PlayStation®

# Introduction to Programming the PSP® Release 1.0

**March 2005**

# Table of Contents

# About This Manual

This is the *Introduction to Programming the PSP® Release 1.0.*

This document is aimed at programmers who are new to the PSP environment. It describes sample code which is included within the same source directory as this document.

The document is aimed at programmers with knowledge of programming languages such as C and C++, as well as a basic understanding of three-dimensional computer graphics. Knowledge of Assembly language concepts is useful for understanding the information in Chapter 5 about the VFPU (Vector Floating Point Unit) coprocessor. Appendix A contains information to support this chapter and contains an introduction to inline Assembler syntax for those who are new to Assembly language.

This document refers to functions or instructions in the Run-time library documentation, available on https://psp.scedev.net. We recommend that you read the Run-time documentation before reading this document or, at least, have it available when you are using this guide.

Please note that this document is not a substitute for the Run-time library documentation, but you may find that some further explanations are given in this guide of concepts that are covered in the Run-time documentation and, possibly, that some material is repeated.

Most of the examples and samples of code in this document contain functions that will work in both the Emulator and the DTP-T1000A hardware environments. However, there is information in Chapter 5 which is only relevant to the DTP-T1000A hardware environment: this is information about writing code that uses the VFPU.

Appendix B contains information about the different function calls in the source code, relating to the Emulator and the DTP-T1000A hardware.

## Changes Since Last Release

None: New manual.

## Prerequisites

Knowledge of PSP™ development.

## Typographic Conventions

Certain typographic conventions are used throughout this manual to clarify the meaning of the text:

| Convention | Meaning |
|---|---|
| `courier` | Indicates literal program code. |
| *italic* | Indicates names of arguments and structure members (in structure/function definitions only). |
| **medium bold** | Indicates data types and structure/function names (in structure/function definitions only). |
| blue | Indicates a hyperlink. |

## Developer Support

### Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

| Order Information | Developer Support |
|---|---|
| Attn: Developer Tools Coordinator<br>Sony Computer Entertainment America<br>919 East Hillsdale Blvd.<br>Foster City, CA 94404, U.S.A.<br>Tel: (650) 655-8000 | E-mail: scea_support@psp.scedev.net<br>Web: https://psp.scedev.net<br>Developer Support Hotline:(650) 655-5566<br>(Call Monday through Friday,<br>8 a.m. to 5 p.m., PST/PDT) |

### Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees only in the PAL television territories (including Europe and Australasia). You may obtain developer support or additional copies of this documentation by contacting the following addresses:

| Order Information | Developer Support |
|---|---|
| Attn: Development Tools Manager<br>Sony Computer Entertainment Europe<br>13 Great Marlborough Street<br>London W1F 7HP, U.K.<br>Tel: +44 (0) 20 7859-5000 | E-mail: scee_support@psp.scedev.net<br>Web: https://psp.scedev.net<br>Developer Support Hotline:<br>+44 (0) 20 7911-7711<br>(Call Monday through Friday,<br>9 a.m. to 6 p.m., GMT/BST) |

# Chapter 1: Preliminaries

This page intentionally left blank.

## Introduction

There are two development environments for the PSP:

- an emulation environment
- a hardware environment using the DTP-T1000A hardware tool

This chapter outlines the differences between the two development environments. It also defines the prerequisites for writing programs for the PSP.

This chapter can help you to get started if you have not already set-up your system, and it provides a list of materials that you need to get started in programming for the PSP.  However, this is not a set-up guide: there are already numerous documents relating to the DTP-T1000A hardware tool and the Emulator which describe the set-up processes for both environments.

## The Emulator

The PSP emulator environment is a set of tools that allows you to compile, run, view and debug your PSP programs on a PC, in the Windows environment. You can download all the files you need to compile and run Emulator code from the download section of https://psp.scedev.net.

The Emulator has proven to be a valuable method to help developers start developing code using the run-time libraries.

For installation information please refer to the documentation provided with these packages:

- Win32 Compiler (gcc)
- Win32 debugging environment
- Win32 PSP emulator program
- Run-time libraries & documents

The Emulator only supports the PSP CPU, graphics engine and hand-held controller.  If you start developing programs for the PSP using the Emulator, rather than the DTP-T1000A hardware tool, you will only have to make minimal changes to the code if you want to run the same code on the hardware tool at a later date.

### ALLEGREX CPU Emulation

This is the main functionality of the Emulator. All code is compiled to MIPS 4000 standard, as run on the ALLEGREX CPU in the PSP. This is a standard MIPS emulator, allowing you to compile and debug MIPS code in the Windows environment.

### Functionality of libGu and libGum Graphics Libraries

LibGu is a well featured graphics library supplied by Sony Computer Entertainment (SCE) that is supplied with the Emulator and the developers' hardware tool . Almost all of the features of this graphics library are supported in the Emulator using OpenGL, so you can write PSP-specific engine code using libGu while you are waiting for access to the PSP DTP-T1000A hardware tool.

You can also use the LibGum API in the Emulator environment. This is another graphics API that sits above libGu.  It provides, mainly, matrix math functionality.

### Controller Support

The Emulator supports controller emulation through standard Windows controller drivers. In this way, a Windows compatible controller can be set to behave in the emulation environment as the controls of the PSP. The Emulator supports all the buttons of the PSP as well as the analogue stick.

## The PSP DTP-T1000A Development Hardware (Hardware Tool)

The hardware tool is the equipment required for developing PSP software. It contains the full functionality of a PSP as well as PC hardware that allows the debugging environment to communicate with the Tool over the network.

The hardware tool is supplied with a PSP Controller that has a built-in screen. It also has a VGA-out connector which allows you to view output on a separate monitor.

For instructions on how to install the hardware tool, please refer to the documentation supplied with it.

To test code you have written, compile the code on a host PC running either Windows or Linux, and then send the program to the hardware tool, via the network, where your software is then executed.

### Differences Between the Emulator and the Hardware Tool

When you develop code using the hardware tool, you can use all the other hardware features that are part of the production PSP console, as well as the hardware features of the CPU and graphics engine mentioned previously. The features of the production PSP console are as follows:

- VFPU (Vector Floating Point Unit)
- Audio & Video
- UMD
- Networking
- Memory Stick
- USB

Information on all of the above can be found in the documentation available on https://psp.scedev.net.

### Software for the Hardware Tool

SCE provides an array of software tools for both the Linux and Windows operating systems. The samples described in this document are provided with makefiles that are compatible with either the Windows or Linux compiler that SCE provides. These samples can be executed from either the Linux or Windows debugger that connects to the hardware tool over a network connection. The following required packages are available from https://psp.scedev.net. They are accompanied by instructions which explain how to install them.

- Linux / Win32 Compiler (gcc)
- Linux / Win32 debugging environment
- Run-time libraries & documents

### Sample Code with This Document

The source code given in this tutorial is divided into two directories: one to be used with the Emulator, and one to be used with the hardware tool. The source code for both environments is very similar, as you will see in the source code examples for both environments. The main differences are in the Library Linkage options, which are specified in the makefiles.

### Differences in Executable File Formats Used in the Emulator and on the Hardware Tool

The executable file formats you use in each environment are different:

- You run .elf files in the emulator
- You run .prx files on the hardware

Before you proceed with the examples, make sure you are familiar with the general compilation and debugging process for the platform you intend to use, that is, Windows or Linux, by compiling and running some of the sample code included with the run-time libraries.

## Information and Support

- For information about the PSP, including news, updates and information about the latest library releases, refer to the following website:
  https://psp.scedev.net
- For information about developer support for licensed PSP developers refer to:
  https://psp.scedev.net/support/about
- For information about dedicated newsgroups for PSP development, refer to:
  http://www.scedev.net/psp/connect.html

This page intentionally left blank.

# Chapter 2:
# PSP Hardware: A System Tour

This page intentionally left blank.

This chapter briefly describes the aspects of the PSP system architecture that are compatible with the Emulator and the hardware tool environments, namely, the CPU and Graphics systems.

The PSP has many features that contribute to its abilities as a cutting edge portable console: a number of these are large subjects within their own right. Hopefully, you will discover the many aspects to PSP development that will enrich your knowledge and, ultimately, the games you develop.

## General Overview

The PSP is a multi-function, portable games system. Its functionality can be grouped into three general areas:

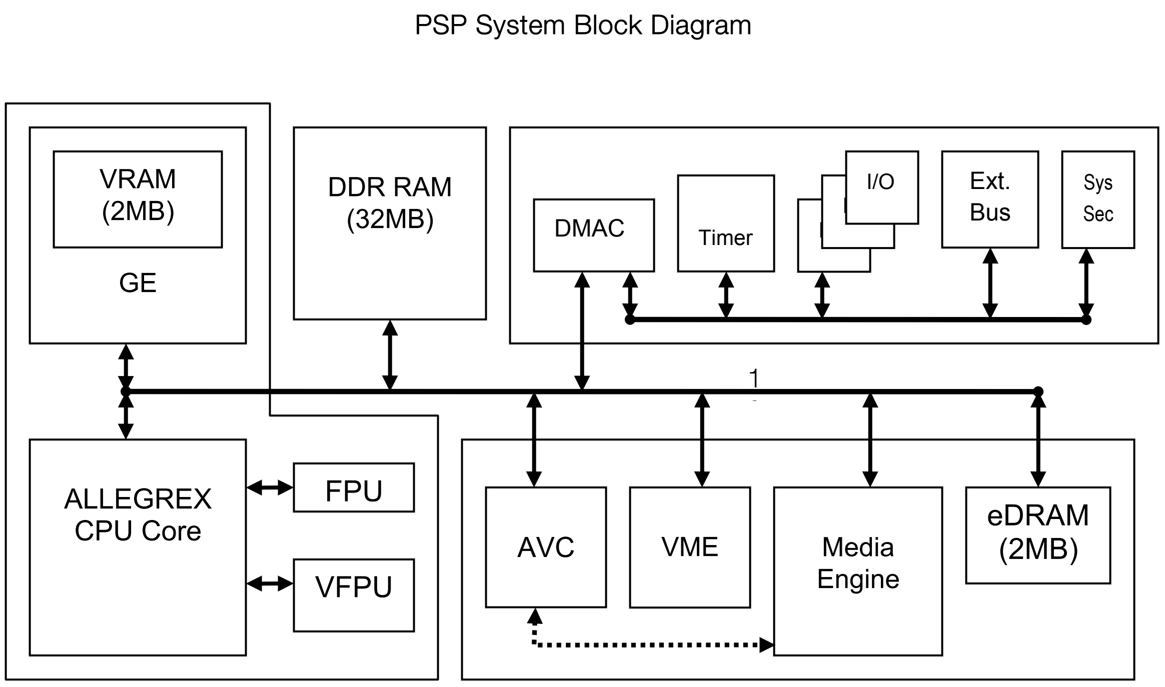
- CPU and Graphics
- Media (Audio & Video)
- Connectivity (WiFi, USB, Memory Stick etc.)

