vx * nD)/4096;
            gsObjectCoord->coord.t[1] +=(currentDirection.vy * nD)/4096;
            gsObjectCoord->coord.t[2] +=(currentDirection.vz * nD)/4096;
            // Because it has changed set flg to redraw object
            gsObjectCoord->flg = 0;
      } //end if
}
```

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

```
      if(PADstatus & PADcross){
              FntPrint ( "PAD cross: Move Forward\n");
     AdvanceModel(&theCar.gsObjectCoord,&theCar.rotation,&theCar.speed,16);
      }
      if (PADstatus & PADsquare){
              FntPrint ( "PAD square Move Backward\n");
    AdvanceModel(&theCar.gsObjectCoord,&theCar.rotation,&theCar.speed,-16);
       }
```

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.gsObjectCoord.coord=GsIDMATRIX;
              theCar.speed=0;
              theCar.rotation.vx=0;  theCar.rotation.vy=0;
              theCar.rotation.vy=0;
              theCar.gsObjectCoord.flg=0;
      }
```

Note that a second consequence 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.

STEP 8. Texture mapping.

Introduction to texture mapping.

The default form of texture mapping implemented by the PSX is image mapping, mapping of 2D images to polygons or whole objects. The PSX supports 3 kinds of images for texture mapping, images may be 16, 8, or 4 bits per pixel. 16 bit mode specifies each pixel by a RGB triple whereas 8 bit and 4 bit images use palettes or Colour LookUp Tables (CLUTs) that also have to be loaded into memory.

The PSX uses a proprietary format for image textures known as the TIM format. The good news is that utilities exist for converting from .bmp files to .tim files. The not so good news is that you have to know where the textures and their palettes are to be stored in video memory. Note that textures are ultimately stored in video memory, not conventional memory.

