PSP Homebrew GameDev with Eclipse

By Paulo Lopes
www.jetcube.com

Getting Started

During this small book you will learn to setup the development environment for your PlayStation Portable using either Microsoft Windows or Linux as your host operating System. Although the book will only focus on these two operating systems you can easily re use the same knowledge to your favorite operating system as long as you manage to cross compile the PSPSDK. At the moment of writing of this book the SDK can be cross compiled to Microsoft Windows (native and emulated Cygwin), Linux, BSD, OpenSolaris and MacOS.

You will learn how to setup a working IDE with full debugging support and then create a minimalist game engine with a blender exporter for your 3D assets. This book will not teach the reader to program and assumes that the reader is already familiar with the C programming language. This does not mean that C is the only supported language in the SDK, you can use C, C++, Objective-C or Objective-C++ for your native development or even Lua as a scripting language (again not covered in this book).

Are you ready? Let's get started!

Required Software

Getting started is easy and only requires one package, however this book will help you to be productive, therefore besides MinPSPW (the minimal required software to compile your game to the PSP) You will install a C/C++ IDE (Eclipse) and a open source 3D modeling tool (Blender3D) to create a full pipeline of game development.

You will get more information on what are these components and are free to replace them with others you like more after you understand their role in the full pipeline.

MinPSPW

Before starting developing for your PSP you need machine code tools. The minimal toolset you will need is a compiler and linker for the PSP CPU. The PSP CPU is known as the MIPS Allegrex. Basically is a modified MIPS 3000 + FPU (Floating Point Unit) + VFPU (Vector FPU) all in one. Although originally prepared to run at reduced CPU clock speed (200MHz) the usage of all these CPUs make the PSP a powerful machine and still up to date (4 years after it's initial release).

The compiler and linker for this CPU is not publicly available, actually Sony has never released it as an open source tool, however a huge base of fans understood the full power of the machine and started to dig into the hardware and binaries that you

can run on the device. The findings allowed us to create a simple tool-chain based on the popular GNU GCC compiler.

This modified compiler is available to all under a GPL/BSD license making it free for everyone. Under Microsoft Windows there were two options, either use a posix emulator (Cygwin) or a native build (devKitPro) which is not updated very often.

In the beginning of 2008 a new project started to port the SDK and keep it up to date without any kind of emulation for windows (MinPSPW). Users could just download a simple installer and once the installation was finished, games could be compiled from the standard dos console window, the same way the Linux/BSD variant can compile from the shell.

Eclipse

PSP development forums are full of posts regarding people trying to get an unified development environment. There are reports about using Visual Studio, Code::Blocks, vi, Eclipse, etc... but it was always tricky to get it working.

Eclipse is a Java-based, extensible open source development platform. By itself, it is simply a framework and a set of services for building applications from plug-in components. Fortunately, Eclipse comes with a standard set of plug-ins, including the well-known Java Development Tools

(JDT).

Eclipse is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools, and runtimes for building, deploying, and managing software across the life cycle. The Eclipse Foundation is a not for-profit member-supported corporation that hosts the Eclipse projects and helps cultivate an open source community and an ecosystem of complementary products and services.

Blender3D

Blender was first conceived in December 1993 and born as a usable product in August 1994 as an integrated application that enables the creation of a broad range of 2D and 3D content. Blender provides a broad spectrum of modeling, texturing, lighting, animation and video post-processing functionality in one package. Through it's open architecture, Blender provides cross-platform interoperability, extensibility, an incredibly small footprint, and a tightly integrated work flow. Blender is one of the most popular Open Source 3D graphics application in the world.

Aimed world-wide at media professionals and artists, Blender can be used to create 3D visualizations or animations, stills as well as broadcast and cinema quality videos, while the incorporation of a real-time 3D engine allows for the creation of 3D interactive content for stand-alone

playback.

Prepare the Environment

During this step You will learn how to install your environment and set it up in order to be productive. This step involves download of open source software from the Internet (during all this book, only open source software will be used, regardless of the license as long as it is OSI approved). Again the only required component is the MinPSPW, however for the course of this book, you will need some extra components: Java Runtime, Eclipse IDE, Python 2.6 and Blender3D.

MinPSPW

The MinPSPW tool-chain is available at: *http://www.sourceforge.net/projects/minpspw*. At the moment of the writing of this book the current version is 0.9.6. Once you download it to your PC, start the installation and follow the wizard:

Illustration 1: MinPSPW Windows Installer

The installer will ask you to agree with the license, which basically is a collection of open source licenses for all the components inside the SDK. BSD license for the SDK itself, GPL for the MinPSPW and some of the included libs and LGPL for some of the included libs. There are no closed source components included.

Once you are done with the license you can choose which components to install, If you are not sure, just select everything unless you don't have around 150Mb of disk space.

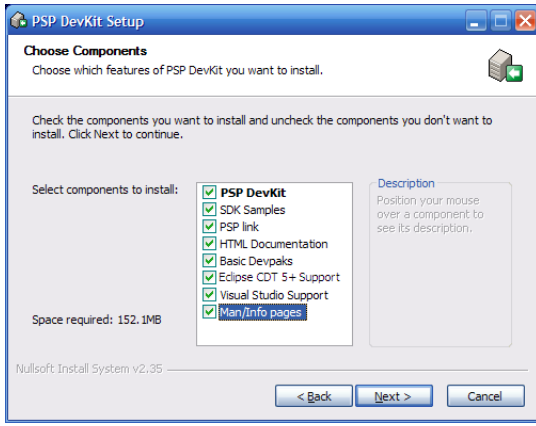

Illustration 2: MinPSPW Windows Installer Options

At this moment you have a working compiler that allows you to unleash the power of your PSP. To make sure that everything works lets make a simple test. Let's check the version of a couple of components that were just installed. Open a DOS console prompt and execute:

make –version

Followed by:

psp-gcc --version

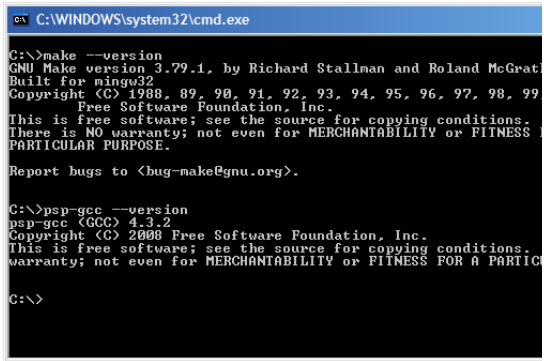

Illustration 3: Command Line check

If you don't know what those commands are, I'll explain: The first one is the build tool, that will automate your build process of compiling and package your code into a PBP file that can run on your PSP. The second is the PSP compiler and linker.

Eclipse

Eclipse IDE can be downloaded from the project web site at: *http://www.eclipse.org*. This book covers Eclipse >=3.5. In order to get eclipse running you need to have an updated Java Runtime. So before starting looking for the right Eclipse for you download the Java Runtime from *http://www.java.com/getjava*. The Java page is quite good explaining on how to download and install the runtime. At this moment you might be asking the question: "*can I code in java for the PSP?"* Well the answer is yes and no; yes you can if you use something like pspkvm; and no because the native

compilers do not support gcj (yet).

I assume that you already have the java runtime installed, so the next step is to install Eclipse. As said before get it from *http://www.eclipse.org* and select the "*Eclipse for C/C++ Developers*". You can choose any of the other options as you make sure that later you install the CDT (C Development Tools) plug-in. For now just download the Eclipse for C/C++ Developers.

Once you download the zip file, unzip it to a simple directory in your PC. I assume you unzip to *C:*\. In order to make it easier make a shortcut to your desktop to the executable file on *C:**eclipse**eclipse.exe*.

Starting Eclipse for the first time

Use the link you just created to start Eclipse. The first time you open it, you see the workspace location page, which informs the IDE where all your work is located in your computer.

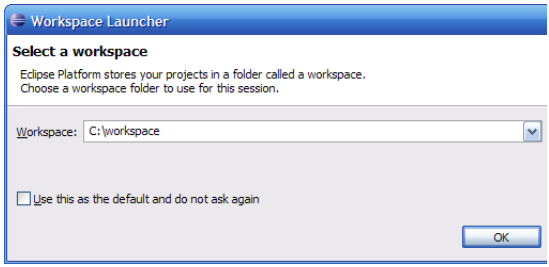

Illustration 4: Eclipse workspace launcher

Once you have decided where to store your work you are shown the welcome page. This page exists inside the workbench. As an Eclipse user, you'll be given a few options of going to an overview page, which I recommend. See what's new, explore some samples, or go through some tutorials.

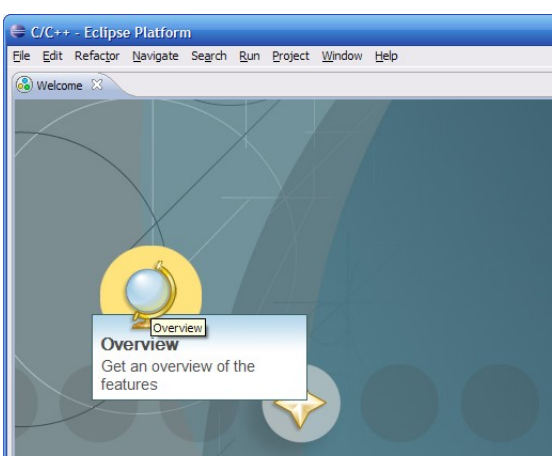

Illustration 5: Eclipse welcome screen

The Eclipse workbench consists of several panels known as views, such as the navigator or outline views. A collection of these views is called a

perspective. One of the most common perspectives is the Resource perspective, which is a basic set of views for managing projects and viewing and editing files in a project.

I recommend most novice users start with the Overview page featured in Figure and learn about Eclipse. The workbench basics section contains a lot of good starter information about the various pieces of Eclipse and how they interact.

Blender3D

Blender3D relies heavily on Python scripting language. In order to get the full potential from this package you should install Python (version 2.6 for binary compatibility) into your PC. Install Python is easy as it was to install the Java Runtime. Python can be downloaded from the project website *http://www.python.org*, go to the download page and download the latest 2.6.x installer. Do not install the latest version since Blender3D might have some trouble using it. Get the release MSI and install it. Follow the wizard and again try to use the default installation path *C:\Python26*.

Once you PC has Python installed download and install Blender3D. By the time of writing this book, Blender 2.48a is the current release, download it from: *http://www.blender.org* and install it. Again I advice you to install to the default path.

Starting Blender3D

Blender3D user interface is not easy for the beginner. Just explore the menus and if you really interested there are some nice tutorials and books that can help you getting into shape with Blender.

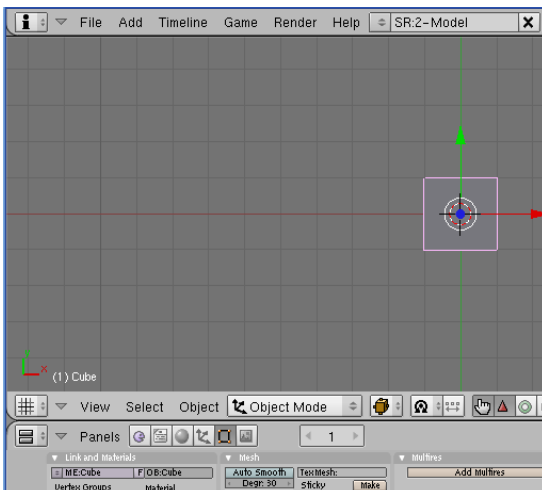

Illustration 6: Blender3D main window

Setup the Environment

Although you have installed all the required tools there are some extra configurations you need to do to your Eclipse in order to have it fully inter-operational with the PSP SDK tools. There are 3 things you need to configure, the first is the C compiler discovery features, the second some external tools that will help you with debugging and the third is a USB driver to allow remote debugging from your PSP within your Eclipse IDE.