Figure 2-1 categorizes the functionality of the three main blocks of the PSP hardware as:

- the CPU/GE block
- Media Engine block
- the I/O block

These three blocks and the 32MB DDR main memory are connected by a 128bit bus.

**Figure 2-1: PSP System block configuration**



## ALLEGREX CPU Core

The ALLEGREX is the main CPU of the PSP. It is a 32bit MIPS RISC processor and has the same instruction set and exception handling as a standard MIPS R4000. However the ALLEGREX also has a separate FPU and powerful Vector Floating Point Unit (VFPU) as coprocessors. It contains two caches for instructions and data, both of 16 KB. Generally, this is a standard MIPS CPU, accepting compiled native C or C++ code quite happily. Data in memory on a MIPS processor must be aligned to a multiple of its size. The VFPU also stipulates this requirement. Chapter 5 explains the VFPU and its possible roles. Further information about the main CPU can be found in [1].

## Graphics Engine

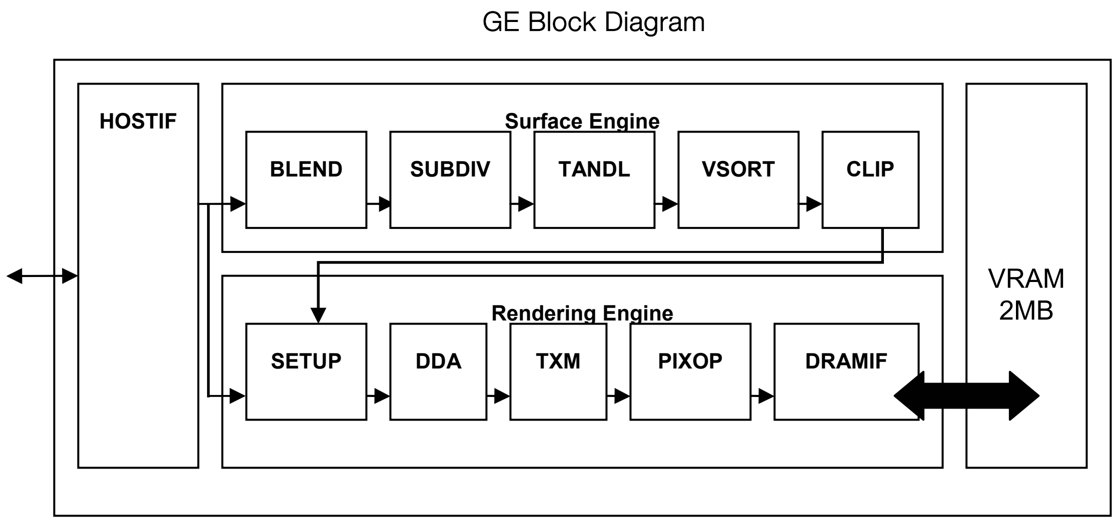
This section will explain some details about the PSP Graphics Engine (GE). As this is a custom built GPU, more information is given about it at relevant points in the document.

The graphics engine is a powerful three-dimensional and two dimensional graphics chip that performs a number of impressive operations in hardware. It is fed and instructed by display lists, which set registers within the graphics engine and inform it of various state changes and start or stop executions to draw primitives to the frame buffer. When you are writing code for the hardware, it is possible to write your own display list manager, but this functionality is provided by the SCE graphics library libGu, as mentioned in the previous chapter. More information about libGu is given later in this document.

### Inside the Graphics Engine

The graphics engine is a fixed-function graphics processor. It has hardware acceleration for a number of powerful features. The various named rectangles in Figure 2-2 are separate functional blocks that perform various parts of the graphics pipeline.

**Figure 2-2: PSP GE Internal block configuration.**



The diagram shows the separation of two main blocks: a ‘Surface Engine’, and a ‘Rendering Engine’. These are, in general, responsible for vertex and pixel operations, respectively. The host interface, HOSTIF, contains a DMA controller and a display list parser. It parses the commands in the display list and pulls the vertex data from main memory, passing it on to the various parts of the GE for processing. So, by parsing the display list commands, the HOSTIF arbitrates rendering operations to the various functional blocks inside.

As can be seen in Figure 2-2, there are two paths from the HOSTIF into the graphics pipe inside the GE. Vertex data can bypass the 'Surface Engine' portion of the GE to render primitives in screen space. This functionality relates to the two main types of vertex modes that the GE supports, 'NORMAL mode' and 'THROUGH mode':

- NORMAL mode specifies that the vertices are to be processed through the whole GE pipeline, starting in the Surface Engine block.
- THROUGH mode vertices bypass the Surface Engine; they are dealt with in the Rendering Engine block. One way to describe it is to say that NORMAL mode vertices, after being processed by the Surface Engine, are transformed into THROUGH mode vertices for the next stage in the GE pipeline.

These formats are explained in more detail in the next chapter.

Functional blocks within the Surface Engine:

- The BLEND block handles the vertex blending operations of the PSP. It is responsible for performing skinning and morphing processes, if required.
- The SUBDIV block manages all the tessellation of Bezier and Spline patch data.
- The TANDL block transforms all the vertices to the various coordinate systems: local→world→eye→clip→screen→draw space.
- The VSORT block sorts the triangle vertices, performing operations such as triangle-strip vertex ordering.
- The CLIP block performs clipping operations on the triangle data.

Functional blocks within the Rendering Engine:

- SETUP calculates all the deltas for the primitive, including the primary RGBA and secondary RGB color values, fog deltas and UV calculations, for perspective correct textures.
- The DDA unit generates the fragment information by interpolating the primitive information from the previous stage.
- The TXM unit does all the texture mapping operations including filtering and Mip-mapping.
- The PIXOP block is the rasteriser, which writes the pixels to the framebuffer. It has a direct connection to the DRAMIF, which is the interface to the 2Mb embedded VRAM. The PIXOP handles all the alpha blending operations, and other activities such as scissoring, stencil and alpha tests.

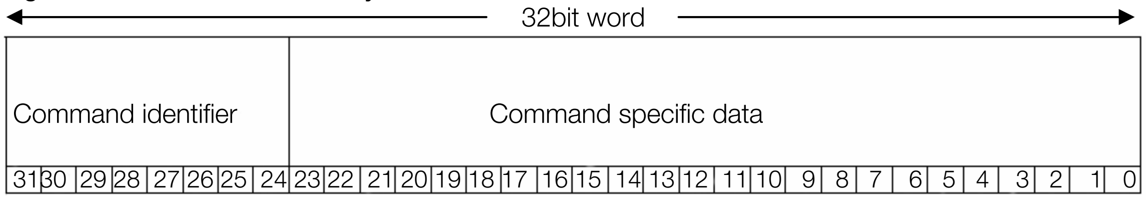
## Display Lists

This guide does not give detailed information about display list programming but it may be useful to know a little about it so that you can understand how the GE works and it's relationship to libGu.

A display list is an area of memory that contains register commands for the GE. They are the internal instruction format for the GE that control the rendering functions of the chip. These commands are 32bit fields with various bits set accordingly. When you call functions from the graphics library, the relevant commands are added to the display list. The display list is sent to the GE, where the HOSTIF parses the display list and arbitrates rendering commands to the relevant sections within the GE.

Figure 2-3 shows the format for a GE command.

**Figure 2-3: GE Command field layout**



The upper 8bits from 24 – 31 are fixed as the command identifier, which is a hexadecimal number ranging from 0x00 to 0xFF. These bits identify the command within the GE, defined in the header file *gecmd.h*.

The other 24 bits are reserved for data that each command may need, and the layout in this part is individual to each command.

## Types of Display List

There are two types of display list:

- IMMEDIATE lists are those that are sent to the GE as they are being created.
- CALL lists can be created, stored and sent at any time.

One example use of CALL would be to double buffer the creation and rendering of display lists within the applications rendering engine. This is also called “deferred rendering”, where one list is being rendered while another list is being created. This can be a more efficient method of rendering.

The uses of CALL lists like this are more suited to tutorials about optimization, and information like this is not covered in this text.

## LibGu

LibGu is the graphics API provided by SCE for you to use to develop software for the PSP. It is similar in syntactical style to OpenGL. It is not essential to know OpenGL but, if you do, it will help you to get started in programming graphics on the PSP more quickly. As time goes on, you may decide to write your own display list manager; however, this document will only cover examples of graphics programming using libGu.

### Example to Demonstrate How libGu Performs

The point of libGu is that many commands can be set with a single function call, which speeds up the development process substantially. As a quick example, let us consider the libGu function `sceGuDrawArray(...)`, referenced in [4]. Looking at the parameters in the function itself and cross-referencing commands in the Command Reference manual [16], we can determine a few commands that this function adds to the display list.

```
void sceGuDrawArray(   int prim,
int type,
int count,
const void *index,
const void *p
);
```

This function instructs the GE to draw an array of vertices as a certain primitive type, such as points, lines or triangles, and will set the following commands in the display list:

- `CMD_PRIM`

  This command tells the GE which type of primitive to draw, and the number of vertices to expect. This command will be generated with respect to the *prim* and *count* parameters.
- `CMD_VTYPE`

  This command specifies the vertex type that the GE should expect, specified with *type*. This gives the GE lots of information about the vertex format, such as whether to expect texture coordinates, and the bit-depth of the vertex data etc.
- `CMD_BASE`

  This command is always set with relation to the address of the vertices, whether in index mode or not.

- `CMD_VADR`

  This command tells the GE the memory address of the vertex data.
- `CMD_IADR`

  This command is set if indexed primitives are used and gives the GE the address for the index array when drawing in indexed mode.

All libGu functions are prefixed with "`sceGu`". Most of these functions need to be called within a display list start and finish block which is invoked as `sceGuStart(...), sceGuFinish()`, however, there are some exceptions to this rule. This is explained further in the next chapter.

Once the display list has been filled, it is sent to the graphics engine via DMA, where it is picked up by the graphics engine's host interface.

## Data in Display Lists

Vertex or texture data is never copied into a display list: this is always supplied as a pointer to the data that the graphics engine will fetch when needed. Display lists only contain commands and small values such as color information and matrices.

