-PAlib- Tutorial

Build 2 (removed edit buttons, images fixed!)

Tutorials

Compiled to PDF with open office (By: David Matthews)

If you find any Problems BESIDES THE EDIT BUTTIONS
email me at bobdotexe@gmail.com
I take no Credit for making this tutorial, I just converted it.

<Day 1>

Installing Your Dev Environment

Here we'll give detailed instructions for installing PAlib on Windows, including setting up Visual C++ 2008 Express Edition. Additional sections below cover Linux installation and Mac OS X installation (may need updating). We strongly advice that all Palib users read the PAlib Template Mysteries Explained section, No matter what operating system you use.

Good luck!

Quick Install for Windows

This Quick Install guide will get you trough the process of installing devkitPro and Palib as fast as possible. If you prefer, you can skip this part and follow the detailed guide instead here.

Downloading the required files

- Latest devkitPro Updater
- devkitARM r21
- PAlib Installer

Installing

- Run the devkitPro Updater and install it in "C:\devkitPro"
- Delete or rename the "**devkitARM**" folder in "C:\devkitPro"
- Run the devkitARM r21 installer, install it in "C:\devkitPro"
- Delete the "**libnds**" folder in "C:\devkitPro"
- Run the Libnds Package and install it in, you guessed it, "C:\devkitPro"
- Run the Palib Installer and again, install it in "C:\devkitPro"
- Reboot

Testing

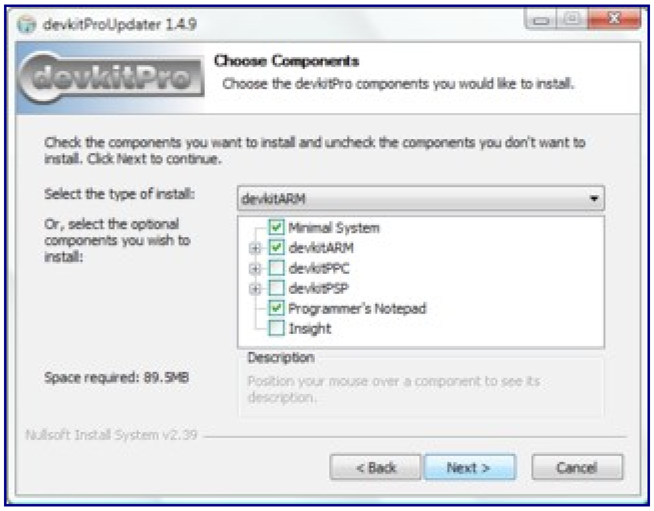
- open "C:\devkitPro\palib\examples\Text\Normal\HelloWorld"
- Double click on "**build**" in that folder.
- Two files should appear, "HelloWorld.ds.gba" and HelloWorld.nds"
- If you get any errors trying to compile the HelloWorld example, please refer to the Forums, remember to use the search function **before** posting your own topic!

Installing on Windows

To set up everything for DS development, you must first download and install devkitPro, to do that you must first download the devkitPro Updater, don't worry about the name "updater", you can use it to install it for the first time too.

Installing devkitPro

Double click the installer you just downloaded, keep "keep downloaded files" if you plan to install devkitPro on other computers aswell, that saves you from downloading the same files again. You will then arrive at a screen with some checkboxes. For developing NDS games and applications DevkitARM is required, so go ahead and select "devkitARM" from the dropbox, you should end up with a screen simmilar to this one.


devkitProUpdater 1.4.9

Choose Components

Choose the devkitPro components you would like to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: devkitARM

Or, select the optional components you wish to install:

- [✓] Minimal System
- [✓] devkitARM
- [ ] devkitPPC
- [ ] devkitPSP
- [✓] Programmer's Notepad
- [ ] Insight

Description

Position your mouse over a component to see its description.

Space required: 89.5MB

Nullsoft Install System v2.39

Click next and choose the default install location, which is "**C:\devkitPro**". Please note that you can't isntall devkitPro or Palib in a directory with a space in it.