Makefile Options

Open your Eclipse and go to the "*Window > Preferences*" menu. Once you are there navigate to the "*C/C++ > New CDT Project Wizard > Makefile Project"* on the navigation tree. This panel will allow you to configure a compiler which is not the host system GCC. Since we are using a cross compiler (not a native compiler) we need to change some parameters to get the full power of Eclipse.

On the first tab, "*Binary Parsers*", enable the "*ELF*" parser. PSP runs binary files in ELF format. Allowing this configuration to parse them will show you once you compile which functions are available in your object files. This might seem not important but can help to identify bugs later on. As a rule of thumb enable it even if you don't really think you need it since there are no side effects either in code size or performance of your system.

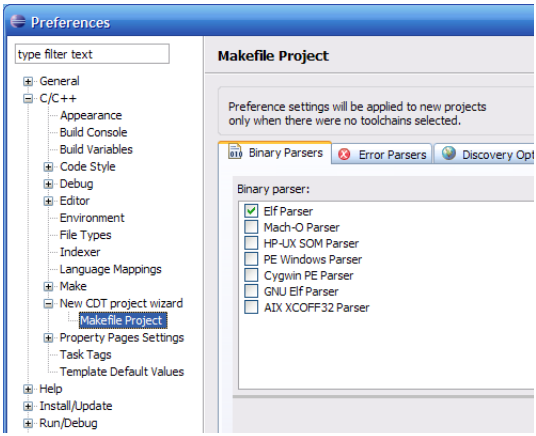

Illustration 7: Eclipse Binary Parser Configuration

You should also configure the Discovery Options; go to the "*Discovery Options"* panel. This is a very important configuration because not doing it, although your system would still work, all the productivity features such as code complete and API navigation will not be available outside the source code of your project. Again remember that using the latest Eclipse this step is not required anymore. Go to the third tab Discovery Options, and change the "*Compiler Invocation Command*" to "*psp-gcc"*. Once you do this Eclipse will pick the correct compiler from your system and enable code complete and code navigation features, assisting you to produce faster code.

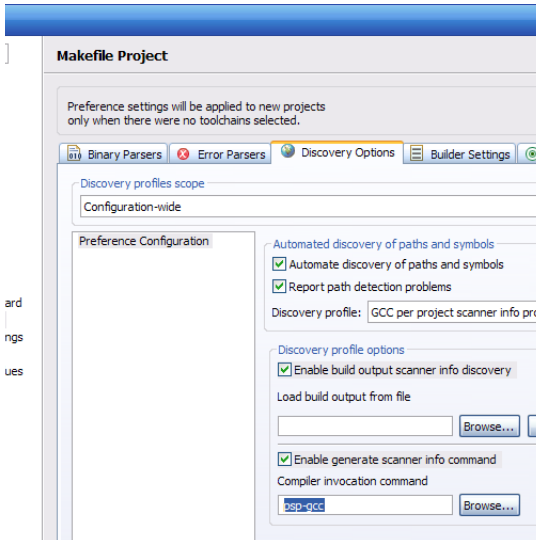

Illustration 8: Eclipse Discovery Options Configuration

External Tools

The MinPSPW SDK is an open source SDK for the PlaySation Portable, it does not contain hardware debugging facilities as the official Sony SDK. To debug a Homebrew application you would usually code your game, compile, link, plug the PSP usb cable, transfer the PBP file to the memory stick, unplug the cable, start the game and on error you would either be lost or you would have some text file in your memory stick with your *printf* debug statements.

This approach can become impractical for real projects. Usually most of the development environments provide a source level debugger, the

SDK is not different but you need some bridge between your development machine and your test environment, that is when *PSPLink* comes into place. We will cover PSPLink later during the debug chapter, for now just be aware that the PSPLink is composed of 4 parts:

- USBHostFS_PC - USB communication between the PSP and your PC.
- pspsh - A simple shell that controls the USBHostFS.
- PSPLink PBP - A PSP application that will be the bridge between your application and all the debugging facilities.
- USB Driver - A special USB block device driver that does the low level communication between the PSP and your PC.

The first 2 features that need to be configured into Eclipse. In the main Eclipse window, locate the "*External Tools button*" This is a simple green run button with a red toolbox under. Note that there is a drop-down option in the button, activate it and select "*External Tools Configurations...*".

Create a new tool under "*Program"* called "*usbhostfs*". On the location find the folder where you installed the PSPSDK and under the *bin* folder you will find an executable named: "*usbhostfs_pc*". Apply your changes. On the working directory use the variable "*project_loc*". Apply your changes and close.

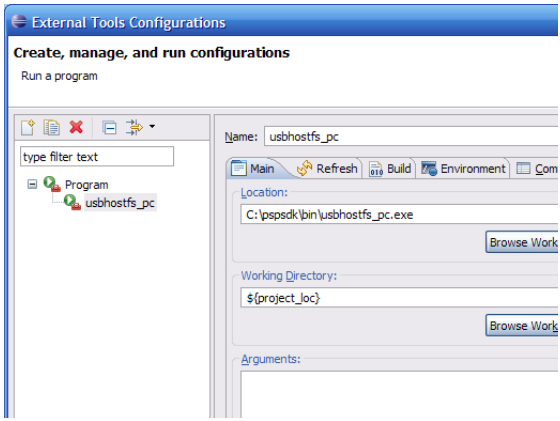

Illustration 9: Eclipse External Tools (USBHostFS_PC) Configuration

Still in the same configuration window, create a new tool under "*Program*" called "*pspsh*". On the location find the folder where you installed the PSPSDK and under the *bin* folder you will find an executable named: "*pspsh*".