This is a significant difference to other APIs such as OpenGL. In OpenGL, immediate mode allows vertices to be created locally, within a function, using calls such as `glVertex3f(…)`.

Please note that vertex data is not copied into the display list itself so it is not plausible to create them locally: local vertex data will go out of scope when a function returns.

This page intentionally left blank.

# Chapter 3:
# Compiling a Simple Program

This page intentionally left blank.

## Introduction

This chapter looks at the code used in a simple program that draws a triangle on screen, and displays some text, all using libGu. We will look at the general format of functions calls to libGu, recalling some of the information covered already, and go through some of its functionality.

The code presented in this example is taken from the source file `hellotriangle.c` which you can find in the source directory for the tutorial provided with this document.

The code is provided for both the Emulator and hardware tool environments, and details about the differences can be found in Appendix B.

## Data Structures

```
static char     disp_list[0x10000];
```

The above array is a static storage area for the display list. As each `sceGu*` function is called, the data within this array will be set for each frame.

```
typedef struct  {
SceUShort16 x, y, z;
} shortVector;

shortVector gVectors[3];
```

`gVectors` is an array of a custom-defined type in which the vertex coordinates for the triangle are stored. The reason unsigned, short integers are used for this is due to the fact that this triangle is drawn with THROUGH mode. When you use THROUGH mode, the vertex positions must be in screen space and they must be of short integer type.

## Initialization

The majority of the code in this program is within the `Init()` function where we will initialize the graphics library, and set some render states that will be used in the main loop.

```
sceGuInit();
```

This is the first `sceGu*` function that must be called before anything else. It initializes the graphics library ready for use.

```
sceGuStart(SCEGU_IMMEDIATE, disp_list, sizeof(disp_list));
```

`sceGuStart(…)` defines the entry point for the display list. It is part of a few display list functions that are very important, especially if you are writing code using the hardware tool. All `sceGu*(…)` functions that you call, have to be positioned between the start and finish display list functions `sceGuStart(…)`, and `sceGuFinish()`.

The only exceptions to this are:

- `sceGuSync(...);`
- `sceGuDisplay(...);`
- `sceGuSwapBuffers();`

The reason for these exceptions is that `sceGuStart(…)` defines the pointer to the start of the display list to which consecutive commands are added before a call to `sceGuFinish()`.

All `sceGu*(…)` functions, apart from the above exceptions, will be writing commands to the display list then incrementing the display list pointer. Therefore, if your code calls `sceGu*(…)` functions outside of the `sceGuStart(…), sceGuFinish()` pair, the code will be writing commands to an unmanaged address.

Please note that the emulator will not prevent you from writing outside of a `sceGuStart(…)`, `sceGuFinish()` pair in the way described above. However, if you do this when you are using the hardware tool, you may see some unexpected results.

```
sceGuStart(      int mode,
void *p,
int size
);
```

This function defines the start of the display list, how big it is, and what mode it is to be used in. As mentioned before, you can create lists that are to be used later, or lists that are to be used immediately. The list in this sample is to be used immediately, so the *mode* parameter is set as `SCEGU_IMMEDIATE`.

The static array declared at the top of the source file is passed in as the display list data pointer, and `sizeof` is used to pass in the display list size. Note that this size is not used in any bounds-checking by libGu, and it is not dealt with internally as a ring buffer. Therefore libGu will allow you to write past the end of the display list. This will produce unexpected results, so make sure that your display list is big enough.

## Draw and Display Buffers

The following two functions initialize the variables needed to set the PSP draw and display buffers. The arguments that are passed into the functions are macros that are defined by SCE. They describe frame buffer information such as the colour depth, the width and height, as well as the memory address in VRAM that is designated as the buffer pointer start address.

The functions are setting up a 32 bit frame buffer, of standard width and height (480 * 272):

```
sceGuDrawBuffer( SCEGU_PF8888,
SCEGU_VRAM_BP32_0,
SCEGU_VRAM_WIDTH
);

sceGuDispBuffer( SCEGU_SCR_WIDTH,
SCEGU_SCR_HEIGHT,
SCEGU_VRAM_BP32_1,
SCEGU_VRAM_WIDTH
);
```

The macros `SCEGU_VRAM_BP***` found in libgu.h are relative to the frame buffer pixel format. These macros are pre-defined VRAM addresses for the draw, display and depth buffers. However, it is not necessary to include a depth buffer in the first program. This is covered in the next chapter which describes three-dimensional graphics programming.

In this example, a 32bit frame buffer is used. `SCEGU_VRAM_BP32_0` is defined as `SCEGU_VRAM_TOP (0x00000000)` cast to a void pointer. This is the memory-mapped address for the start of VRAM. This is passed in as the draw buffer start address.

For the display buffer, the pointer is calculated by multiplying the screen height (272), `SCEGU_VRAM_WIDTH` (512) and the pixel size in bytes (4) and adding it to `SCEGU_VRAM_TOP`. This provides us with the next available area of VRAM after the draw buffer. The screen width is 480 pixels wide, yet the buffers are calculated with a VRAM width of 512. This is because the buffer width must be specified in units of 128 or 256 bytes, depending on whether the pixel color depth is 16 or 32 respectively. For either color depth, the display buffer width must be set at 512.

## Viewport Initialization

```
sceGuViewport(2048, 2048, SCEGU_SCR_WIDTH, SCEGU_SCR_HEIGHT);
```

`sceGuViewPort( int x, int y, int width, int height )` defines the viewport - that is the viewport transformation from clip coordinate space into screen space. The *`width`*, *`height`*, *`x`* and *`y`* of the clip coordinate system become the equivalent in screen coordinates. For example, the first two parameters here, *`x`*, and `y`, define where, in screen space, the origin of the clip coordinate system will be. However, the origin in clip space is 0,0 – so the question is: why is that defined in the code as 2048, 2048?

The reason for this is that the overall screen space coordinate system ranges from 0 - ~4096 in both *`x`* and *`y`*. Hence 2048, 2048 is the centre of screen space, therefore objects in clip space will be transformed into screen coordinates correctly.

`sceGuOffset( int x, int y )` is the final transformation into *draw space*, that is, a subsection of screen space that resides in the xy ranges 0 – 1023. The following example shows screen space to draw space transform for the *`x`* coordinate.

```
Xd = Xs - OFFSETX
```

In the same way that the *`x`* origin is defined as 2048 after the viewport transformation into screen space, we can see how the centre of *`x`* in screen space (2048) becomes the centre of *`x`* in draw space (240):

```
Xd = 2048 - OFFSETX
Xd = 2048 - ((4096 - SCEGU_SCR_WIDTH) / 2)
Xd = 2048 - 1808
Xd = 240
```

`sceGuScissor( int `*`x`*`, int `*`y`*`, int `*`width`*`, int `*`height`*` )` defines the scissor region. The scissor test is the first pixel test to determine whether a fragment gets rasterised. Only pixels within this area will get drawn.

## Finalizing the Initialization Section

`sceGuColor( unsigned int `*`col`*` )` sets the drawing colour. You could think of this as a constant vertex colour that can be applied to geometry that doesn't have explicit per-vertex colour. So that when a large model is being rendered, if the vertex colour doesn't change across the object, this function can be used and the colour data can be removed. However, this colour value is over-ridden by explicit vertex colors.

`sceGuFinish( )` adds a command to the list to signal the end of drawing if in IMMEDIATE mode, and adds a return command if in CALL mode.

`sceGuSync( `*`int`*` mode, `*`int`*` block )` This function will synchronize drawing depending upon the parameters passed in. In this case we are asking it to wait until the GE has finished rendering.

The last function in the initialization phase relates to the V-Blank. This is one of the few functions in this program that is used differently on the Emulator to how it is used on the hardware tool. It simply waits for the beginning of the next vertical blank before proceeding. Further information about differences between the Emulator and the hardware tool are described in Appendix B.

## The Main Loop

After initialization, the main functionality of the program simply loops through three functions:

```
StartFrame();
Render();
EndFrame();
```

As stated previously, all `sceGu*` functions, apart from a couple of exceptions, must appear within a `sceGuStart`, `sceGuFinish` pair. In this instance these 'containing' functions appear in `StartFrame` and `EndFrame`.

`sceGuClear( int mask )` clears the relevant buffers in VRAM. The function arbitrates between colour, depth, stencil, or all three, to be cleared, depending upon the mask passed in. These are defined in libgu.h. In this case, it is just clearing the colour buffer, so the example sets the `mask` to `SCEGU_CLEAR_COLOR`.

`sceGuDebugPrint( int x, int y, unsigned int col, const char* str)` prints a message to the screen. The x and y arguments are the coordinates in draw space for the start of the text. `col` is the colour, defined as a 32bit field, eight bits per component. `str` is the string that we would like to print to the screen, in this case, “Hello triangle”.

`sceGuDrawArray( int prim, int type, int count, const void *index, const void *p)` is the function that was deconstructed earlier into GE commands. It is the standard interface to make the GE render most primitives in libGu. This function tells the GE about the type of primitive to draw, what data to expect in the vertex format, how many vertices are to be rendered, and whether to render indexed primitives or not. It passes in a pointer to the vertices themselves for the GE to fetch, and a pointer to the indices if this mode is being used.

If the display list is in IMMEDIATE mode, such as this sample, the GE will start parsing the display list directly and render the primitives.

In this case, we tell the GE that the vertex is of `short` type and should be rendered in THROUGH mode, with `SCEGU_VERTEX_SHORT`, OR-ed with `SCEGU_THROUGH` respectively. This parameter is passed in as *`type`* . This example draws a triangle, so we specify the *`prim`* type as `SCEGU_PRIM_TRIANGLES`, and we define the number of vertices to draw, being three.

## Vertex Formats and How They Are Interpreted by the Graphics Engine

As stated in [3], the GE accepts a fixed order vertex format. The order of the elements within the vertex format is also documented in [3]. The minimum that the graphics engine needs to define a vertex is the position vector. As stated, the *`type`* field in `sceGuDrawArray(…)` is used to tell the graphics engine what to expect. This is an OR-ed bitfield that the graphics engine will interpret to arbitrate offsets within the vertex data. This means what you define as a vertex is flexible even though the order of the various vertex elements is fixed.

The final function call to describe is: `sceGuSwapBuffers()`. This call swaps the draw and display buffers, presenting the frame that we’ve been rendering to the screen.

## Indexed versus Non-Indexed Primitives

When calling `sceGuDrawArray(...)` as in this sample, there is another pointer argument that can be passed in that we have not covered: `const void *`*`index`*. This parameter is a pointer to a list of indices, used for rendering with indexed primitives.