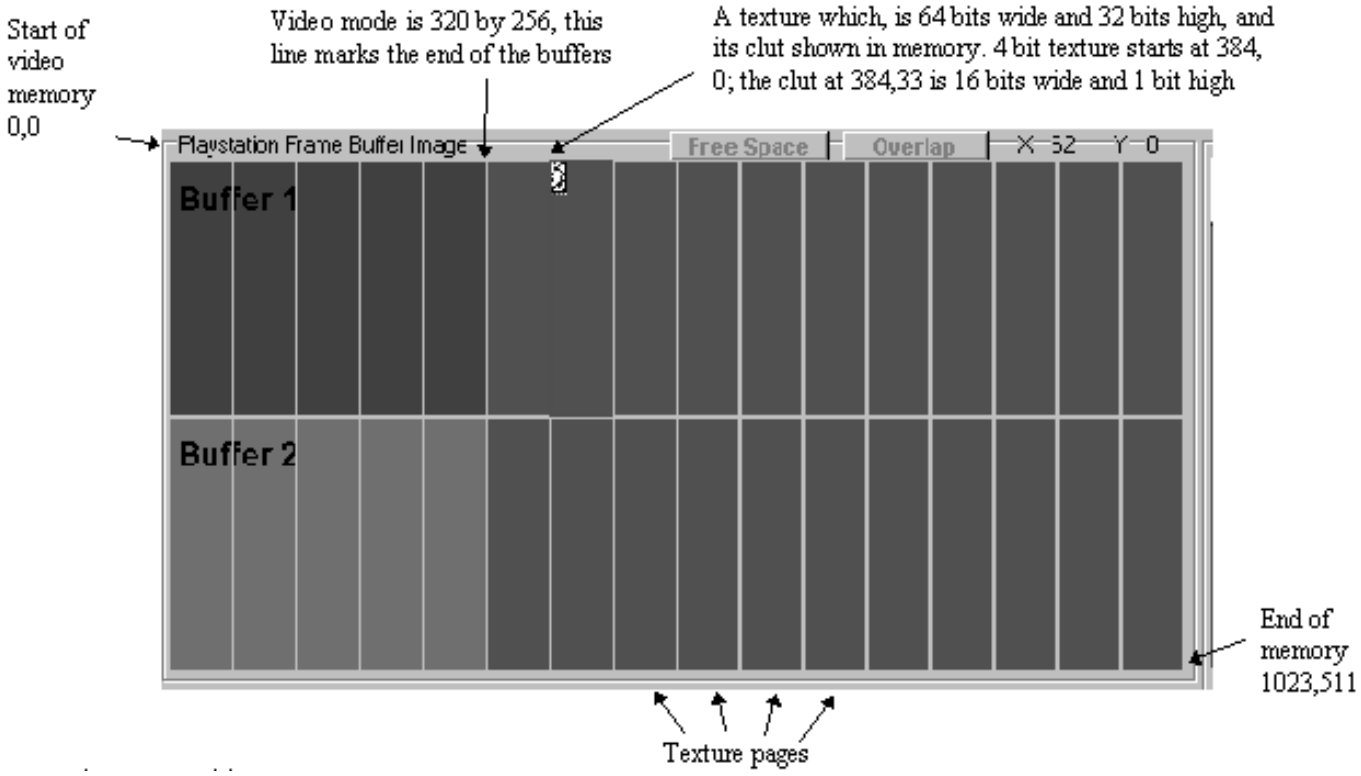


Figure 1. Video Memory.

As shown above video memory is 1 megabyte big. It starts at 0,0 and extends to 1023, 511. Two buffers are allocated from this memory when you initialise the graphics system to the size of the screen width and height, marked as buffer 1 and buffer 2 above. The rest of the memory is free to use for textures and their cluts as necessary. The memory is subdivided into texture pages which are 64 pixels wide by 256 pixels high. The 4 bits per pixel com.tim texture which is 32 bits wide and 16 bits high is shown starting at video memory address 384, 0; its clut is 16 bits wide and 1 bit high and is loaded from video memory address 384,33.

Our car's number plates have appeared green so far, in actual fact the car model specifies the number plates as being textured by a file called com.tim. In order to get the texture to map onto the number plates we have to load the texture into conventional memory via our batch file, and then load it from conventional memory into the frame buffer within our program. The stunning graphic is shown below (though in greyscale).

Figure 2 The com.tim texture file.

Loading a texture into memory.

Firstly you need to assign a conventional memory address to load the texture into, and specify it in your auto batch file:

```
local dload data\car\car01.tmd  80090000
local dload data\car\com.TIM 800A0000
local load main
go
```

Then you have to define the same hex address in your program.

```
#define CAR_TEX_MEM_ADDR    (0x800A0000)
```

Note that we use the convention of giving the constant a sensible name followed by `tex_mem_addr` to indicate it is the memory address of a texture image. To move the texture from conventional memory into the video buffer we need to call the following function.

```
// load up from conventional memory into video memory
int LoadTexture(long addr)
{
      RECT rect;
      GsIMAGE tim1;

      // get tim info/header, again a little bit of magic is needed the
      // pointer
      // is incremented past the first 4 positions to get to this!
      GsGetTimInfo((u_long *)(addr+4),&tim1);
```

```
      // set the rect struct to contain the images x and y offset, width
      // and height
      rect.x=tim1.px;
      rect.y=tim1.py;
      rect.w=tim1.pw;
      rect.h=tim1.ph;
      //load image from main memory to video memory
      LoadImage(&rect,tim1.pixel);
      // if image has clut we need to load it too,
      //pmode =8 for 4 bit and 9 for 8 bit colour
      if((tim1.pmode>>3)&0x01)
    {
      // set the rect struct to contain the clut's x and y offset, width
      // and height
            rect.x=tim1.cx;
            rect.y=tim1.cy;
            rect.w=tim1.cw;
            rect.h=tim1.ch;
      // load the clut into video memory
            LoadImage(&rect,tim1.clut);
      }
      // wait for load to finish by calling drawsync.
      DrawSync(0);
      return(0);
}
```

The `LoadTexture` function is pretty straightforward (apart from the curious pointer incrementing at the start ...) and can really be treated like a black box. You want to load a texture into video memory, then you use this function and it will do it regardless of whether it is a 4 bit or 8 bit texture with a clut, or a 16 bit clutless texture. We call it in the obvious way during initialisation:

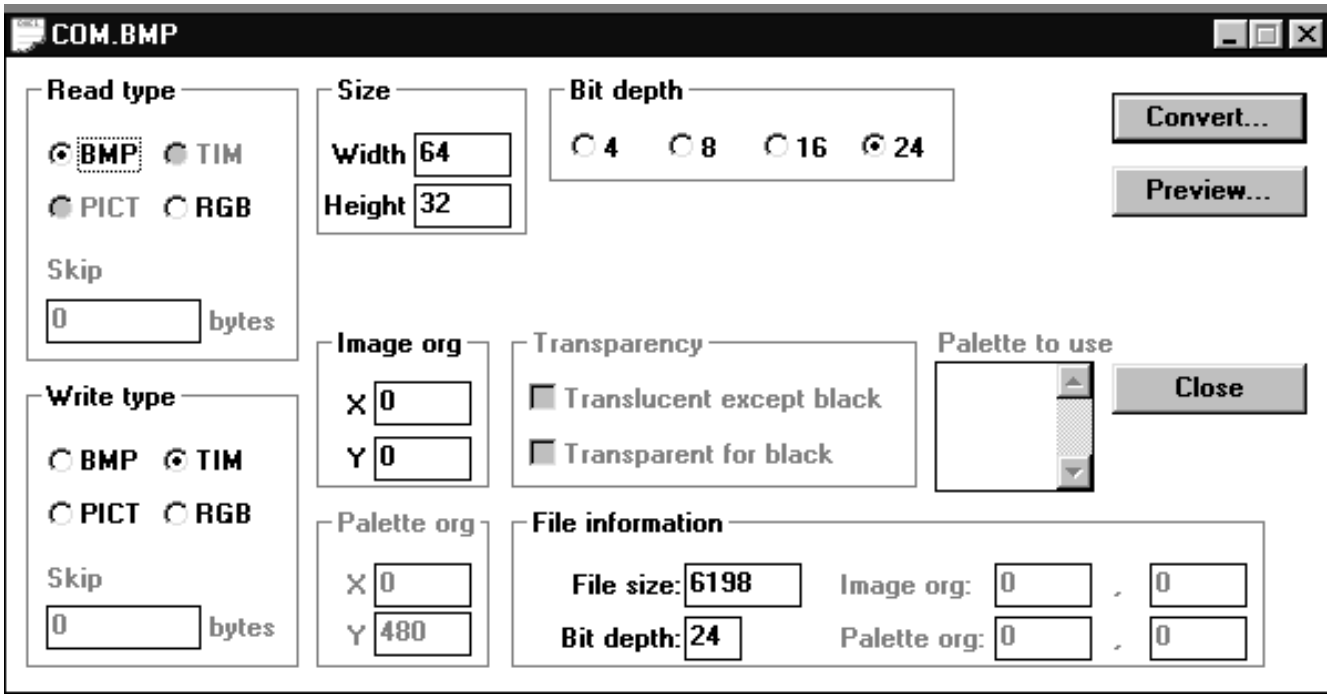
```
LoadTexture(CAR_TEX_MEM_ADDR);
```

With the above amendments your program should now show the car with both the front and back number plates textured by the com.tim image.

Making your own texture files.

Firstly note that it is handy to use tim files whose dimensions are powers of 2 and that they should be kept under 256 x 256 or special techniques are needed to deal with them. Length or widths which are odd numbers can lead to problems and are to be avoided. Secondly note that we want to keep texture files as small as possible so that they can be drawn quickly and can be cached during rendering. Obviously then we have a conflict between wanting to have lots of large highly detailed possibly odd shaped textures in our worlds and needing to restrict memory and processor usage. Bearing this in mind you can now set about making your own image to replace the com.tim image. The com.tim image has a 16 colour palette and is 64 pixels wide by 32 pixels wide. Make or convert an image of your own and save it as com.bmp, making sure you have reduced the palette to 16 colours.

There are two windows based tools for converting from bmps to tim files. The standard one which comes with the Yaroze is timutil.exe and can be found in the PSX\BIN directory. Run this utility and load up your com.bmp file, the following window appears:



The bit depth radio button should already be set to 4 if you saved your image as a 16 colour image, if not setting the bit depth to 4 will force it for you but timutil uses a naff algorithm for colour reduction so it's better to use a more sophisticated tool. Set the image org (origin) to x value 384 and y value 0, set the palette org (origin) to x value 384 y value 32. You have now set the position of your image and its clut to be the same as specified for the original com.tim. Press the convert button and choose com.tim a the filename. Now when you run the program your own image will texture the number plates.

Well you managed to get your own texture on the screen, and there's still much to learn about texturing on the PSX but before we do we're going to learn about tmd model files and how you create them. Note that for the moment you can't use 8 or 16 bit textures or place a texture on a different part of the car.

STEP 9. 3D Modelling

Making your own model.