Replacing devkitARM

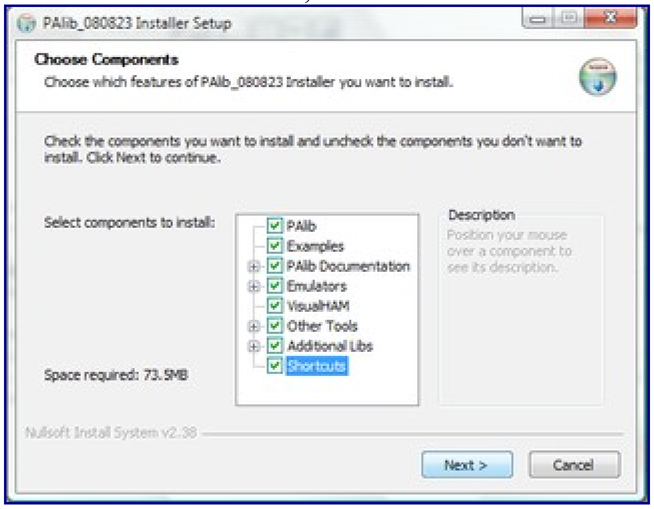
Sadly, the current version of Palib is not compatible with the latest version of devkitARM, so we have to swap it for an older version. To do so, you need to download devkitARM r21 which you can download here then rename or simply delete the old "devkitARM" in "C:\devkitPro" and run the program you just downloaded, extracting the files to C:\devkitPro

Replacing Libnds

Now you need to downgrade your version of "libnds", "libfat" and "dswifi". Andrew Hathaway is hosting a ZIP file of a working libnds directory. You can download the ZIP file here. Delete or rename the "old" libnds folder, and extract the ZIP file to C:\devkitpro

Installing Palib

Now, it's finally time to download and install Palib itself, you can download the installer here. Install Palib in "C:\devkitPro", it's best to leave all boxes ticked so you are sure that you wont miss anything.


PAlib_080823 Installer Setup

Choose Components

Choose which features of PAlib_080823 Installer you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] PAlib
- [✓] Examples
- [✓] PAlib Documentation
- [✓] Emulators
- [✓] VisualHAM
- [✓] Other Tools
- [✓] Additional Libs
- [✓] Shortcuts