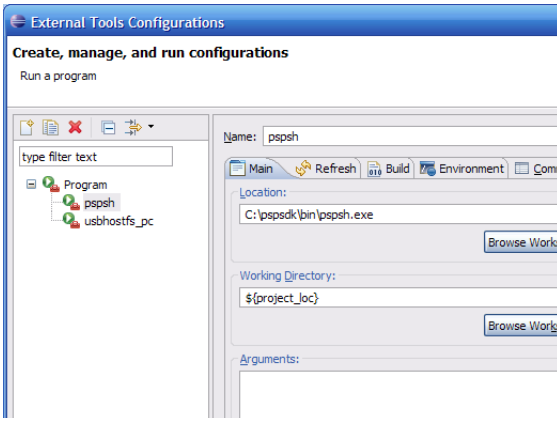

Illustration 10: Eclipse External Tools (PSPsh) Configuration

Drivers & PSPLink

Your environment is almost ready, locate the MinPSPW installation folder and inside you will find another folder named: "*psplink*". Inside the "*psp*" sub folder you see a couple of sub folders. Pick the right folder name according to your firmware version, for most people the right one is the *oe*. Copy the folder to your *GAME* folder in the PSP memory stick. Optionally rename it into something more readable since your might forget what it is in your memory stick later on.

PSPLink communicates through a special USB driver. This allows you to remotely debug your application. It needs Administration rights in your machine since it is a Windows OS driver. To install follow the next steps:

- Connect your PSP to the USB cable and make

sure that the USB link is disabled, in other words you should not be able to see the memory stick contents from your PC.

- Start the PSPLink application on the PSP
- A New Hardware Wizard will popup asking for the correct driver. The driver is in the MinPSPW "*bin*" folder. Locate it and point to the "*driver*" folder.
- If you have a 64 bit OS use the "*driver_x64*" folder.

At this moment your USB driver should be installed and ready for use. To try out and see if everything when right, go back to your eclipse and start the USBHostFS external application, once you have done it, the console window in the bottom should show something like this:

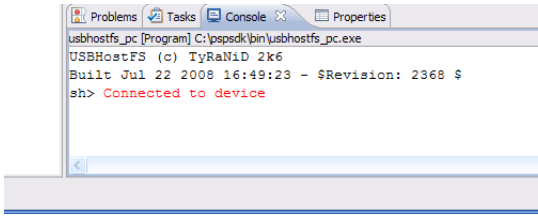

Illustration 11: Eclipse connected to PSPLink

C programming

In this chapter you will learn how to code your first game. You will learn how to setup your project under eclipse and use the features that it gives to you such as code complete. You will also learn some of the internal differences of coding for the PSP and coding for a normal PC.

Not an Hello World

For your first example you will not use the *Hello World* example. You will be more advanced and create a spinning cube. I will not cover the details of programming the code is self explanatory and there are lots of good tutorials and books on C development available.

For this first application you will not even need to code, we will use one of the samples that ship with the MinPSPW SDK.

Create a Project

Open Eclipse and create a new *C project*. Select *File > New > Project* and finally select *C project*. From the project type tree pick *Makefile Project > Empty Project*. This will be your standard project type for most of the PSP development with Eclipse. Selected this project for *--Other Toolchain--* and then enter "*Spinning Cube*" in the project name and point the location to "*C:\pspsdk\psp\sdk*

samples\gu\cube" after checking out the default location check box. Finally click Finish.

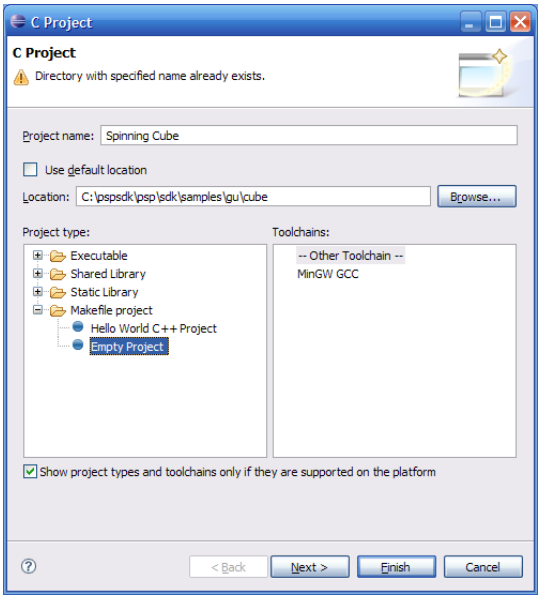

Illustration 12: New Eclipse C Makefile project

Next, we'll take a look at the C/C++ perspective (if you aren't already there). Depending on how you like to manage your screen, you can change the perspective in the current window by selecting *Window > Open Perspective > C/C++* or you can open a new window by selecting *Window > New Window* and selecting the new perspective.

The C/C++ perspective, as you might expect, has a set of views that are better suited for C/C++ development. One of these includes, as the top-left view, a hierarchy containing various C/C++

namespaces, classes, includes, libs and miscellaneous files. This view is the called the Package Explorer. Also notice that the main menu has expanded to include two new menu items: Source and Refactor.

If you have not noticed yet, you just compiled your first PSP application. This is because Eclipse build your project automatically. Now that you are familiar with the workbench you can see in the project explorer that a file named EBOOT.PBP is present. Before running and putting this file in your PSP lets look a bit closer to the workbench. Double click on the cube.c file. Eclipse will highlight the C language keywords and even bold out your function names.