We'll assume that in order to make your own model you will start of with a dxf file that has been exported from some modelling program. The aim of the exersice is to now create a flat landscape of textured polygons to drive around on, so we start with a file called square1.dxf which is a one sided quadrilateral and to be found in the directory data\square1.

Converting from dxf to rsd.

As a preliminary step you may want to scale or translate your dxf model in which case you should use the **rsdform** utility to be found in the PSX\bin subdirectory. For this example we do not need to scale or translate, and it is often easier to do this in a modelling package anyway.

The PSX converts dxf files to tmd files via an imtermediate format called rsd. The rsd format is attractive because at this stage all information is given in readable ascii text files, so you can get in there and edit them directly. In actual fact when you convert to rsd format you get a group of four files that describe the transformed object, and these must be kept in the same directory for successful final conversion to tmd format. Definitive documentation on the format of these files can be found on the web page `rsd.htm` to be found in the PSX\docs subdirectory, this is a supplementary file which was downloaded from the Yaroze web site. The four files that are created after conversion from dxf are:

PLY file: Describes information about vertices, polygons and polygon normals, and we will not consider it further here.

GRP file: Describes grouping information. A group of polygons in the PLY file can be assigned a name and a group of polygons can be operated by the material editor, and certain polygons can be accessed from the program. This is obviously a very useful facility but we will ignore it for the moment.

RSD file: Describes relationships between PLY/MAT/GRP and texture (.tim) files. From our point of view what is important is that the rsd file contains information about the textures used by the model. An example file is shown below:

```
@RSD940102
PLY=sample.ply
MAT=sample.mat
GRP=sample.grp
NTEX=3
TEX[0]=texture.tim
TEX[1]=texture2.tim
TEX[2]=texture3.tim
```

The first line is just an ID which we can ignore, and the next three lines specify the names of the ply mat and grp files associated with this rsd. The next line says the number of textures used is 3, and the next three lines specify the names of the textures.

MAT file: Describes material information on a polygon ie: all material properties of a polygon, is it to be lit or not, will it be flat or Gouraud shaded or coloured, is it semitransparent etc. Check out the `rsd.htm` page mentioned above for full details but we will return to this in a moment when we texture our quadrilateral.

So the first step is to convert from dxf to rsd. Two tools **dxf2rsd** and **dxf2rsdw** 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.

Polygon reduction is a vital process to us because the less polygons you have the faster your program will run, so it's important to try and use the minimum number of polygons possible to represent your models. Note than most modelling packages tend to give you very polygon rich models. For this example we have a quadrilateral which is made up of two triangles so it can only be reduced to a single quadrilateral.

To convert the square1.dxf file we we use the DOS dxf2rsd utility. As we are likely to use this facility a number of times it is handy to create a batch file that stores the operation. Create a .bat file called conv1.bat and type in the following:

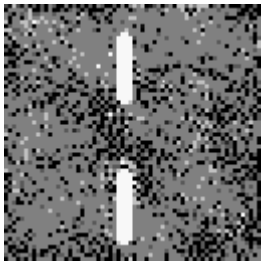
```
dxf2rsd -o square1.rsd -quad2 20 -back  square1.dxf
```

Following some experimentation I found that the above converts the file in the way I 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 - this turns our quadrilateral defined by two triangles into a single quadrilateral, which is going to use less memory and make it easier to texture.

Note that there are 21 different flags that can be passed to dxf2rsd, and we will not go through them all here. Your best approach in dealing with converting other models 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: 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).

Assigning texture to the quadrilateral.

Our quadrilateral is roughly as wide as a road in relation to the car and we have an image to texture it with which is 64 x 64 pixels wide, square1.tim as shown below:



There is a tool which helps you colour and texture your models, called rsdtool which is written in Direct3D. It is fairly useful but does not come with the PSX release software and I'm not sure it will be installed in time for the course so to assign texture to the quadrilateral we have to go into the .rsd and .mat files and edit them by hand. The dxf2rsd function made the four rsd related files for us and the square1.rsd file now looks like this:

```
@RSD940102
```

```
PLY=square1.ply
MAT=square1.mat
GRP=square1.grp
# Number of Texture Files
NTEX=0
```

As this file specifies that no textures are used we need to amend it to use our texture of choice:

```
@RSD940102
PLY=square1.ply
MAT=square1.mat
GRP=square1.grp
# Number of Texture Files
NTEX=1
TEX[0]=square1.tim
```