Indexed primitives are a method of fetching vertices to render based upon its particular index in an array. This method can save bandwidth on hardware that has a vertex cache, however the GE has no vertex cache and, therefore, we do not recommend using indexed primitives. The reason for this is that, as the GE is a ‘pull’ style chip, it will fetch the index from the index pointer, and then each corresponding vertex from the vertex pointer. This results in more bus activity, and that is best kept to a minimum otherwise performance is affected.

## Summary

This chapter described a simple program you can use to get started, including how the rendering context is initialized, which is essential for drawing, how a simple render loop is set-up to display text and draw a triangle, and it also explained all the libGu commands that were used to achieve this.

The example in this chapter demonstrated two-dimensional graphics programming. The following chapter describes three-dimensional graphics programming and introduces another library: libGum. The following example builds from the sample code in this chapter to create a simple scene with lighting. It also includes controller input so that a user can interact with the PSP using the example program.

This page intentionally left blank.

# Chapter 4:
A Simple 3D Scene

This page intentionally left blank.

## Introduction

This chapter describes the code that creates the three-dimensional scene example that accompanies this document, the code can be found in the file: `3dscene.c`.

This sample creates a simple three-dimensional scene, consisting of a floor plane, a pair of donuts and a light. The objects in the scene are rendered with the GE hardware patch tessellation. As this sample produces three-dimensional effects, it involves additions to the setup calls for libGu.

This sample also uses an additional library, mentioned in the Introduction, called libGum, and adds user interactivity in the form of controller input using the controller library.

This chapter aims to explain some of the theory behind the functions used in the example. For example, the description of some of the PSP lighting functions is accompanied by a brief explanation of the lighting model that the GE adheres to in order to provide more insight into the relationships between the functions and the hardware. Hopefully, this information will provide the reader with a good overall view of some graphics features of the PSP, and how easy they are to implement.

## Depth Buffer

The PSP GE has a hardware depth buffer, or Z-buffer, fixed at 16bit. You must explicitly define the area of VRAM that the depth buffer occupies: this is not done automatically. You do this in a similar way to how you define the area of VRAM that the draw and display buffers occupy, using `sceGuDepthBuffer( void *zbp, int zbw )` where in this case *`zbp`* is `SCEGU_VRAM_BP32_2`. This is defined in libgu.h as the beginning of the next area of free memory, after both the draw and display buffers, when set to 32bit.

As the depth buffer width is 16-bit, it must be defined as a multiple of 128, so *`zbw`* is defined as 512 with the macro `SCEGU_VRAM_WIDTH`.

The Z-buffer is used mainly for hidden surface removal, by testing pixels and rejecting them if they are behind ones already present. The Z-buffer can choose to draw or reject pixels under a variety of tests. This example will draw pixels when the depth value is greater or equal to the value in the depth buffer, set with `sceGuDepthFunc(SCEGU_GEQUAL)`.

The next initialization necessary is to set the depth range. The values set in here become the depth range mapped by the viewport transform. Any pixels outside of this range will not get rasterized. The Z-buffer passes or rejects fragments based on the result of comparing the z values of each pixel. If you set a very small depth range, Z-test artefacts will occur when the pixels are rasterised, as the pixel depths will be comparatively far apart. This example sets the depth range to the default setting of 65535 for the near z value, and 0 for the far z using `sceGuDepthRange(int nearz, int farz)`.

`sceGuClearDepth( int depth )` sets the value that the GE will set into the Z-buffer when clearing it.

Finally we enable the depth test with `sceGuEnable(SCEGU_DEPTH_TEST)`.

## Three-dimensional Coordinate System and libGum

As stated in [3], the GE uses six coordinate systems in the pipeline to render primitives to the screen. In the previous chapter, the example showed how primitives are rendered in THROUGH mode, which requires vertices specified in *draw space*.

This 3dscene sample uses the full three-dimensional transform set and therefore uses all six coordinate systems to transform the 3D primitives that are rendered to the screen. When programming the PSP, you do not need to manually transform primitives through the transform pipeline. The GE handles all of this functionality in hardware, and this is exposed through libGu and libGum.

There are four matrices that you have access to; these are the perspective, view, world and texture matrices.

As stated in the Graphics Engine user manual, the above four matrices make up the pipeline to transform an object from three-dimensional space to render it to the two-dimensional screen and the ways in which they work are:

- The **world matrix** transforms an object from local space to world space
- The **view matrix** transforms objects from world space to view space
- The **perspective matrix** transforms the eye space object to clip space
- The **texture matrix** is used for effects such as projected textures, which are not covered in this guide.

## LibGum

LibGum [4] is essentially a matrix manager and math library that adds some useful functions such as matrix transforms and stacks. If you are familiar with OpenGL, you will notice that there are similarities in the syntax of OpenGL and LibGum. The matrix transform functions can be summarized as follows:

| | |
|---|---|
| • `sceGumTranslate(...)` | applies translation operation |
| • `sceGumScale(...)` | applies scaling operation |
| • `sceGumRotate*(...)` | applies a rotation in x, y, and /or z |
| • `sceGumLoadIdentity(...)` | resets the matrix to identity. |

These functions act as consecutive transforms on the current matrix so, when functions are called, the next transform is multiplied into the current matrix. These matrices are column order, meaning that the operations are post–multiplied so, when you think about the order of transforms, you must consider them back to front.

Stack functionality can be useful when a transform needs to be retained and restored later. Some objects may share a base transform, but they will require further individual transforms. For example, the base transform can be pushed on to the stack, then each object can add their own transforms, restoring the base transform for the next object. In this way it is possible to save processing of the base transform for each object. A simple example of this is demonstrated in the 3dscene example, covered later.

You initialize a matrix stack on PSP by first declaring an array of 4x4 matrices. There is no theoretical limit on how many you can declare here. The function to set the matrices for the stack is:

```
sceGumSetMatrixStack(  ScePspFMatrix *m,
                       int proj,
int view,
int world,
int tex
);
```

Usually, the Perspective matrix is likely to be set once per application and the View matrix is likely to be set once per frame. Since this sample does not modify the Perspective matrix during the application, it is set once in the initialization phase.

`sceGumMatrixMode( int mode )` Sets the current matrix stack. All consecutive libGum matrix functions will apply transformations to the top level matrix on the stack, which becomes the current working matrix.

The sample sets the current matrix to be the projection matrix, then calls `sceGumPerspective( float fovy, float aspect, float near, float far)`. This function creates a Perspective matrix with the parameters and multiplies it into the current working matrix.

The GE is capable of flat and Gouraud (per vertex interpolated across the triangle) shading. The way to specify whether to use flat or smooth shading is with the function `sceGuShadeModel( int model )`, specifying `SCEGU_FLAT`, or `SCEGU_SMOOTH` respectively as the *`model`* parameter.

On PSP, lighting is performed in hardware. There is a maximum of 4 hardware lights that are exposed through libGu, and the next section will look at how to set them up, and explain a little about some of their properties, and materials.

## Lighting

All shading calculations, whether flat or Gouraud, require normals as part of the lighting equation. For flat surfaces, face normals are required, for Gouraud shading per-vertex normals are necessary. This sample uses patches to render the geometry. By using this function, the GE generates all the normals automatically. As lighting is done on a per-vertex basis, a reasonably highly tessellated mesh is needed for adequate results.

The GE lighting model calculates the lighting based on four parameters: ambient, diffuse, specular and emissive.

- The *ambient* light contribution is thought of as the base environment lighting contribution. This is light that is so spread and reflected from the scene that is has become omni-directional, and therefore appears flat.
- *Diffuse* lighting is directional, in that the contribution is calculated from a specific source. Diffuse lighting is matte in appearance, and will appear the same across the object from any view angle. The definition of this is that diffuse lighting is *view-independent*.
- *Specular* lighting, like diffuse light, is directional, and it appears to reflect off the object it hits in a particular direction. Specular lighting is one way of providing a shiny appearance to a surface. Specular lighting is calculated with the light position, the surface normal, and the camera viewpoint. It will appear to move across the surface as the object or the camera moves and is termed "*view-dependent*".
- The *emissive* factor of the equation is thought of as light originating from the object itself, as if it had an inner glow. This is added to the overall contribution to light the surface, yet it doesn't actually emit light that is picked up by other surfaces.

All of the above factors are combined with the material properties for the surface, which will be covered later.

The GE will calculate up to four vertex lights in hardware, and each of these can be directional, point or spotlight. All of the lighting functionality is exposed by libGu. Therefore, it is very easy to attach lighting effects, which will give greater realism and add depth to the rendered scene.

This sample will set up a single diffuse and specular point light to illuminate the scene. The light color is set with `sceGuLightColor( int n, int type, unsigned int col )`. In this case *`n`* is the light identifier number from 0 – 3, *`type`* is the component of the light equation; ambient, diffuse, or specular and *`col`* is the color to define for each component.

`sceGuSpecular( float power )` sets the specular exponent of the light. This is equivalent to setting the shininess value of the material properties of the surface to be rendered. The higher the value, the shinier the surface will appear to be, and the specular light will appear smaller and more focused. This sample is assuming that all materials will have the same shininess value.

The two requirements for diffuse and specular lighting are a light position and a view position. The view position is covered later in this guide, as this is set on a per-frame basis. There is no need to move the light in this example, so the position is set in the initialization phase with the function `sceGuLight( int n, int type, int comp, const scePSpsFVector3 *vec )`. The sample defines what *`type`* of light it is with `SCEGU_LIGHT_POINT`. The type of lighting it provides is set to

`SCEGU_DIFFUSE_AND_SPECULAR` using the *`comp`* parameter. The position vector passed into this function has to be defined in world space for the lighting to work correctly – this is passed in as the *`*vec`* parameter.

Finally, light ‘0’ is enabled with `sceGuEnable (SCEGU_LIGHT0)`. Lighting itself is enabled for all successive rendering with the call to `sceGuEnable(SCEGU_LIGHTING)`. This concludes the initialization for this sample.

## Patches

The GE’s powerful feature of hardware patch tessellation is used in this sample. The following section provides a brief description of the Bezier and Spline surfaces that are involved, and explains some of the advantages of using them. However, this is not intended to be a mathematical tutorial about splines and their use. The mathematics involved can be found in text [5], listed in the Reference section at the end of this document, and text [3] also contains lots of information about the methods the GE uses to calculate the patch surface.

Both Bezier and the other Spline surfaces, that the GE supports, are types of parametric surface. Parametric surfaces are generally defined by a series of control points that form the basis of the surface calculation. The surface itself, however, does not necessarily pass through these control points, depending upon the equation that is used in the surface calculation.