Description

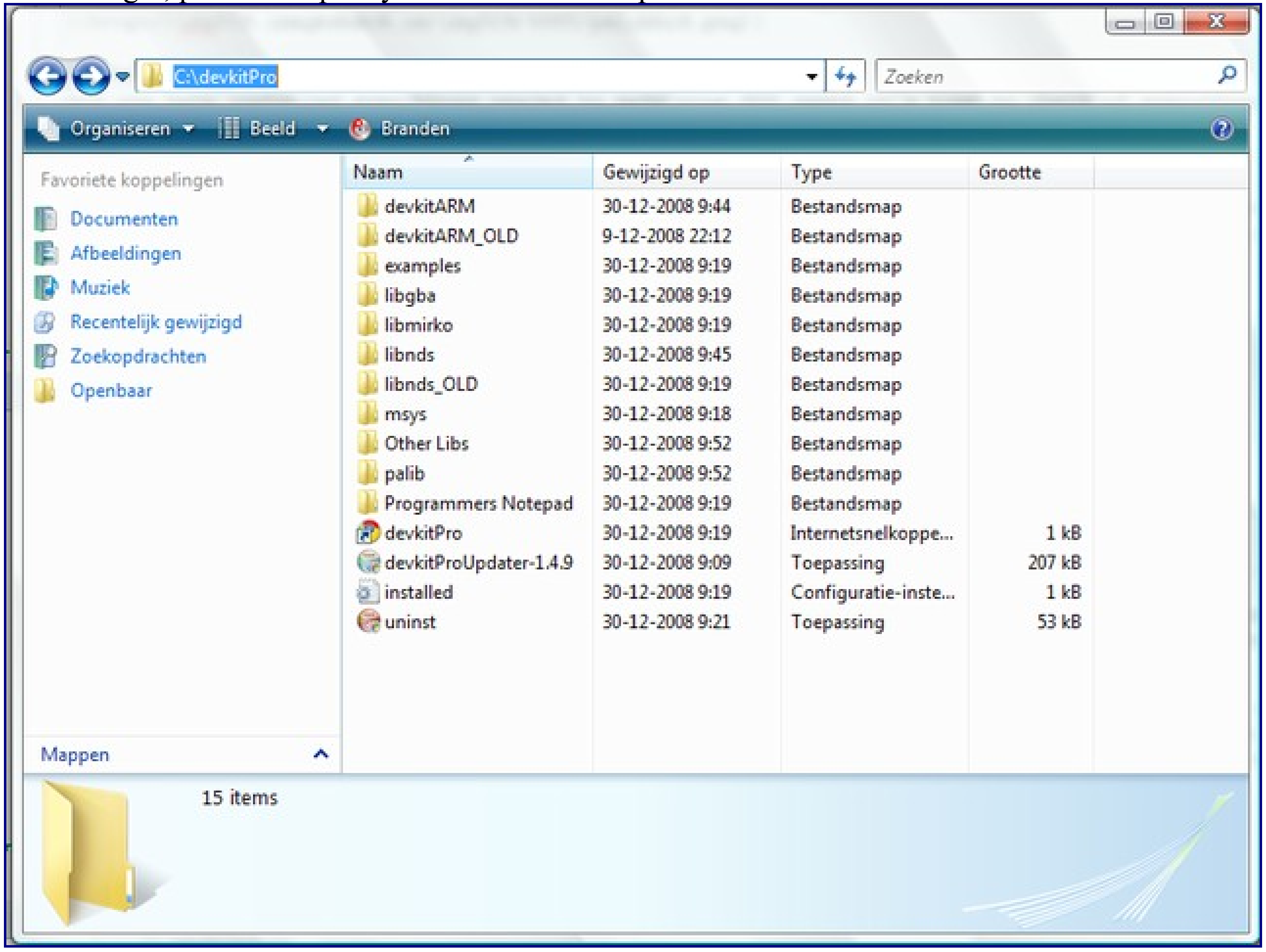
Position your mouse over a component to see its description.

Space required: 73.5MB

Nullsoft Install System v2.38

Final Folder Layout

Now you have installed everything needed to make your NDS games it's time to check if everything went allright, please compare your folder with the picture below.



Naam	Gewijzigd op	Type	Grootte
devkitARM	30-12-2008 9:44	Bestandsmap	
devkitARM_OLD	9-12-2008 22:12	Bestandsmap	
examples	30-12-2008 9:19	Bestandsmap	
libgba	30-12-2008 9:19	Bestandsmap	
libmirko	30-12-2008 9:19	Bestandsmap	
libnds	30-12-2008 9:45	Bestandsmap	
libnds_OLD	30-12-2008 9:19	Bestandsmap	
msys	30-12-2008 9:18	Bestandsmap	
Other Libs	30-12-2008 9:52	Bestandsmap	
palib	30-12-2008 9:52	Bestandsmap	
Programmers Notepad	30-12-2008 9:19	Bestandsmap	
devkitPro	30-12-2008 9:19	Internetsnelkoppe...	1 kB
devkitProUpdater-1.4.9	30-12-2008 9:09	Toepassing	207 kB
installed	30-12-2008 9:19	Configuratie-inste...	1 kB
uninst	30-12-2008 9:21	Toepassing	53 kB

.Net Framework
Finally, certain parts of Palib require that you have the .Net Framework installed. This is needed for PAGfx (graphics converter), PAFS (file system), and the VHam (IDE) install... Get .Net here!

That's it. You're done with the installation! Reboot your computer now to proceed.