- we change the number of textures to 1, and specify square1.tim as the texture to use.
The .mat file currently looks like this:

```
@MAT940801
# Number of Items
1
# Materials
0-0   0 F C 200 200 200
```

The full .mat file specification is given in the rsd.htm web page, but the little that you need to know is given here. The .mat file format is as follows, the ID number on line 1 is followed by a comment on the line 2 that indicates the third line contains the total number of material descriptors. A material descriptor takes one line and in this case we only have one. The material descriptor format is as shown below:

Polygon No	Flag	Shading	Material information
0-0	0	F	C 200 200 200

The Polygon No field can specify a single polygon or a range or list of polygons. In this case it specifies the range 0 to 0 - ie: applies to the first (and only) polygon, we could have just specified a single 0 here.

The flag field is a single hex number where the various bits of the flag indicate things like whether the polygons are to be one or two sided, whether they will be semi transparent, whether they will be shaded etc. A value of 0 indicates shading is on, polygons are one sided, and there is no transparency.

The shading field takes a value of either F or G to specify Flat or Gouraud shading respectively.

The material field specifies the type of material to be applied, in this case a colour, which is specified by the ‘C’ character, and the rgb values are specified by the last three numbers. We need to change the .mat file to indicate that our model is going to be textured, so edit your .mat file to the following:

```
@MAT940801
# Number of Items
1
# Materials
0-0 0 F T 0 0 0 63 0 0 63 63 63
```

As highlighted by the table below the first three fields of the material descriptor are the same, what has changed is the material information.

Polygon No	Flag	Shading	Material information
0-0	0	F	T 0 0 63 0 0 63 63 63 0

A table describing the format of a textured polygon is shown below.

TYPE	TNO	U0	V0	U1	V1	U2	V2	U3	V3
T	0	0	63	0	0	63	63	63	0

In the first column is specified the material type which is T instead of a C to indicate a texture is to be used instead of colour. The TNO column specifies the TIM data file to be used, and this references the texture number as described in the RSD file ie: it is 0 because texture 0 as specified in the texture file (square1.tim) is going to be used.

The U and V parameters specify the positions of the vertices in the texture file As we have four vertices four positions are specified. Recall that our texture is 64 by 64 pixels wide. Thus the position of the first vertex is named by U0,V0 is specified as 0,63 whilst the last vertex is named U3,V3 is specified as 63,0. Thus the vertices are aligned to the four corners of the texture and the full texture will be mapped to the quadrilateral, and with the uv mapping described above the texture will come out in the orientation we expect.

Creating a tmd file from an rsd file

We are now ready to convert the .rsd file to a .tmd file and the utility to do this, found in PSX\bin is **rsdlink** (which may have been better named rsd2tmd...) It, for a change is a very simple tool to use, but I recommend that you also put the following line into a batch file called say conv2.bat to save on key presses as you experiment:

```
rsdlink -v -o square1.tmd square1.rsd
```

The arguments given to this function are very simple; -v specifies the function should give us output on how it got on; -o square1.tmd specifies square1.tmd as the name of the output file; and square1.rsd is taken as the imput file. It gives us the following output:

```
(C) 1994-1996 Sony Computer Entertainment Inc. All Rights Reserved.
rsdlink Version 3.72 Tue Oct  1 00:00:00 1996
[0] =============== RSD ==============
RSD files : D:\AFIRST\12\DATA\SQUARE1\square1.ply, square1.mat, square1.grp

   TEX[0] = square1.tim
   POLYGON     :     1
```

```
   VERTEX      :     4
   NORMAL      :     1
   MATERIAL    :     1
   Range       : (-602, 0, -602)-(602, 0, 602)
================ TMD =================
   Output TMD  : square1.tmd
   TMD header  :       (    12 bytes)
   Objects     :     1 (    28 bytes)
   Primitives  :     1 (    32 bytes)
                     1  ... Flat Textured Quadrangles
   Vertices    :     4 (    32 bytes)
   Normals     :     1 (     8 bytes)
---------------------------------------
Total file size:           112 bytes
```

The output tells us that useful things like the model only contains 1 flat textured quadrangle and the total size is only 112 bytes, pretty small huh?