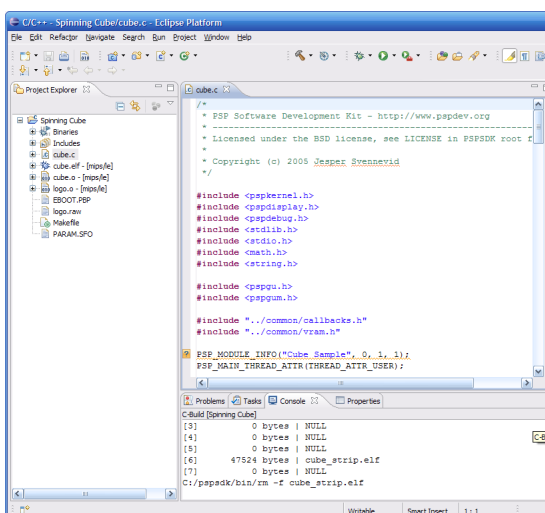

Illustration 13: Eclipse C perspective

Eclipse will allow you to navigate through

the code by "*Ctrl + Click"* a function name or include file. This will open the definition of the method or open the file. While typing "*Ctrl + Space*" will start the code complete feature like visual studio's Intellisense. Hovering over a function will trigger Eclipse to parse any documentation in the source code written in a Doxygen/Javadoc compatible way. These are just some of the features you can have with Eclipse, feel free to explore the menus and the Internet for more productive tips.

Debug your Game

Now that you are already know how to code for the PSP in a productive way, let's cover the debugging of your game. Sooner or later you will need to debug, that is a fact. Eclipse makes it easy just by using the visual IDE. Starting from the previous example, open the *cube.c* file.

This is the main source code for your game, lets say the spinning is to slow and you want it to be doubled. Assuming you did not know anything about the code one possible solution would be to add a breakpoint on the last instruction of the main loop. Locate the line containing *val++* and with the mouse double click on the gutter bar to add a breakpoint.

However this looks like enough it will not work. The default Makefiles from the MinPSPW are not compiling your sources with debug information and the current PSPLink for OE firmware only allows the debugging of PRX modules. To get it working (and I assume you also know a bit from Makefiles)

you should open the Makefile by double click it on the project resources and add "*-g*" to the *CFLAGS* variable. Also to build a PRX you should just the *BUILD_PRX=1* line to the Makefile. Now clean your project using the *Project > Clean* menu and you have a debuggable game.

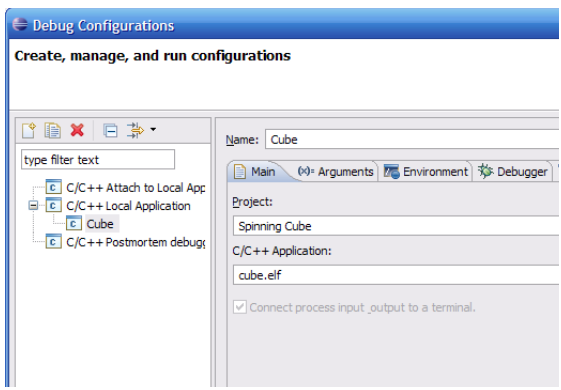

Illustration 14: Eclipse C Debug Config 1

The debugging process is a bit tricky and can be described as the following steps:

- Plug the PSP to the PC using the USB cable.
- Start PSPLink on the PSP
- Start the USBHostFS application by selecting the project and using the external tools button.
- Start the PSPSh external tool by selecting the project and using the external tools button.
- On the PSPSh type: *debug cube.prx* where cube.prx is your project PRX file.
- Start the Eclipse debugger (the little bug button).

If you are debugging for the first time you need to configure the debugger for the project. This is a project specific configuration that is the reason it was not included in the previous chapter. Select the drop down from the debug button and pick "*Debug Configurations...*" A new window pops up and you should select *C/C++ Local Application*. The project should be filled in be default with your working project and in the *C/C++ Application* your application ELF should be entered, for out example it should be cube.elf.

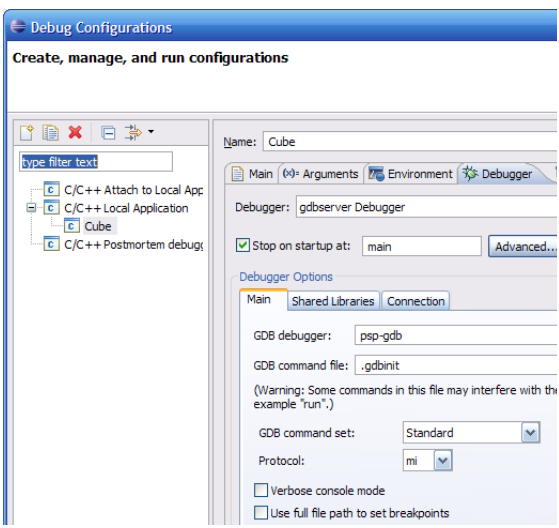

Illustration 15: Eclipse C Debug Config 2

This is the first step, the second step is to inform Eclipse that we are running a remote debugger, so switch to the debugger tab and the GDB debugger should be replaced from *gdb* to *psp-gdb*, once this is done on the connection sub tab, the

type should be *TCP* and the port *10001*.