The compiling test
To make sure your install works correctly we are going to compile (build from sourcecode) an example that comes with Palib. To do that open the following directory: "C:\devkitPro\palib\examples\Text\Normal\HelloWorld". Then double click on "build" a black box with some text appears, and if you installed everything correctly, two files should appear in the folder, "HelloWorld.ds.gba" and "HelloWorld.nds". If you get an error however, refer to the Forum and use the search function to search for a solution, if you still can't make it work create your own topic in the appropriate section. Don't forget to post the error you get!

— *DblK 29/11/2008* If, like me, you had problems with the VisualHam editor after installing the setup "PAlib 080823", I've done a new Setup to fix those problems : PALib Install Corrector

PAlib Template Mysteries Explained

Now that you've got a working installation of PAlib, you'll probably want to jump right in and start coding. You should start by checking out the PAlib template in the \PAlib\template\ folder. Go there and you can see that it contains a few files, and three directories. I'll explain a bit about how it works...

Template Description

Lol, just thought of something... it has been asked before.... What is a template? I know! You could say that a template is a device to provide the form or structure for content. In other words, it's an empty structure that you will eventually fill with all your development files for a given project like C/C++ (code) files, graphics images, sounds, etc. Then, because you used the template that was already set up with the correct structure, you'll be able to compile your working NDS file easily.

Here's the structure of the PAlib template:

The directories :

- *source* → contains the most important part, the source code, which is basically the executing code (.c or .cpp files)
- *include* → will contain the header files, which are sort of helpers/informations about the source files (.h files)
- *data* → will contain some files for images, sound, etc...

The files :

- *Makefile* → important file explains how to compile the program, you'll need to open it up and change some parts of it for your projects
- *logo.bmp* → icon image that will be appended to your NDS file, please customize it for your projects
- *build.bat* → important file used to build (doh !!) the NDS file of your project
- *clean.bat* → when you want to release the source code, or when you're finished coding, clean.bat cleans up your project directory to make the total size much smaller. It removes all files that are created when using build.bat.
- *project.vhw* → this one is cool–it opens the whole project in VisualHam, the editor that comes with PAlib...

Template's Main Code

Now you should open project.vhw (VisualHam), template.pnproj (Programmer's Notepad) or just go into the "source" folder and open main.c in the editor of your choice. Like its name suggests, main.c is your main program file. It contains the main() function and all the stuff that'll be loaded by default when the rom starts. The template's main.c contains all the basic stuff needed to run PAlib. If you compiled and ran the rom in an emulator or on DS, it wouldn't do anything, though, just display 2 white screens because you haven't done anything yet but initialize the basics...

Here is what you see when you first open main.c :

```
// Includes
#include <PA9.h>       // Include for PA_Lib
```

```
// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL

        // Infinite loop to keep the program running
        while (1)
        {
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

Decoding the Code !

If you already know C programming, this is probably very clear to you. For others, it might not be, so I'll detail everything line by line, even though the comments are there...

```
// Includes
```

Simple : // Means that the following text is a comment... When putting //, you actually tell the compiler to not compile what comes right after that... so you can write any text as comments ! This line is just there to tell that you can put files to include here (images, etc...)

```
#include <PA9.h>
```

This means : please include PAlib into my code, I'll need it ! If you don't put this, the PAlib functions will not be recognized, and you will get errors. The concept of using includes will become very important later on as includes are used to add in graphical files, other important code, and more.

```
int main(int argc, char ** argv)
```

This may seem weird, but you'll have in every project : it's the start of your program, where everything begins. Don't bother trying to get what the (int argc, char argv) stands for, you don't care and you'll never use it :) just know that anything beyond the '{' will be the executing code !