We are now ready to try and get our new model on board. The simplest way to test this is to take one of our previous programs say step 7 or 8 and edit the auto file to put the square1 tmd and tim files in place of car01.tmd and com.tim respectively. Thus you should have:

```
local dload data\square1\square1.tmd  80090000
local dload data\square1\square1.TIM  800A0000
local load main
go
```

Now run your program and you should be able to drive your quadrilateral textured by the square1 road image around the screen instead of the car. However you will note that the texture seems to warp as you move the car around, this is a consequence of texture mapping with a small texture using few polygons - never fear it will behave appropriately (well almost) when we use it as a road in the next step.

STEP 10. Building a world.

Building a road 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 world that consists of the road. To do this we are going to have to go through all the same steps for the world as we did for the player, build a struct to define it, initialise it, build a function to add models to the world, write a function to draw the world, and we will need and extra function to take a map of the world and build a road according to it.

The world 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. Consequently we define arrays of relevant data structures

eg: an array of `GsDOBJ2`, an array of `GsCOORDINATE2`, and an array pointers to tmd models in memory `(unsigned long *)`. We also include an extra variable `nTotalModels` which will track the number of objects the world contains. The world struct is defined:

```
typedef struct
{
	// A variable to track the total number of models in the world
	int nTotalModels;
	// We need one of each of the following two data structures for each
	// object in the world thus we have arrays of them
	GsDOBJ2		gsObjectHandler[MAX_WORLD_OBJECTS];
	GsCOORDINATE2	gsObjectCoord[MAX_WORLD_OBJECTS];
	// We also need an array of pointers to tmds in memory
	unsigned long *lObjectPointer[MAX_WORLD_OBJECTS];
} WorldStructType0;
```

We therefore need to create an instance of the world.

```
// create an instance of the world
WorldStructType0	theWorld;
```

Creating a world map.

The easiest kind of world to create is a cell based one, a world made up of a grid of quadrilaterals. We can define an array of characters to represent this world, this is both efficent use of memory and easily editable so we can change the map. Then we can interpret the individual characters in the array as representing different objects. Our world will lie in the XZ plane so for convenience we define global constants for these, and define a global constant for the maximum number of objects:

```
#define MAX_X (15)
#define MAX_Z (15)
#define MAX_WORLD_OBJECTS   (225)
```

The world data can then be defined globally as:

```
char worldGroundData[MAX_Z][MAX_X] ={
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','0','0','0','0','0','0','0','0','0','0','0','0','0','1'},
{'1','1','1','1','1','1','1','1','1','1','1','1','1','1','1'},
};
```

We plan that where ever there is the character 1 in the array we will place a square1.tmd textured by the square1.tim, so our road will be a square one, right around the perimeter of the grid. If the character is 0 we plan to ignore it. Before we progress to the next stage we need to define addresses in memory to hold our square1.tmd and square.tim:

```
#define SQUARE1_MEM_ADDR       (0x80093000)
#define SQUARE1_TEX_MEM_ADDR   (0x800B0000)
```

NB - Don't forget to amend your auto batch files to load up these to the appropriate addresses!

The AddModelToWorld function.

As was the case for the player we write a function to add models to the world structure. You will note that it is almost exactly the same apart from the basic data structures being in arrays:

```
voidAddModelToWorld(WorldStructType0 *theWorld, int nX, int nY, int nZ,
                        unsigned long *lModelAddress)
{
   theWorld->lObjectPointer[theWorld->nTotalModels] =
      (unsigned long*)lModelAddress;
   //increment the pointer to move past the model id. (weird huh?)
   theWorld->lObjectPointer[theWorld->nTotalModels]++;
   // map tmd data to its actual address
   GsMapModelingData(theWorld->lObjectPointer[theWorld->nTotalModels]);
   //initialise the objects coordinate system - set to be that of the WORLD
   GsInitCoordinate2(WORLD, &theWorld->gsObjectCoord
            [theWorld->nTotalModels]);
   //increment pointer twice more-to point to top of model data (beats me!)
   theWorld->lObjectPointer[theWorld->nTotalModels]++;
   theWorld->lObjectPointer[theWorld->nTotalModels]++;
   // link the model (tmd) with the players object handler
   GsLinkObject4((unsigned long *)theWorld->lObjectPointer
            [theWorld->nTotalModels], &theWorld->gsObjectHandler
            [theWorld->nTotalModels], 0);
   // set the amount of polygon subdivision that will be done at runtime
   theWorld->gsObjectHandler[theWorld->nTotalModels].attribute = GsDIV1;
    // Assign the coordinates of the object model to the Object Handler
    theWorld->gsObjectHandler[theWorld->nTotalModels].coord2 =
                        &theWorld->gsObjectCoord[theWorld->nTotalModels];
    // Set The Position of the Object
    theWorld->gsObjectCoord[theWorld->nTotalModels].coord.t[0]=nX;   // X
    theWorld->gsObjectCoord[theWorld->nTotalModels].coord.t[1]=nY;   // Y
    theWorld->gsObjectCoord[theWorld->nTotalModels].coord.t[2]=nZ;   // Z
    // Increment the object counter
    theWorld->nTotalModels++;
    // flag the object as needing to be drawn
    theWorld->gsObjectCoord[theWorld->nTotalModels].flg = 0;
}
```