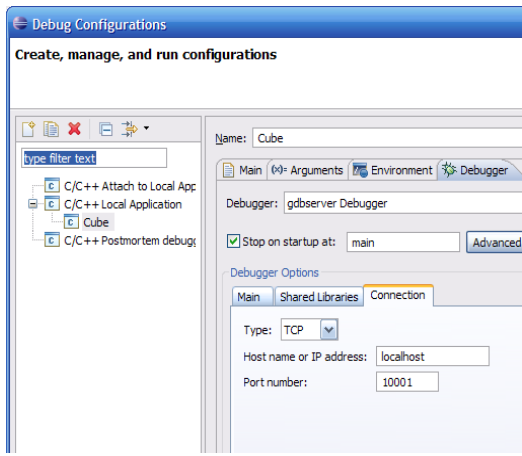

Illustration 16: Eclipse C Debug Config 3

Your debugger is now ready and should allow you to debug your code. Once the debugger starts you are asked to switch to the debug perspective, do it and inspect your code.

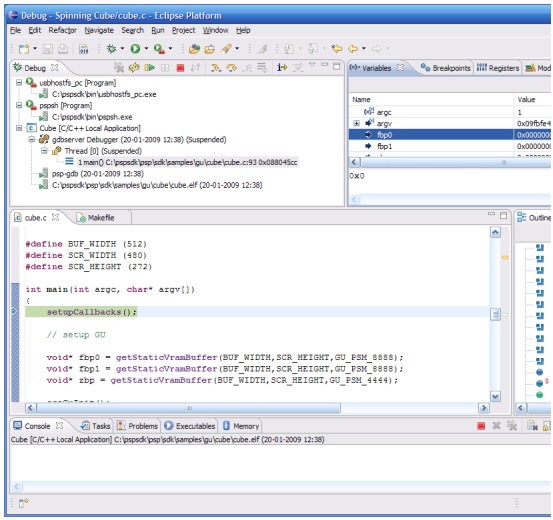

Illustration 17: Eclipse Debug Perspective

One very important fact about using the debugger is how to run it several times. To be able to run the debugger over and over, we need to understand first how does the *USBHostFS* and *PSPsh* works. The PC side of *USBHostFS* connects to the PSPLink on the PSP and should be running all the time. One best practice would be to once you decide to debug start the *USBHostFS* external application and leave it running (unless you need to test several projects and need to keep changing the project path).

To debug several times in a row you have to start only once the *PSPsh* and instruct the PSP unit that you are going to debug as explained before.

Now the important thing is to remember the step, before starting the Eclipe debugger, go to the

PSPsh console and execute:

debug prxfile.prx

Where *prxfile.prx* is the output of your project.

Profiling your Game

Sometimes, debug is not enough, your game is slow and apparently there are no bugs. For all those familiar with GNU environment there is no need to introduce gprof. Everybody else should get familiar with this tool by reading documentation and examples. To make a long story short, it is a tool for statistical code analysis and code instrumentation.

These two techniques, when combined, provide you with information of how many times a particular function was called (instrumentation) and how much time it spent executing (sampling).

The good news is that gprof is already a part of PSPSDK and gcc can automatically put required stubs into your source code, although it is not enough for it to work. You need to declare and call *gprof_cleanup()* in your code at the end of main() or whenever appropriate (for example ExitCallback). This will write profiling data to current directory on Memory Stick.

However to instruct GCC to instrument the code you need to change the *CFLAGS* variable in your *Makefile* to contain *-pg*. This will tell GCC that is should instrument all the code compiled in this

Makefile. Run the code.

You should now have gmon.out file on your Memory Stick. Copy it to your computer and use *psp-gprof* to analyze it. Take a look at the output file. To do it, use *psp-gprof elf-file gmon.out-file*. You can pass many options to gprof, all documented in it's manpage, but for now the most interesting will be -b or --brief which will produce a profile and a call graph for your program without the additional bloat coming from comments.

```
Flat profile:
Each sample counts as 0.001 seconds.
  % cumulative self self total
  time seconds seconds calls s/call s/call name
 49.78    2.00    2.00     2   1.00   1.00 wait1second
 49.78    4.00    2.00     1   2.00   2.00 wait2seconds
  0.42    4.02    0.02     1   0.02   4.02 main
  0.02    4.02    0.00     1   0.00   0.00 SetupCallbacks
  0.00    4.02    0.00    10   0.00   0.00 loops_10_times
  0.00    4.02    0.00     5   0.00   0.00 nested
  0.00    4.02    0.00     1   0.00   0.00 gprof_cleanup
```

This is called a at profile. It shows you time spent in each function and how many times each of these functions was called. This is great for taking an overall “look” at the code to spot most time-consuming parts of code. As you can see results precisely reflect the source code.

Using the profiler you can identify which functions take the most time in your code and the focus on optimizing those. Of course this is just a reference it does not mean that your code is not already optimal in those functions.

Understanding crash dumps

If you are testing your application using the *PSPLink* application it can happen that your program

might crash. It is a fact of life that no one codes perfectly at the first try (maybe Donald Knuth does). So lets say that your game was running good until it suddenly stops and PSPLink reports to you something like this on the console:

```
host1:/> Exception - Bus error (data)
Thread ID - 0x048E6D07
Th Name   - user_main
Module ID - 0x00CD497B
Mod Name  - "PSPWebCam"
EPC       - 0x08805518
Cause     - 0x1000001C
BadVAddr  - 0x0B089540
Status    - 0x20088613
zr:0x00000000 at:0xDEADBEEF v0:0x00000000 v1:0x000000FF
a0:0x00000000 a1:0x4400C754 a2:0x00000008 a3:0x4400BF54
t0:0x00000008 t1:0x00000080 t2:0x00000000 t3:0x00000001
t4:0x00000000 t5:0x4400BF54 t6:0x000000DB t7:0xDEADBEEF
s0:0x00000015 s1:0x0BBBFE34 s2:0x00000001 s3:0x0BBBFEE0
s4:0x00000015 s5:0x00000013 s6:0xDEADBEEF s7:0xDEADBEEF
t8:0xDEADBEEF t9:0xDEADBEEF k0:0x0BBBFF00 k1:0x00000000
gp:0x0882B2E0 sp:0x0BBBFDF0 fp:0x0BBBFDF0 ra:0x08804CC0
0x08805518: 0xA0400000 '..@.' - sb         $zr, 0($v0)
```

At a first look this is too cryptic to understand, it shows some pointers and some register content, but how can this help you out? Well in fact it can. Since you have access the EPC address you can find out where exactly in the code your game crashed with a Bus error.