```
PA_Init();    // Initializes PA_Lib
PA_InitVBL(); // Initializes a standard VBL
```

These 2 lines will be found in every project using PAlib. The first one is a general initialization. It prepares your program to use sprites, backgrounds, text, everything. Not putting this will prevent PAlib from working. The second one, with the VBL, can be modified by advance users. Basically, the VBL allows to synchronize the program with the screen, at 60 frames per second. If you don't initialize the VBL, either the program will work way faster (like 100000 frames per second, which is unplayable) or not work at all (staying stuck at the first frame). Always leave these two commands in your program.

```
// Infinite loop to keep the program running
        while (1){
```

The while command executes the code that follows, in brackets, while the result of the command in it (here, it’s 1), is different from 0. Here, it’s always 1, so basically this is a loop that will go on forever. What for ? Well, if you don’t have such a loop, every line of code would be executed just once, and then the program would exit, not displaying anything anymore. This allows the program to continue forever, or at least until the player hits that power button. The concept is that in each run through the while loop, the program will figure out everything it needs to know for the next time it will draw to the screen and then it will draw it.

```
PA_WaitForVBL();
```

This last command synchronizes the infinite loop we just saw with the screen rate (60 fps). If you don’t put it, the program will function, like said before, MUCH faster than the screen rate and many things just won’t work at all.

```
        }

        return 0;
} // End of main()
```

All this is the closing code, always there, always like this, don’t bother about it, just leave it alone :p

Now that you know everything about the template, go on and see how to display text !

— *Micheal Bach 30/11/2005 09:37*modified 24/08/2008 12:04

Making a Hello World

Let’s alter the template to make a simple Hello World just to show how easy it can be... This will not enter into the real details of programming on the DS yet, just show you that it can be really simple and already possible on Day 1 :p

To make a Hello World example, we’ll use PAlib’s basic text system. The text system uses the backgrounds–well, one background (you have four of them per screen)–and you can choose which screen and which background to use. Some more functions are available, like setting the text font, or changing it’s color, and much much more, but for now we’ll keep it simple :p

All you need to do is add the following lines to the template’s main.c file just after PA_InitVBL();