We then need a function to initialise the world which will go through the world ground data array and add instances of our piece of road where 1's are found. Before we can do this we define:

```
#define SEPERATION (1200)
```

which is the width (and length) of the square1 quadrilateral. We can then initialise:

```
void InitialiseWorld ()
{
int tmpx,tmpz;
      char c;

      //initialise total number of models to zero
      theWorld.nTotalModels=0;
      // load up the square1 texture to video memeory
      LoadTexture(SQUARE1_TEX_MEM_ADDR);
      // then for each element of the worldGroundData array if we find a 1
      // then place an instance of square1.tmd at the appropriate position
      // in the world.
      for (tmpz = 0; tmpz < MAX_Z; tmpz++ ){
            for (tmpx = 0; tmpx < MAX_X; tmpx++ ){
                  c= worldGroundData[tmpz][tmpx];
                  if(c == '1')  {
                        AddModelToWorld(&theWorld, (tmpz * SEPERATION),(0),
                                                (tmpx * SEPERATION),
                                                (long *)SQUARE1_MEM_ADDR);
                  } //end if
            )//end for tmpx
     } //end for tmpz
}
```

We now need a DrawWorld function which will go through and draw all the objects in the world, which again is like the DrawPlayer function except that it has to go through the arrays of objects to draw them. Thus:

```
void DrawWorld(WorldStructType0 *theWorld, GsOT *othWorld)
{
   MATRIX  tmpls, tmplw;
   int nCurrentModel;

   for (nCurrentModel = 0; nCurrentModel < theWorld->nTotalModels;
            nCurrentModel++)
   {
      //Get the local world and screen coordinates, needed for light
      //calculations
      GsGetLws(theWorld->gsObjectHandler[nCurrentModel].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( &theWorld->gsObjectHandler[nCurrentModel], othWorld,
            14 - ORDERING_TABLE_LENGTH,(u_long *)getScratchAddr(0));
   }
}
```

We then can to call this function from RenderWorld:

```
....
      GsClearOt(0, 0, &othWorld[currentBuffer]);
      // Draw the world
      DrawWorld(theWorld,  &othWorld[currentBuffer]);
      // Draw the player
      DrawPlayer(thePlayer, &othWorld[currentBuffer]);
....
```

and have to remember to call InitiaiseWorld from our main function:

```
....
      InitialisePlayer(&theCar,  0, -200,0, (long*)CAR_MEM_ADDR);
      LoadTexture( CAR_TEX_MEM_ADDR);
      InitialiseWorld();
      while(PLAYING){
....
```

Ok you can now drive down the road...

Allow a couple of views.

We set the initial view and the car's position to the following parameters in the main function:

```
....
      InitialiseStaticView(&view, 250, 0, 0, -500,-1000, 0, 0, 0 );
      InitialiseAllLights();
      InitialisePlayer(&theCar,  0, -200,0, (long*)CAR_MEM_ADDR);
....
```

and we set the parameters in the view flipping code in the joypad function to:

```
      if (PADstatus & PADL1){
            FntPrint ( "PAD padL1:STATIC_VIEW\n");
            InitialiseStaticView(&view, 250, 0, 0, -500,-1000, 0, 0, 0 );
      }
      if (PADstatus & PADR1)
      {
            FntPrint ( "PAD padR1: TRACKER_VIEW!\n");
            InitialiseTrackerView(&view, 250, 0, 0, -300, -1500, 0,200,0 );
      }
```

Project: Build the rest of the track and give the car some dynamics.