The control points used to generate the surface can be thought of as a kind of cradle, or mesh that is an outer, low resolution representation of the surface that will be produced. The GE will take a set of control points and interpolate across them to produce the triangulated surface representation that is rendered. In this way, the vertex data for each triangle does not need to be given to the GE, which dramatically reduces the quantity of data that is transferred across the bus. The GE calculates the normals required for lighting the patch, so their inclusion in the vertex format is also not required. Texture coordinates for the triangles are interpolated across the surface via those provided per control point.

The sample initializes some position vectors for the control points. Positions to describe a plane, a sphere and a torus are created.

In hardware code, at the end of each spline initialization function, a call to `sceKerneDcacheWritebackAll()` is made. This function makes sure that everything in the Data Cache (D-Cache) is written back to main RAM before continuing. This is necessary sometimes – in general more with dynamic data. The reason for this is that the libGu functions such as `sceGuDrawArray(…)` will initiate a DMA transfer to the GE that is asynchronous with the main CPU. As the GE will start fetching the vertex data as soon as it is instructed, it will start drawing vertices that may not be ready, as initialized data may still be residing in the D-Cache. This will generally result in strange artifacts, such as vertices in the wrong place. They may even be way off screen.

It is possible to manage the D-Cache in such a way that you don’t have to write all of the data back. However, tracking the D-Cache is a complicated procedure and, therefore, it is not covered here. Further information about the Cache functions can be found in [17].

## Controller Handling

This section gives detailed information about the functions that access controller input data. There are important differences between the hardware tool and the Emulator in this respect: they do not share the same API. All the code to handle user input in this sample, and the following sample, is contained in the function `ReadPad()`; the platform-dependant code contained in `ReadPad()` is described next.

## Controller Input for the Emulator

The way that the Emulator handles code for the Controller is very straightforward, and is part of the emulator library. Therefore there is no need to include a specific header. The function used in the sample is listed here:

```
unsigned int sceEmuAnalogRead ( unsigned char* buffer );
```

The above function gathers information about the button states and the analogue stick movement. It returns an unsigned integer as a bitfield that contains information about which buttons have been pressed. The `unsigned char*` *`buffer`* parameter is a pointer to an array of four unsigned 8bit integers. `sceEmuAnalogRead` fills these in with information about analogue sticks. There is support for up to two analogue sticks. This example uses the left hand analogue stick for input, the data being stored in the last two elements of the array for *`x`* and `y` after `sceEmuAnalogRead` returns.

## Controller Input for the Hardware.

The controller handling code for the hardware is a separate API and therefore a different header must be included:

```
#include <ctrlsvc.h>
```

It is possible to set the controller library to distinguish between servicing digital buttons only, or to include the analogue sticks in the service. This is to avoid processing the analogue stick if it is not needed in an application. The sample initializes the controller service library with a call to:

```
sceCtrlSetSamplingMode( unsigned int uiMode );
```

In this case, the sample is demonstrating the use of both the analogue and the digital controls, so the parameter passed into the function is `SCE_CTRL_DIGITALANALOG`. Within `ReadPad()`, in the hardware source code, one function gathers the required information about the controller state:

```
sceCtrlReadBufferPositive(    SceCtrlData *pData,
int nBufs
);
```

The `SceCtrlData` data structure is designed to contain the information required to determine the controller input parameters:

```
typedef struct SceCtrlData {

   unsigned int  TimeStamp;
   unsigned int  Buttons;
   unsigned char Lx;
   unsigned char       Ly;
   unsigned char       Rsrv[6];

} SceCtrlData;
```

A call to `sceCtrlReadBufferPositive(...)` fills in this data structure. To then extract the button information, use AND on the results against the relevant bit mask.

Data for the analogue stick is encoded within 8 bits each, in the *`Lx`*, and *`Ly`* elements of the structure for the x and y axis, respectively. This includes both positive and negative information, encoded into the `unsigned char` in such a way that from 0-127 is the negative value, and 128-255 is the positive value per axis. The sample converts this data into the -1.0 to +1.0 ranges, which is a standard method for reading controller input.

A dead-zone is also set up that in this case is 30% of movement from the centre, as analogue controller will generally exhibit a certain degree of inaccuracy. This inaccuracy may result in a ‘floating movement’, where the controller appears to stick in a certain direction after it has been released.

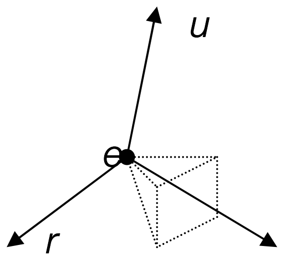
User interaction in this case simply increments rotation angles that are used by the camera to rotate around the scene. This is covered in the next section that deals with the view transform within the render loop.

## The View Transform: A Simple Camera

The first thing to do for each frame is to set the view transform. As explained previously, specular lighting requires the view position to be set. This is encoded within the view matrix. Since the view is being moved in this sample, we must set this every frame. The view matrix is made up of four vectors that represent information for a camera. The first 3 vectors are the cameras 'basis' which is defined as a set of three orthogonal vectors of unit length. This is otherwise known as an "orthonormal basis". This basis forms the rotation part of the matrix. The fourth vector is the position in world coordinates of the camera.

A simple way to define a view matrix is to invert a world matrix composed only of rotation and translation elements. Introducing scale operations into this would make the operation fail as the matrix will cease to be orthonormal.

**Figure 4-1: Camera and View Matrix Relationship**



$$\begin{bmatrix} r_x & u_x & -n_x & e_x \\ r_y & u_y & -n_y & e_y \\ r_z & u_z & -n_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

Firstly, `sceGumMatrixMode(...)` is called to set `SCEGU_MATRIX_VIEW` as the current working matrix, which is then set to identity. Translation and rotation functions are then applied to the quantities required away from and around the scene using `sceGumTranslate(...)` and `sceGumRotate(...)` respectively. The rotation values passed in are the variables modified by user input through the controller.

This matrix is retrieved with `sceGumStoreMatrix( ScePspMatrix4 *m )` for inversion. There are no native matrix inverse functions in either libGu or libGum, so one is provided here. The matrix inverse function only needs to invert 12 of the 16 elements of a 4 * 4 matrix as the internal format for the view matrix used by the GE is 4x3.

This matrix, after it has been inverted, represents a camera looking at the origin rotated around it, a certain distance away. `sceGumLoadMatrix(ScePspMatrix4 *m )` then loads it back as the top–level matrix in the view stack. This is a simple and quick way to define a camera for the view matrix.

For the remainder of this frame, operations with the model matrix are used. This is set as the current matrix and is set to identity. Lighting is enabled, as the sample will disable it towards the end of the render loop to draw the light representation.

The first object to be rendered in this scene is the floor plane.

## Materials

Material colors form the other part of the overall lighting equation that determines the final colour of a vertex when lighting is enabled. The full equation for lighting is documented in the Graphics Engine User manual [3] (p.56). To avoid reproducing this information, it is only necessary here to explain that the material colors are multiplied into the equation at each level; i.e. a diffuse lighting component is multiplied in with the material diffuse component and the specular lighting component is multiplied with the material specular component, and so on. Generally, materials can simply be thought of as the colour of the object to be rendered. However, you can get subtle effects with these material parameters, explained later in this chapter.

**Equation 4-1: Light and Material Contribution to Vertex Colour**

$$\begin{aligned} Colour &= Emissive_{mat} + Ambient + Diffuse + Specular \\ &= Emissive_{mat} \\ &+ Ambient \times Ambient_{mat} \\ &+ Ambient_{light} \times Ambient_{mat} \\ &+ Diffuse_{light} \times Diffuse_{mat} \times \left( \frac{\max(L \bullet N, 0)}{|L||N|} \right) \\ &+ Specular_{light} \times Specular_{mat} \times \left( \frac{\max(H \bullet N, 0)}{|H||N|} \right) \end{aligned}$$

Equation 4-1 shows the complete equation for light and material contributions to a vertex colour. This is defined here for a single light. For multiple lights the equation would sum the light and material contributions for each light.

The function `sceGuMaterial( int` *`type`*`, unsigned int` *`col`* `)` is used to set the material colour for the succeeding vertices, where *`type`* is either `SCEGU_AMBIENT`, `SCEGU_DIFFUSE` or `SCEGU_SPECULAR`, and *`col`* is a 32bit integer colour field.

## Patches

The floor plane is rendered with the GE's hardware spline surface functionality. In order that the render functions use the correct matrices to transform in libGum, there are duplicates of the relevant `sceGuDraw*()` functions from libGu in libGum, such as:

```
sceGumDrawSpline( int type,
int ucount,
int vcount,
int uflag,
int vflag,
const void* index,
const void *p
);
```

This function has some parameters in common with `sceGuDrawArray(…)`, covered earlier, in the *`type`*, `*`*`index`* and `*`*`p`* arguments. Parametric surfaces are defined in general in a two-dimensional notation. Like the convention for two-dimension texture coordinates, they are defined as ranges in *`u`* and *`v`*. *`ucount`* and *`vcount`* are the respective numbers of control points in each direction. In this example, an array of 10 * 10 vectors was set up as the grid of control points, so *`ucount`* and *`vcount`* are each 10.

The *`uflag`* and *`vflag`* parameters relate to the way patches are constructed. These values relate to knots that are important in the calculation of B-Spline surfaces. The knots can be set to open or closed in both the start and end of *`u`* and *`v`*. The effect of this is that if they are set to open the patches behave like Bezier patches. B-Spline surfaces will generate more triangles per control point, though these will cover a smaller area, which means that the created surface will be of much greater density. Beware of this when creating a large area of B-Spline surfaces, as tessellation should be managed accordingly, or performance may suffer.

The patch rendering hardware will accept subdivision levels between 1 and 64 in either *`u`* or *`v`* directions. Higher numbers produce smoother geometry but require more processing, and will therefore take longer to render. You need to decide how you want to balance quality against performance. The sample sets the subdivision levels separately for the floor plane and the torii with `sceGuPatchDivide( int udiv, int vdiv )`.

More detailed information about the advantages, disadvantages and idiosyncrasies of Bezier vs. B-Spline patches, including the mathematical notation, can be found in the Graphics Engine User Manual [3] and in the texts referenced [5].

## Matrix Transforms

The next set of world transforms provide a simple demonstration of the matrix stack that was introduced earlier in the chapter.