```
        PA_InitText(1, 2); // Tell it to put text on screen 1, background number 2
        PA_OutputSimpleText(1, 1, 1, "Hello World !");  // Print the desired text
on screen 1, with coordinate 1, 1
```

The first line initializes the text system, by loading the text font into the DS’s video RAM. If this wasn’t done, then the text output would look like garbage... it would be like writing, but replacing each letter by a random drawing... not too good ;) PA_InitText takes 2 arguments :

- Screen (0 for bottom screen or 1 for top screen)
- Background to use (0 for the topmost layer, 1 for the next, 2, and 3 for the bottom most one...)

PA_OutputSimpleText, like the name tells you, outputs simple text on the screen...IF the background was initialized with InitText. (At this point, you probably realized that all the PAlib functions start with PA_...) The OutputSimpleText function has 4 arguments :

- Screen (0 or 1)

- Horizontal (X) tile from which to start, from 0 to 31. Each tile is a 8×8 pixel square, so tile 0 is at position 0, tile 1 at position 8 (in pixels), etc...
- Vertical tile position, same as horizontal, but from 0 to 23...
- Text (your text must be enclosed in double-quotes ("), that's how the C compiler recognizes it...)

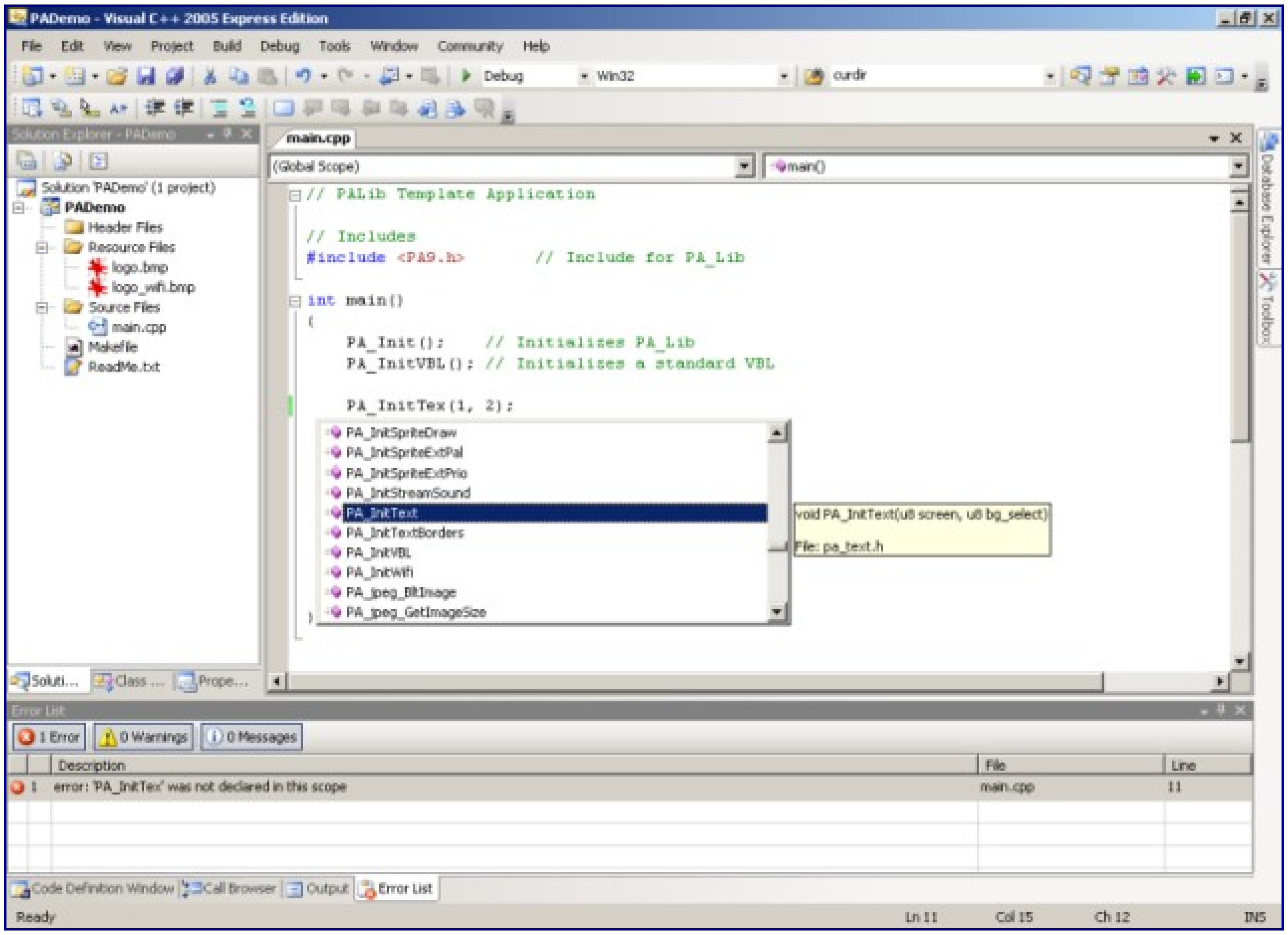
Once you understood this, if you're using VisualHam, just press F6 to compile and launch in DualiS, or F7 to compile and launch in DeSmuME ! Pressing F5 does a normal compilation without loading the rom (can be used to compile and then pass on your DS...). Or click on the build.bat file.

Using PALib With Visual C++ 2008 Express

Original tutorial by Rodrigo Sieiro revised for VC++ 2008 by Christian Crawford

As a developer, I work with Visual Studio every day. So, when I decided to start programming for NDS, I found myself trying a lot of different IDEs, hoping to find the features I had in Visual Studio. I tried Programmers Notepad, VisualHAM, EditPlus, PSPad, and so on. No one seemed to satisfy me.

So I started researching how could I use the (free) Visual C++ 2008 Express with PALib. And I learned a few things from different places: a little in PALib's French Wiki (I don't read French, just followed the figures), a little in devkitPro's FAQ, until I could manually make a PALib project inside Visual C++. Then I found OGRE's (a 3D Engine) AppWizard for VCExpress, so I tweaked my base project a bit and made it an AppWizard.