In the old days (meaning, for people who are using firmware 1.5), finding the code location would be a simple task, however I am assuming people are using an custom firmware (that was the reason why we installed PSPLink-OE). It is not so simple but not a big issue, to do this you need to execute two commands. The first one will calculate the base address of your game since your are building PRXes. In case you are not familiar with the PRX term, it means that the code is relocatable when loaded by the OS, meaning that it's base address can be somewhere where there is some free

memory. To calculate this we use the PSPLink shell and execute:

calc $epc-$mod

Once we have this we can use another tool that comes with the SDK, it is called *psp-addr2line*. To use it just open a *Command Prompt* and execute the command:

psp-addr2line -fe cube.elf 0x08805518

You should replace cube.elf with your game elf file, and the last hex number is what you just received from PSPLink.

Minimalist Game Engine

In this chapter I will show how you can create a very simplistic game engine. It will not be suited for real life games but is just a follow up on how to use a modeling tool to create your 3D assets and import them into the rendering pipeline of the PSP.

From Blender to the PSP

The basic structure for 3D manipulation is a Vertex. Under the 3D Graphics Unit (GU) a Vertex is represented by a C struct of 192 bits. This struct contains information about its normals (u, v), the vertex color and the 3D location in space (x, y, z).

One can model it as the following piece of code:

```
struct Vertex
{
  float u, v;
  unsigned int color;
  float x,y,z;
};
```

Why do you need to care about this struct and why is there a small section dedicated to it is simply because it is a very important component for our 3D pipeline and I will need to use it from other external applications (namely Blender) when we will write a 3D exporter.

Blender Exporter

I am going to write a simple exporter to a binary format that we can pick up later and render on the PSP with minimal effort. For the sake of the demonstration, the code is simple and not designed for optimal performance. It is an academic exercise to demonstrate how easy is to use Blender and the PSP.

Blender3D is a powerful 3D modeling and animation package that is powerful but at the same time simple to script using Python as its internal language. All Blender objects are exported to Python giving us the full power of this tool to fit our needs.

Although Blender has its own code editor, I feel more comfortable using a generic text editor or Eclipse. Adding a simple exporter is as simple as creating a text file and copying it to a specific folder under your Windows user profile.

Register your exporter with Blender is as simple as writing the following line:

```
Blender.Window.FileSelector(do_export, "Export",
    sys.makename(ext='.jc'))
```

The interesting side of Blender API is that we just associated the *.jc* extension as a known exporter that will invoke the *do_export* function when we press the export JetDrone model from the menu. Exporting is also some trivial piece of code:

```
def do_export(filename):
  file = open(filename, 'wb')

  # Get the current scene and object
```

```
# Get the object color model/texture mode

# Count the number of vertex
# Count the number of faces

# write the header of the file with these counts

# iterate the object and write the vertex data

file.close()
```

This is just a simple exporter so in this piece of code we will only export meshes. After reading the code it all comes together, first we open a file to export, we get a reference to the current Blender scene in order to get the render data. From the render data we can capture the current frame. This is like this because Blender can also be used for animation requiring therefore us to iterate through all the frames.

Writing the Header

The Header of file is one of the most important feature of our exporter. The reason behind this is that the header will provide our C code with some precomputed values allowing faster parsing. Please remember that this tutorial is all about making the 3D rendering on the PSP the simpler as possible. This is the reason why we are not using some XML format or any other 3D popular format. Also using XML would require additional libraries that are not part of the homebrew SDK making the hole project more complex.

The write header function will iterate through your blender object to collect 3 values:

• flags - A number that represents the color/texture mode.

• number of vertex - Total vertexes in the object.

• number of faces - Total number of faces in the object.

Information about the flags your model can have are available on the *gu.h* file. This header file contains all the constants that the PSP hardware can understand when rendering a mesh. For now we will assume that the base for the flags for any object exported from Blender are:

```
flags = GU_VERTEX_32BITF | GU_NORMAL_32BITF |
GU_INDEX_16BIT
```

This means that we are exporting our vertex data in 32bit floats, normals in 32bit floats and the indexes are just simple 16bit integers. This is the base for all object however some objects are different. If our object has an UV map it means that it is textured, in that case we need to all that to the flags. Also if we painted our object then we need to inform the hardware about the color model. To keep it simple our color model is a 32bit int.

```
def do_export(filename):
  file = open(filename, 'wb')

  vertex_list = []
  vertex_map = {}
  index_list = []

  # Get the current scene and object
  scene = bpy.data.scenes.active
  obj = scene.objects.active

  if obj.type != "Mesh":
    return

  mesh = obj.getData(mesh=1)
```

```
  # Get the object color model/texture mode
  # mesh flags
  flags = GU_VERTEX_32BITF | GU_NORMAL_32BITF |
GU_INDEX_16BIT
  if mesh.faceUV:
    flags |= GU_TEXTURE_32BITF

  if mesh.vertexColors:
    flags |= GU_COLOR_8888

  # Count the number of vertex
  # Count the number of faces

  # write the header of the file with these counts

  # iterate the object and write the vertex data

  file.close()
```