The control points for the torii set up are in model space, situated about the origin. For this example, they both share a base translation matrix. This is set with a call to `sceGumTranslate( &offset )`. The first Torus that is drawn provides a rotation that is post-multiplied to rotate it while still in model space. This rotation will make the torus appear to be lying horizontal. As this rotation will interfere with the base transform used by Torus #2, the matrix is first stored on the stack to preserve it. The Torus is then rendered with a call to:

```
sceGumDrawBezier( int type,
int ucount,
int vcount,
const void *index
const void *p
);
```

This function is identical in syntax to the Spline equivalent, except that you do not need to specify whether the patch is open or closed. A call to `sceGuPopMatrix()` then restores the saved matrix to the top of the stack.

The second matrix rotates in an orbit around the ring of the first matrix. The transform for this is to translate a certain distance from its origin to fit snugly around the other. This will offset it from its origin, so then applying a rotation to this will complete the transform. Remember that these are post-multiplied, so the transforms have to be declared in reverse order. The Torus is then rendered.

As mentioned earlier, object materials can be used to affect the way an object appears and subtle use can produce pleasing effects. In the case of the torii, the specular value has been declared as a very dark grey. As this is multiplied into the lights specular value, the intensity of the final specular component is reduced to produce an agreeable waxy finish.

The final object in the scene is the light representation. A simple white sphere is used, rather than anything more elaborate. To achieve this, we disable lighting and set the drawing color to white. The GE will now stop calculating vertex colors via lighting, and use the constant drawing color. The model matrix is cleared to identity and the light position is applied as a translation before rendering. This concludes the 3dscene sample.

## Summary

This chapter explained the introduction of the third dimension into the sample code. We set up the depth buffer and view matrix, and introduced another library, libGum. The PSP hardware features of lighting and patch tessellation were introduced, and the code demonstrated a simple three-dimensional scene with only a few lines of code. The next chapter introduces the VFPU, and describes how to enhance the current sample with a VFPU assembly code example.

# Chapter 5:
# VFPU (Vector Floating Point Unit)

This page intentionally left blank.

## Introduction

In the previous chapters, the source code that has been explained has been compatible, with a few exceptions, on both the hardware tool and the Emulator. This has been a conscious decision in the design of the document. However, the VFPU is not emulated; therefore the source code described in this chapter is not supported on the Emulator.

This chapter introduces the VFPU and explains its relationship to the rest of the system. It also discusses some applications of the processor, and how to initialize it within an application. It will then explain how the VFPU is used for a simple math function that enhances the previous example. This example should help you to start writing VFPU code.

## Assembly Language

The PSP VFPU is not programmable in a high level language such as C or C++. This means that the programmer must use the VFPU assembly language instruction set. This document does assume some previous assembly language knowledge – at least to an introductory level. Good books to read for relevant assembler tutorials and explanations are [6], [7]. The chapter will only be looking at the specific assembler opcodes that are present in each function, and will avoid explicit explanation about such topics as gcc inline assembler syntax. This information is included in Appendix A.

## The VFPU

The VFPU is a powerful 128bit Vector Floating Point Unit. It acts as a coprocessor to the ALLEGREX CPU, and it is very well suited to maths functions such as matrix and vector operations. It contains 128 32bit registers that are all available for your use. These are called the 'Matrix register file', and are covered in the Matrix Registers section, below.

The VFPU can only run in coprocessor mode, so it is unable to run parallel to the ALLEGREX. However, it is much faster at calculating vector operations than the main CPU, so it can significantly improve performance. The final example in this introduction builds upon the previous example by using the VFPU to rotate the light source around the scene.

## Libvfpu

SCE provides a C library with source code in the tool-chain, named "libvfpu". It contains a full set of maths functions including 2, 3 & 4 element vector and matrix operations all using inline VFPU assembly code. Source code can be found in `/usr/local/psp/devkit/src/vfpu`. The next section contains a description of the functions used in the next sample.

To use libvfpu functions within a program, you must `#include libvfpu.h`, and link the application with the libvfpu in the makefile by adding "`-lvfpu`" to the list of linked libraries.

The PSP kernel software has extensive multi-threaded capabilities, and will manage threads in a priority-based system. It manages the VFPU registers within a thread context so, when threads are switched, the VFPU registers are saved and restored. As the VFPU context contains information about the 128 registers, plus more information, performance can be affected when saving and restoring the VFPU context. To solve this, the PSP kernel includes functionality to specify whether or not a thread will use the VFPU during its lifetime. The default for this is "off", and if the thread attempts to access the VFPU without enabling it, a coprocessor exception will occur and the program will terminate. However, enabling the VFPU in a thread is quite a simple procedure. It is explained here.

Execution starts at `main()`, which is the only thread that this sample needs to manage. Therefore, the only requirement is that the VFPU is enabled at some point before we use it. The function used is the first one called inside `main()`.

`sceKernelChangeCurrentThreadAttr( SceUint *clearAttr*, SceUint *setAttr* )` changes the attributes of the calling thread. In this case, there is no need to clear any specific attribute, so zero is passed in as the first parameter.

`SCE_KERNEL_TH_USE_VFPU`, passed in as the second argument, will inform the kernel to set the VFPU attribute for this thread. This completes the thread management for this sample. If you want to learn more about the PSP kernel and the thread library, we recommend that you study the texts listed in the Reference section:[9] and [10], in which much more detailed information is given.

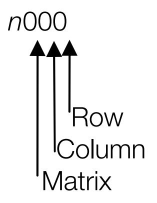
## Matrix Registers

In order to write VFPU code, it is worth learning a little about the matrix registers in the VFPU chip itself. The matrix register file in the VFPU consists of 128 32bit registers that are accessible in a variety of ways. Single scalars, 2, 3 and 4 element vectors are supported. Also, it is possible to access the registers as 2x2, 3x3 by 4x4 matrices. All of the vector and matrix accessibility options are available in row and column format. This can at first appear like a bewildering array of options, especially when you first see them in the VFPU User Manual [8].

The matrix register file is a sequential list of registers named $0 - $127. Each register is a 32 bit word.

One way to imagine the register file is to think of it as eight 4x4 matrices sitting next to each other. In this view, the top row of all eight matrices are the continuous set of registers from $0 - $31, the second row is from $32 - $63 etc. This corresponds well to the access naming convention that the VFPU assembly language prescribes - a four digit alphanumeric string with the following properties:

**Figure 5-1: VFPU Matrix Register Naming Conventions**



*n* = S, C, R, M or E

S = Scalar
C = Column Vector
R = Row Vector
M = Row Matrix
E = Column Matrix

So, for example, a matrix register access code of S000 would mean the scalar in the first row, of the first column, of the first 4x4 matrix.

Defining vectors and matrices is a little more involved, as the numbers translate to where the vector or matrix *starts* in the register file, and the elements will, if available, wrap around to define the other components.

**Figure 5-2: The M000 Access Code**

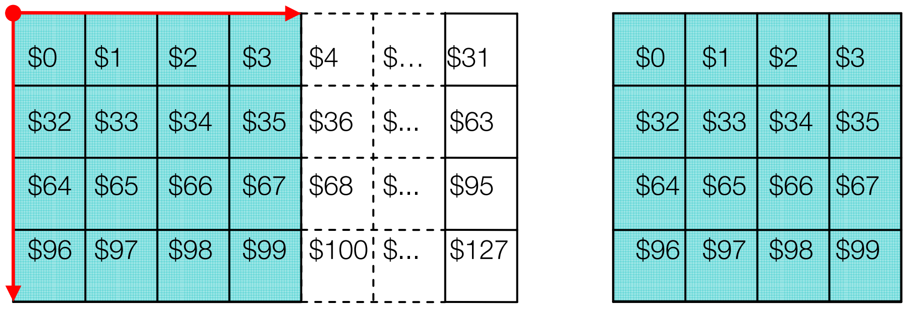


Figure 5-2 depicts the M000 access code. It is a row matrix starting from the first row, of the first column, of the first matrix in the matrix register file. The matrix on the right shows the resulting matrix register layout from M000.

**Figure 5-3: The M012 Access Code**

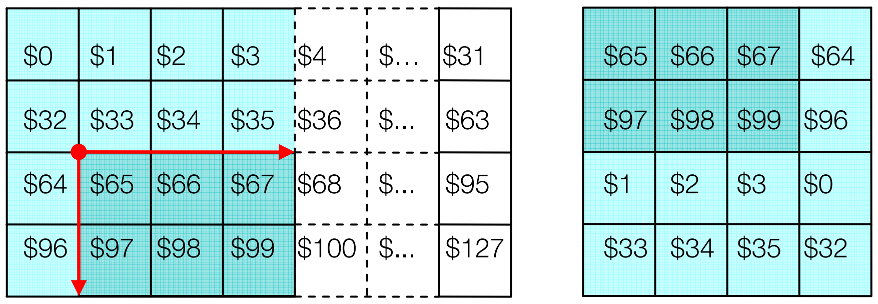


Figure 5-3 represents the matrix access code M012 which is a row matrix defined to start in the third row, of the second column of the first matrix. As the diagram shows, the start position for this matrix is offset into the matrix register file, and the lighter shade shows the values that wrap. The matrix to the right shows the resulting matrix element layout from M012. However, not all configurations are possible and the full list of access codes, with their corresponding element descriptions, is documented in the text listed as [8] in the Reference section.

## The Code

This section describes the VFPU functions that are used from libvfpu in this sample. It looks at the assembly code instructions in detail.

### The Functions

The light position is a vector in world coordinates. It is not subject to the GE transforms in the way that geometry is. As such we have to move the light position vector manually, and then call `sceGuLight( int n, int type, int comp, const scePSpsFVector3 *vec )` to re-define the lights position. The aim here is to rotate the light clockwise around the floating torii. To do this we need to create a 3x3 rotation matrix around the Y axis, and use this to multiply the light position vector. Three functions from libvfpu can help to achieve this.

```
sceVfpuMatrix3Unit( scePspFMatrix3 *pm0 );
```

This function takes a 3x3 matrix pointer and sets it to identity. There are only two VFPU instructions of note here, covered next:

```
__asm__ (
    ".set        push\n"
    ".set        noreorder\n"
    "vmidt.t     e000\n"
    "sv.s        s000,  0 + %0\n"
    "sv.s        s001,  4 + %0\n"
    "sv.s        s002,  8 + %0\n"
    "sv.s        s010, 12 + %0\n"
    "sv.s        s011, 16 + %0\n"
    "sv.s        s012, 20 + %0\n"
    "sv.s        s020, 24 + %0\n"
    "sv.s        s021, 28 + %0\n"
    "sv.s        s022, 32 + %0\n"
    ".set        pop\n"
    : "=m"(*pm)
);
```

The example above reproduces the VFPU code from this function. The greyed-out areas are the standard gcc inline assembly syntax that is used in all the examples. Some aspects of the syntax are explained throughout this chapter but it is covered in more detail in Appendix A.