```
// PALib Template Application

// Includes
#include <PA9.h>       // Include for PA_Lib

int main()
{
    PA_Init();    // Initializes PA_Lib
    PA_InitVBL(); // Initializes a standard VBL

    PA_InitTex(1, 2);
```

```
PA_InitSpriteDraw
PA_InitSpriteExtPal
PA_InitSpriteExtPrio
PA_InitStreamSound
PA_InitText
PA_InitTextBorders
PA_InitVBL
PA_InitWifi
PA_Jpeg_BltImage
PA_Jpeg_GetImageSize
```

```
void PA_InitText(u8 screen, u8 bg_select)
File: pa_text.h
```

```
1  error: 'PA_InitTex' was not declared in this scope   main.cpp   11
```

In this guide I'll explain how you can set up a fully working development environment with Visual C++ 2005 Express and PALib. As you can see in the image above, here are some features you can expect by using this method:

- Fully integrated IDE with Project Management
- Start working with a basic project in a few clicks
- Code Completion / IntelliSense
- Compilation errors reported in the IDE
- Build your whole app by pressing a key
- Test you app by pressing another key

Prerequisites

Before you begin, make sure you already installed the following stuff:

- Visual C++ 2008 Express (free download from Microsoft)
- devkitPro
- PALib

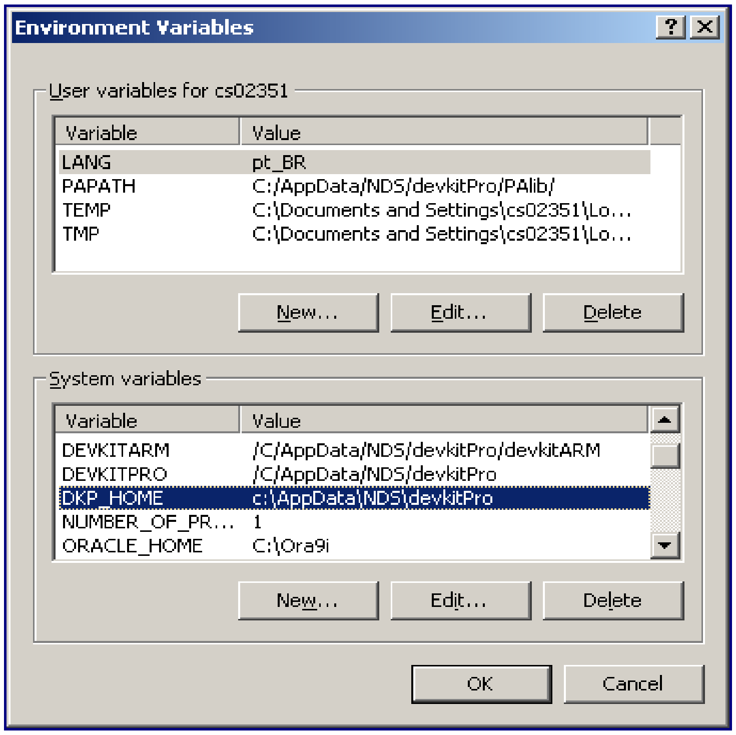
Follow the installation guide above to be sure you can at least compile and run a basic project (using any editor and "make" command line), so you can be certain that your basic environment is configured correctly before configuring Visual C++ Express.

Important: I suggest you to install PALib inside devkitPro folder, because the AppWizard (code completion, mainly) probably won't work if you install it in a different place.

Setting up the Environment

When you install devkitPro and PALib, they already add some environment variables for you. But these variables are for the make tool, so the path names are written in "UNIX Style". So, in order to compile PALib and devkitPro inside Visual Studio, first we need to set a new Environment Variable.

Open "Start → Settings → Control Panel → System". Select "Advanced" then click on "Environment Variables". Click on the "New" button inside "System Variables" (the lower one) and add a variable with the name "DKP_HOME". In the "Value" box, write the root folder of your devkitPro installation, "Windows Style" (i.e. "c:\..." instead of "/c/...").



User variables for cs02351

Variable	Value
LANG	pt_BR
PAPATH	C:/AppData/NDS/devkitPro/PAlib/
TEMP	C:\Documents and Settings\cs02351\Lo...
TMP	C:\Documents and Settings\cs02351\Lo...

System variables

Variable	Value
DEVKITARM	/C/AppData/NDS/devkitPro/devkitARM
DEVKITPRO	/C/AppData/NDS/devkitPro
DKP_HOME	c:\AppData\NDS\devkitPro
NUMBER_OF_PR...	1
ORACLE_HOME	C:\Ora9i