So far you only have a texture which is correctly aligned for the direction you are pointing in when you start, and the corners are unreasonably sharp. If you had a set of textures as shown below...

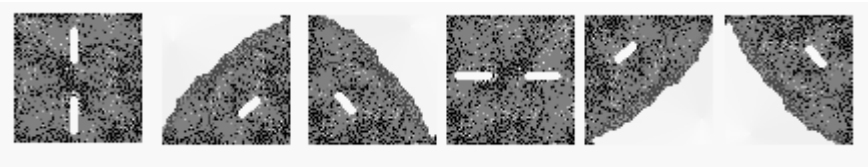


Figure 3. A set of road textures.

...you could have smoother corners and make sure that the white line is always in the center of the road as it should be. This set of six textures can be used to make turns in all directions eg:

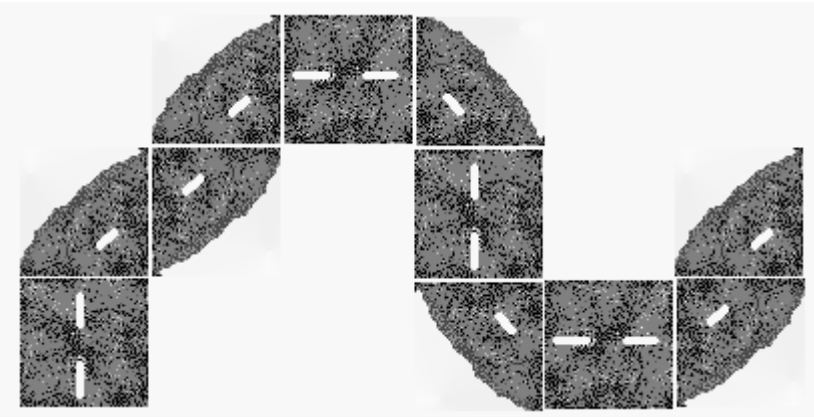


Figure 4. You can use the set to turn in either direction.

If you had these textures in your program you could make more interesting and sensible tracks for your car to follow. Although it is still a bit limited you can drive along the diagonals and have fast and slow corners.

Tips.

The game concept.

Note that the first thing you should do is think about a game/simulation concept. I have taken you down the path of writing some sort of driving game because it is a useful example for teaching purposes, but with a few subtle variations you can make it into something quite different. Plan your game now and storyboard important sequences in the game - this is in fact the most important step in the whole process!

Modelling the world

You will obviously need to make all the textures but note that once you have made textures 1 and 2 all the rest are rotations of these two textures. Thus an obvious solution to building your track is to make six squares which are textured with the textures 1-6 above using a paint/draw tool to perform rotations on patterns 1 and 2 above.

The texture associated with a tmd is hard wired before you get your model into the program so this means that you should have a separate copy of the square1.tmd for each of your track building blocks which you should renumber for readability.

Physically based modelling

To model the car dynamics you need to consider physically based modelling: ie modelling of the underlying physics which will include acceleration/deceleration, friction, skidding etc. If you put ramps into your scene you have to model projectile motion. Any A level/college level physics book has the equations for these in it. You job is to implement or fake them in the most computationally efficient manner you can.

Collision detection.

At some point you will want to model collisions. On a flat world this is really 2D, in general you only have to detect whether two bounding boxes overlap. It becomes useful to store and update a velocity vector for your player so that its easy to manipulate changing direction on collisions.

Behavioural modelling

Sooner or later you will want to put some opponents into your world which means you want to model their behaviours.

Speed considerations.

You have already noticed that the frame rate goes down when you translate, and more when you rotate at the same time. As your processing cycle gets longer and the frame rate goes down this means that your model's speed is affected by the transformation you are applying. A way to make the processing step take a standard amount of time is to apply all possible transformations every cycle. Thus even if you are only rotating you apply translation with translation factors of 0 etc so the transformation step will take the same amount of time per cycle regardless of the operations performed.

The car model we are using is quite complex, a model with much fewer polygons could be substituted.

If you fill the world map with tiles then the system performance will be strongly degraded by the number of tiles visible leading to a slowing down or speeding up of the game depending on the current view. One solution to this is to only show the portion of the world that surrounds the player, by setting a visibility flag for each tile on the basis of the cars current position and possibily direction. These calculation will have to be quick to justify their cost.

Remember, you can always find a way to speed things up.