Most VFPU instructions are prefixed by, or will contain, a "v" for "vector" to differentiate them from ALLEGREX or FPU instructions. When necessary, there are also suffixes to an instruction that differentiates between the quantities of vector elements for which the instruction is valid. The suffixes are:

| | |
|---|---|
| s | relates to scalar elements |
| p | relates to pair elements |
| t | relates to triple elements |
| q | relates to quadword elements |

The names of instructions are generally acronyms for the operation that they perform.

```
"vmidt.t        e000\n"
```

The `vmidt.t` instruction creates a 3x3 identity matrix and stores it in the matrix register file starting at the location specified. In this case, the "`midt`" portions of the instruction relate to "Matrix Identity", and we can see from the "`t`" suffix that it is working with triple words, that is, a 3x3 matrix. Some acronyms are more easily recognizable than others at first.  However, they will soon become familiar to you.

```
"sv.s                 s000,  0 + %0\n"
```

This instruction stores a single word to the memory address plus the offset specified. In this case the memory address is specified as offset (`0`) + address (`%0`). The offset part of the code determines how many bytes offset from the address to write to. The address part is an address variable passed into the assembler portion using the gcc inline assembly syntax. In the code extract above, the last line before the closing bracket contains the following:

```
: "=m"(*pm)
```

This is a gcc assembler directive to pass in a write-to memory variable. This is referenced within the assembler as "`%0`". The "`%`" sign indicates that it is one of the variables declared in the list below the code, and "`0`" means that it is the first one declared.  If there were more declared, we would access them numerically in the order they were declared starting at `0`.  Refer to Appendix A for further details.

There is no specific store pair or triple instruction, so "`sv.s`" must be called nine times with the offset incremented correctly to store all the matrix elements in the correct places. However, when you use four element vectors and matrices, you may use the `sv.q` to write one quadword at a time to memory.

```
sceVfpuMatrix3RotY( scePspFMatrix3 *pm0,
                    const scePspFMatrix3 *pm1,
              float ry
);
```

The above function creates a rotation matrix, *ry* radians about the Y axis and stores it in matrix *pm0*. If the matrix *pm1* is not NULL, it will multiply it by the created matrix before storing. The VFPU code for this function is reproduced below.

```
__asm__ (
".set           push\n"
".set           noreorder\n"
"mfc1           $8,   %2\n"
"mtv            $8,   s100\n"
"vrot.t         c000, s100, [c, 0,-s]\n"
"vidt.q         c010\n"
"vrot.t         c020, s100, [s, 0, c]\n"
"beql           %1,   $0,  0f\n"
"vmmov.t        e200, e000\n"
"lvr.q          c100,  0(%1)\n"
"lvl.q          c100, 12(%1)\n"
"lvr.q          c110, 12(%1)\n"
"lvl.q          c110, 24(%1)\n"
"lvr.q          c120, 24(%1)\n"
"lvl.q          c120, 36(%1)\n"
"vmmul.t        e200, e000, e100\n"
"0:\n"
"sv.s           s200,  0 + %0\n"
"sv.s           s201,  4 + %0\n"
"sv.s           s202,  8 + %0\n"
"sv.s           s210, 12 + %0\n"
"sv.s           s211, 16 + %0\n"
"sv.s           s212, 20 + %0\n"
"sv.s           s220, 24 + %0\n"
"sv.s           s221, 28 + %0\n"
"sv.s           s222, 32 + %0\n"
".set           pop\n"
: "=m"(*pm0)
: "r"(pm1), "f"(ry * (2.0f/3.14159265358979323846f))
: "$8"
);
```

Floating point variables passed into a VFPU assembly function are at first stored as general purpose registers in the FPU. They must be moved into the VFPU matrix register file for the VFPU to use them. The first two instructions in this function achieve this:

```
"mfc1                 $8,   %2\n"
"mtv                  $8,   s100\n"
```

The ALLEGREX instruction "`mfc1`" means "Move from co-processor 1". It moves the variable from the FPU general purpose register into a CPU general purpose register. The instruction "`mtv`" - "Move word to VFPU" then moves the variable from the CPU register into the area in the VPFU register file for further operations.

The next three lines deal with creating the rotation matrix. A column rotation matrix about the Y axis can be defined as follows:

**Figure 5-4**

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These three VFPU instructions fill in the matrix elements with the relevant variables:

```
"vrot.t          c000, s100, [c, 0,-s]      \n"
"vidt.q          c010                       \n"
"vrot.t          c020, s100, [s, 0, c]      \n"
```

The "`vrot.t`" instruction is perfect for this task. It takes a scalar, and calculates the cosine and sine of the variable, then places the results within the triple word area in the matrix as defined. It can also be instructed to operate with positive or negative variables.

One thing to note here is the internal angle format of the VFPU. In the assembly parameter list for this function, the *`ry`* variable has been modified:

```
"f"(ry * (2.0f/3.14159265358979323846f))
```

This is a unit conversion procedure. The internal angle format of the VFPU is not in either radians or degrees, but in quadrants. A quadrant is one quarter of a circle, and a full rotation about a circle is equal to four quadrants. The conversion here is the method to convert an angle in radians into quadrants. The format of the "`vrot.t`" instruction is:

```
vrot.t          vDst, vSrc, Writemask
```

The *`vDst`* and *`vSrc`* are the destination and sources to the operation respectively. The *`Writemask`* is a mnemonic that will convert to a 5-bit number that the VFPU uses to determine in which pattern to write the variables into the *`vDst`* register. This mnemonic will accept a list of variations with some limitations. The "`c`" and "`s`" elements represent whether the cosine or the sine should be used, and the "-" symbol is valid for negative numbers. The order that these are set within the brackets determines the write positions within the destination triple. "`0`" will set the relevant element to zero. These are the only valid characters that can be used.

The "`vidt.q`" instruction will set a vector specified as the corresponding part of the identity matrix. In this case the second column, that corresponds with the rotation matrix requirements.

```
"beql           %1,   $0,  0f\n"
```

This ALLEGREX instruction means "Branch on equal likely". In this case it checks the second input parameter against the register "`$0`" which is always zero. The input parameter is the matrix pointer *`pm1`*. If the matrix pointer is NULL, the program counter is set to the address specified. In this case it is the subroutine specified by "`0:\n`" and referenced in the code as "`0f`". This instruction executes the instruction in the delay slot (the instruction following the branch) only if the branch test succeeds.

```
"vmmov.t        e200, e000\n"
```

This instruction copies a triple word vector from "`e000`" to "`e200`". Looking ahead a little, it is possible to see that "`e200`" is the matrix register section that all the values are written back to main memory from, though in scalar form. So, if the matrix pointer is NULL, the rotation matrix is copied to the new location, then execution branches to the section that writes the data back to main memory.  You may have noticed at this point that by passing in NULL to this function you do not need to call `sceVfpuMatrix3Unit( ... )` as a prerequisite. However, this has been left in as an example. If the branch does not execute, the matrix data is not copied. Instead there are a series of instructions executed, which are covered next:

```
"lvr.q          c100,  0(%1)\n"
"lvl.q          c100, 12(%1)\n"
```

Generally when a variable is loaded from main memory to the VFPU, it has to be aligned to its equivalent size. So, a word address must be aligned to a word boundary, and a quadword address aligned to a quadword boundary. If the memory addresses fail this prerequisite, the CPU generates an exception and execution will terminate. The "`lvr.q`" and "`lvl.q`" instructions help to avoid this problem. They work best when they are used in sequence, as demonstrated in the code. They store the quadwords from the address specified into the VFPU matrix register file, but the order and storage justification is important, as this is what differentiates them from standard load-word / load-vector instructions.

**Figure 5-5: *lvr.q* and *lvl.q* Instructions**

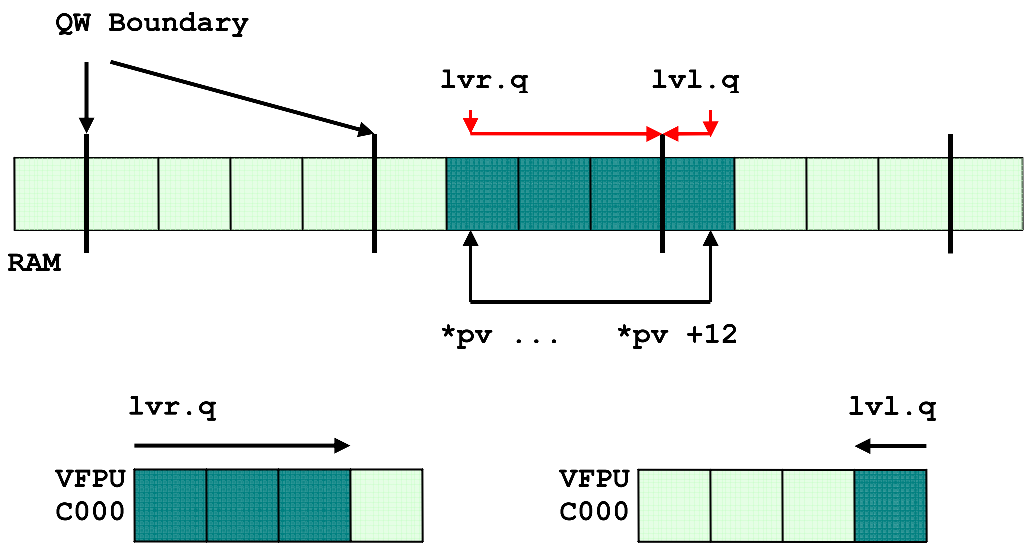


Figure 5-5 illustrates how the "`lvr.q`" and "`lvl.q`" instructions operate on memory and the VFPU matrix registers in this example. The upper part of the diagram shows a section of main RAM, the quadword pointed to by "`pv`" in a darker shade. As shown, the vector straddles a quadword boundary. The "`lvr.q`" instruction will effectively read right from "`pv`" until the end of the quadword boundary. It will store the words from left to right in the matrix register file. The "`lvl.q`" instruction reads from "`pv+12`" to the left until the quadword boundary and stores the words from right to left in the matrix register file. In this way, using both functions will load all the quadword elements into the matrix register file even if they are not aligned to a quadword boundary.

```
"vmmul.t        e200, e000, e100\n"
```