Our export function is getting more complete. One of the three values is already calculated, lets pick the missing 2. This is a simple task, all we need is to iterate all the nodes of the object and count.

```
def do_export(filename):
  file = open(filename, 'wb')
  ...
  # add every vertex
  for vert in mesh.verts:
    vx,vy,vz = vert.co
    nx,ny,nz = vert.no
    v = {"co":(vx,vy,vz), "no":(nx,ny,nz)}
    vertex_list.append(v)
    if v["co"] in vertex_map:
      vertex_map[v["co"]].append(len(vertex_list)-
1)
    else:
      vertex_map[v["co"]] = [len(vertex_list)-1]

  # iterate through faces, filling uv attribute in
vertices
  # and duplicating them if needed
  for face in mesh.faces:
    for vert_i,vert in enumerate(face.verts):
      vx,vy,vz = vert.co
      nx,ny,nz = vert.no
      co = (vx,vy,vz)
      no = (nx,ny,nz)
```

```
    if mesh.faceUV:
      u,v=face.uv[vert_i]
      print "(u,v)=(%f,%f)"%(u,v)
    else:
      u,v=0,0

    if mesh.vertexColors:
      r = face.col[vert_i].r
      g = face.col[vert_i].g
      b = face.col[vert_i].b
      a = face.col[vert_i].a
    else:
      r,g,b,a=255,255,255,255


    # find matching vertex
    for i in vertex_map[co]:
      match = True
      if vertex_list[i]["no"] != no:
        match = False
      if "uv" in vertex_list[i]:
        if vertex_list[i]["uv"] != (u,v):
          match = False
      if "col" in vertex_list[i]:
        if vertex_list[i]["col"] != (r,g,b,a):
          match = False

      if match:
        break

      if match:
        matching_vertex = vertex_list[i]
        matching_vertex["uv"] = (u,v)
        matching_vertex["col"] = (r,g,b,a)
        matching_index = i
      else:
        matching_vertex = {
            "co":co,
            "no":no,
            "uv":(u,v),
            "col":(r,g,b,a)
        }
        vertex_list.append(matching_vertex)
        matching_index = len(vertex_list)-1
        vertex_map[co].append(matching_index)

      index_list.append(matching_index)
      # print matching_index
      print "end face"

# write header
file.write("jcMD")
file.write(
    struct.pack("<III",
    flags,
```

```
      len(vertex_list),
      len(mesh.faces)))
  ...
  file.close()
```

This was simple, we iterated through the object and collect the totals, after we just needed to write initially a control string, so when we parse the file on the PSP we know that we are reading a model file and then the three numbers.

Write the Model

Writing the header was the complicated part, now we need to write the model which is again simply to write the coordinates and UV mapping (if present) to the file.

```
for vert in vertex_list:
  u,v = vert["uv"]
  r,g,b,a = vert["col"]
  nx,ny,nz = vert["no"]
  x,y,z = vert["co"]
  if mesh.faceUV:
    # 8 bytes each
    file.write(struct.pack("< ff", u, v))
  if mesh.vertexColors:
    # 4 bytes each
    file.write(struct.pack("< BBBB", r,g,b,a))
    # 24 bytes each
    f.write(struct.pack("fff fff", nx, ny, nz,
x,y,z))

  for i in index_list:
    file.write(struct.pack("<H", i))
```

Now that we have a full exporter all we need is to import on the PSP side and that will lead us to the next section.

From file to the screen

We now have a simple file with a mesh and are going to render it on the PSP screen. In order to do this I will start by defining a simple data structure to hold the mesh data.

```
typedef struct jcModel {
  int vtype;
  int num_vertices;
  int num_faces;
  unsigned char *vertices;
  unsigned short *indices;
} jcModel;
```

And declare a function to load a file into a *jcModel* structure:

```
int jcModelLoad(jcModel* model, const char* file)
{
  SceUID uid;
  char magic[4];

#ifdef DEBUG
  pspDebugScreenPrintf("Reading model %s\n",
file);
#endif

  uid = sceIoOpen(file, PSP_O_RDONLY, 0777);

  if(uid < 0) {
    ioError("Open fail %s\n", file);
    return -1;
  } else {
    if(0 > sceIoRead(uid, &magic, 4)) {
      ioError("Read magic fail %s\n", file);
      return -1;
    }

    /* Verify the magic */
    if (strncmp(magic, "jcMD", 4)) {
      ioError("Error, not a model file %s\n",
file);
      return -1;
    }

    /* Proceed to read the file */
    if(!jcMeshLoad(&(model->mesh), uid)) {
      sceIoClose(uid);
      return -1;
```

```
    }

    if(!jcTexLoad(&(model->texture), uid)) {
      sceIoClose(uid);
      return -1;
    }

    sceIoClose(uid);
    return 0;
  }
}
```

At the moment we already know how to write and read the binary files we are creating to hold our 3D models. What is so special with this simple format is what comes next, the rendering. Until now it would probably make more sense to use some other widespread formats such as 3DS, OBJ, etc, but you will see now why we did bother to make it a bit complicated.

Rendering is one of the most expensive components of your game so it should be done as fast as possible, using our simplified format this is how it looks like:

```
void jcModelDraw(jcModel* model) {
  jcTexBind(&(model->texture));
  sceGumMatrixMode(GU_MODEL);
  sceGumLoadMatrix(&(model->initialTransform));
  sceGumPushMatrix();
  sceGumTranslate(&(model->position));
  sceGumDrawArray(
      GU_TRIANGLES,
      model->mesh.header.vtype | GU_TRANSFORM_3D,
      model->mesh.header.num_faces * 3,
      model->mesh.indices,
      model->mesh.vertices);
  sceGumPopMatrix();
}
```

As you can see, simply 7 instructions compose the rendering of a full 3D model using the basic SDK, no extra dependencies or libraries.

Getting the Engine

The full engine source code is distributed as a seperate download from this book website and will be updated and made available from *http://www.jetdrone.com*.