With the matrix loaded into "`e100`" the "`vmmul.t`" instruction is executed. This instruction is the 3x3 matrix multiply instruction which, in this case, stores the result in "`e200`". It is then written back to the destination address.

```
sceVfpuMatrix3Apply(   scePspFVector3 *pv0,
                       scePspFMatrix3 *pm0,
                       scePspFVector3 *pv1
);
```

The final function we are dealing with is shown above. It will transform a 3x3 matrix by a 3 element vector. This is the function that is actually responsible for rotating the vector about the Y axis:

```
__asm__ (
          ".set             push\n"
          ".set             noreorder\n"
          "lvr.q            c100,  0 + %1\n"
          "lvl.q            c100, 12 + %1\n"
          "lvr.q            c110, 12 + %1\n"
          "lvl.q            c110, 24 + %1\n"
          "lvr.q            c120, 24 + %1\n"
          "lvl.q            c120, 36 + %1\n"
          "lvr.q            c200,  0 + %2\n"
          "lvl.q            c200, 12 + %2\n"
          "vtfm3.t          c000, e100, c200\n"
          "sv.s             s000,  0 + %0\n"
          "sv.s             s001,  4 + %0\n"
          "sv.s             s002,  8 + %0\n"
          ".set             pop\n"
          : "=m"(*pv0)
          : "m"(*pm0), "m"(*pv1)
);
```

Most of the instructions in this example have been covered. As in the previous example, the matrix and vector are loaded with the "`lvr.q`" and "`lvl.q`" instructions. The new addition is:

```
"vtfm3.t         c000, e100, c200\n"
```

This single instruction will do the matrix multiply. The resulting vector is stored in column format, and then written back to memory with "`sv.s`".

## Summary

This chapter has provided an introduction to the VFPU coprocessor, and looked at the functions in libvfpu, a VFPU maths library provided by SCE. This chapter has developed the previous sample in Chapter 4 to rotate the light with the VFPU, and the explanations of the code have studied functions instruction by instruction.

The VFPU is a very powerful part of the PSP system, and using it is essential if you want to obtain maximum performance from the hardware. As well as the maths functions provided, the VFPU is well suited to many other applications, such as particle systems and physics simulation. Developers are already taking advantage of it, and will continue to find novel uses for it as time goes on.

## Conclusion

This concludes the document, which should have given the reader some insight into programming for the PSP. It explained some basic two-dimensional and three-dimensional graphics, and controller input, and it briefly described how to start using the VFPU within an application.

The PSP is, at the time of going to press, the most powerful and interesting portable games console available, and has many more features besides those outlined in this guide. Hopefully, you will see how easy many of these great features are to implement within their applications, and will be inspired to investigate further into the system that marks a new era in handheld entertainment.

# Appendix A:
# gcc Inline Assembly Syntax

This page intentionally left blank.

Inline assembly is a very useful feature that allows assembly code to be inserted within a C or C++ source file. This is handy for writing optimized VFPU code within a function for example. Gcc provides this option natively but you need to follow some rules to use it. This appendix will provide an overview of the gcc inline assembly syntax for general use. A typical VFPU inline assembly function is given below:

```
ScePspFVector4 *sceVfpuVector4Scale( ScePspFVector4 *pv0, const ScePspFVector4
*pv1, float t)
{
__asm__ (
          ".set        push\n"
          ".set        noreorder\n"
          "mfc1        $8,   %2\n"
          "mtv         $8,   s010\n"
          "lv.q        c000, %1\n"
          "vscl.q      c000, c000, s010\n"
          "sv.q        c000, %0\n"
          ".set        pop\n"
          : "=m"(*pv0)
          : "m"(*pv1), "f"(t)
          : "$8"
   );
   return (pv0);
}
```

This function scales a four element vector by a scalar. As this section will be looking only at the inline assembly syntax, the actual VFPU code has been greyed out, and will not be discussed in the same way as the previous chapter.

```
__asm__ ( ... );
```

All inline assembly code must be placed within the above statement. This is a compiler directive to tell gcc that this is assembly code so that it doesn’t try to compile it as normal source code as this code is reserved for the assembler.

A translation of the above code into the relevant naming conventions that will be described in this appendix follows:

```
__asm__ (

“set            assembler option                        \n”
“set            assembler option                        \n”
“opcode         register,    variable                   \n”
“opcode         register,    register                   \n”
“opcode         register,    variable                   \n”
“opcode         register,    register, register  \n”
“opcode         register,    variable                   \n”
“set            assembler option                        \n”
: “constraint”   (output variable)
: “constraint”   (input variable), “constraint” (input variable)
: “clobber”
);
```

## Setting Assembler Options

The assembly code is enclosed by three "`.set ...\n`" statements. These set assembler options are quite important to the code. These three are probably the mostly common used:

"`.set push\n`" saves the current assembler options to a stack, so that the current state can be restored if need be.

"`.set noreorder\n`" is important as it tells the assembler not to reorder the instructions, which it may try to do if it thinks the code needs optimizing. Normally, we want to have the utmost control of this, as hand-optimized code is always going to be better than the assemblers attempted solution.

"`.set pop\n`" restores the previous assembler options as they were before "`.set noreorder\n`"

## The Assembler Code

The assembly code itself is a series of instructions that perform operations upon data read from, and written to, input and output variables and registers. The input and output variables are declared in the assembler with "`%`" signs followed by numbers. These variables are defined in the first two sections, separated by a colon after the main code block.

## Output / Input Variables

These are variables that the assembler code is to write to and read from. The main first difference between variables in inline assembler and those within C, or C++, is the order. In C/C++, the variable is declared above its use. To allow their use in inline assembly, they are declared below the assembler code, in the input/output variable lists, with the following syntax:

```
: “constraint” ( output variable 0 ) // referenced with %0
: “constraint” ( input variable 1 ) // referenced with %1, etc
```

Output and input variables are declared in the assembler code in the order shown above, increasing from `%0`. The output variable in the example code is a vector pointer `pv0` passed in as a parameter to the function. Input variables follow the same syntactical rules, and are accessed in the same way in the code.

## Constraints

A constraint is additional information to the assembler to define how the variable should be used. Some common examples are listed below:

```
“r”, “f”
```

This states that the variable must be placed in an integer register for "`r`" and floating point register for "`f`".

```
“m”
```

This states that the variable is an address.

In addition, there are modifiers that can be added to the constraints, some common ones are:

```
“=”
```

This stipulates that the variable is to be write-only.

```
“+”
```

This is a read / write variable.

## Clobbers

The final colon separated part of the block is the clobber list. This is the list of registers that the assembler code uses. If the code uses a CPU register, it *must* be included here: there is no need to include VFPU registers.

Further reading:

[12], [13], [14]

# Appendix B: Emulator and Hardware Tool Library Differences

This page intentionally left blank.

This appendix covers the different functions used in the samples that differentiate the code between the Emulator and the hardware tool.

## Emulator Specifics

Emulator #includes & functions

```
#include <libemu.h>
```

This file must be included to call emulator functions.

In the `Init()` function, before libGu is initialized, the emulator must be initialized, this is done with a call to:

```
sceEmuInit()
```

Similarly when the program shuts down there is a call to:

```
sceEmuTerm()
```

this will shut down the emulator library.

The vertical sync on the emulator is:

```
sceEmuVSync(SCEEMU_SYNC_WAIT)
```

These are the only emulator specific functions that need to be invoked apart from the controller differences that have already been covered in Chapter 4.

## Hardware Tool Specifics

Firstly, to allow code that was written for Emulator to work with the hardware tool, remove all the previously defined calls to any Emulator functions. After that, there are some important additions to the code that are explained in this section.

The following file needs to be included:

```
#include <modulexport.h>
```

The following macro is needed on the hardware tool to generate PSP applications, namely, “modules”:

```
SCE_MODULE_INFO( hellotriangle, 0, 1, 1 );
```

The above macro generates information for the modules. Many modules can be loaded at the same time. They are managed by the PSP kernel. The above macro is needed so that the kernel can identify the module.

The name of the module must be the same as the name of the executable.

```
#include <displaysvc.h>
```

This is needed for display function calls, such as:

```
sceDisplayWaitVblankStart()
```

This function is the hardware vertical sync function, that takes the place of `sceEmuVSync(SCEEMU_SYNC_WAIT)` as used in emulator code.

Another hardware specific display function that must be invoked, in the `InitGFX()` function, is:

```
sceGuDisplay(SCEGU_DISPLAY_ON);
```

This explicitly activates the PSP hardware screen. If this function is not used, the screen will not display anything.

Although the samples accompanying this tutorial are not dynamically allocating memory, this feature will be needed by almost all applications above a certain size. It is therefore important to be aware of the related memory management issues, described below.

## Memory Management

The PSP hardware has 32MB of DDR RAM as main memory. However, the kernel reserves 8Mb of this. The executable, minus debugging information, must reside in the remainder of main memory. So, if an executable is 3MB, for example, there will be 21MB of RAM left for the application. To allocate memory from within this remaining 21MB block using `malloc()`, the code must include the following line:

```
int sce_newlib_heap_kb_size = <size of heap in kilobytes>;
```

This is the amount of memory that is available to `malloc()` within the application. You may have your own memory management scheme, which will then take this block to manage independently of `malloc()`,however this is the general way to allocate the memory in the first place.

## References

1. *ALLEGREX Manual – SCE*
   https://psp.scedev.net/projects/hwmanuals
2. *LibGu Reference Manual – SCE*
   https://psp.scedev.net/projects/library
3. *Graphics Engine Users Manual – SCE*
   https://psp.scedev.net/projects/hwmanuals
4. *Computer graphics, C version – Second Edition*
   Donald Hearn and N. Pauline Baker, Prentice Hall
5. *Introduction to RISC Assembly language Programming*
   John Waldron, Addison Wesley
6. *See Mips run*
   Dominic Sweetman, Elsevier, Morgan Kaufmann
7. *Vfpu Users Manual – SCE*
   https://psp.scedev.net/projects/hwmanuals
8. *Kernel Overview – SCE*
   https://psp.scedev.net/projects/library
9. *PSP Thread Manager Reference – SCE*
   https://psp.scedev.net/projects/library
10. *Vfpu Instruction Reference Manual – SCE*
    https://psp.scedev.net/projects/hwmanuals
11. *Dylan Cuthberts Asm Webpage*
    http://www.geocities.com/dylan_cuthbert/asmrules.html
12. *Gcc Extended Asm Page*
    http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm
13. *Inline Assembler Constraints*
    http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_16.html#SEC175
14. *OpenGL Programming Guide*
    Fourth Edition, Shreiner, Woo et al Addison Wesley