Pay attention on the environment variable "PAPATH". If you have installed the packages as administror your other users don't have this variable automatically. Without creating it manually for the user, the compilation will fail with an error message saying "PA9.h" couldn't be found.

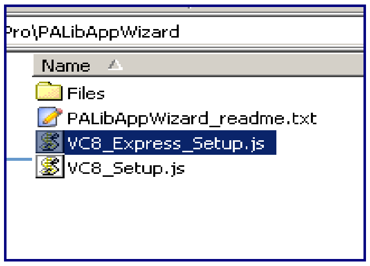
If running Windows Vista you will need Admin rights and you go "Start → control panel → system" then click on the "change settings" it will open the window above and click "Advanced"

Installation

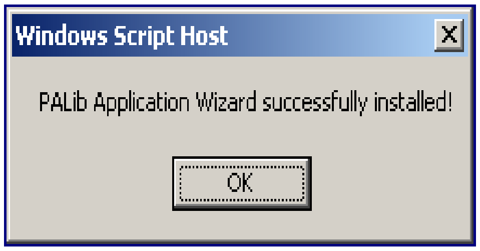
Get the AppWizard here or alternatively you can download NightFox's version which is installed the same way and include his libraries (eg: EFS) here Now let's install the PALib AppWizard. Extract the contents of the ZIP file inside a PERMANENT folder. The template files will stay there and can't be removed, otherwise you will break your AppWizzard. I recommend using a folder called

“PALibAppWizard” inside devkitPro root folder, so it sits together with “PALibExamples” and “PALibTemplate”. That’s what I use.

After you extract the contents, you will see two JavaScript files: “VC9_Express_Setup.js” and “VC9_Setup.js”. As you probably figured, the first is for Visual C++ 2008 Express and the second is for other versions of Visual C++ 2008. Double-click the one corresponding to your version and it will install the wizard automatically. I only have Visual C++ 2008 Express, so don’t blame me if the wizard doesn’t work with VC++ 2008 Pro. I can’t test it. From DtD Software: I have tested the script with Microsoft Visual Studio 2008 Pro and it works like a charm.



If everything went well, you will see the following message:


Windows Script Host

PALib Application Wizard successfully installed!

If running Windows Vista you will see the message above but it wont install properly. manually copy the files explained below.

Alternate Installation Method 1 - Start->Run

If your .js files are registered to another program, such as Dreamweaver, the previous method won’t work (the script will open inside the editor, and won’t run). If that’s your case, you can run the .js script manually, via the Run box. Click on Start then Run and type:

WScript.exe <location of setup js>

The location above should be the full path and filename of the .js file you need to use. That should work.

Alternate Installation Method 2 - Manually Copying Files

If you are afraid that I put inside the .js some ultra-power virus that will explode your monitor and erase all DVDs up to 1 mile far from your computer, you can manually copy the AppWizard files to their location. Follow this steps:

1. Open PALibAppWizard.vsz in your text editor.
2. Change “ABSOLUTE_PATH” to the folder “Files” you extracted the AppWizard ZIP.

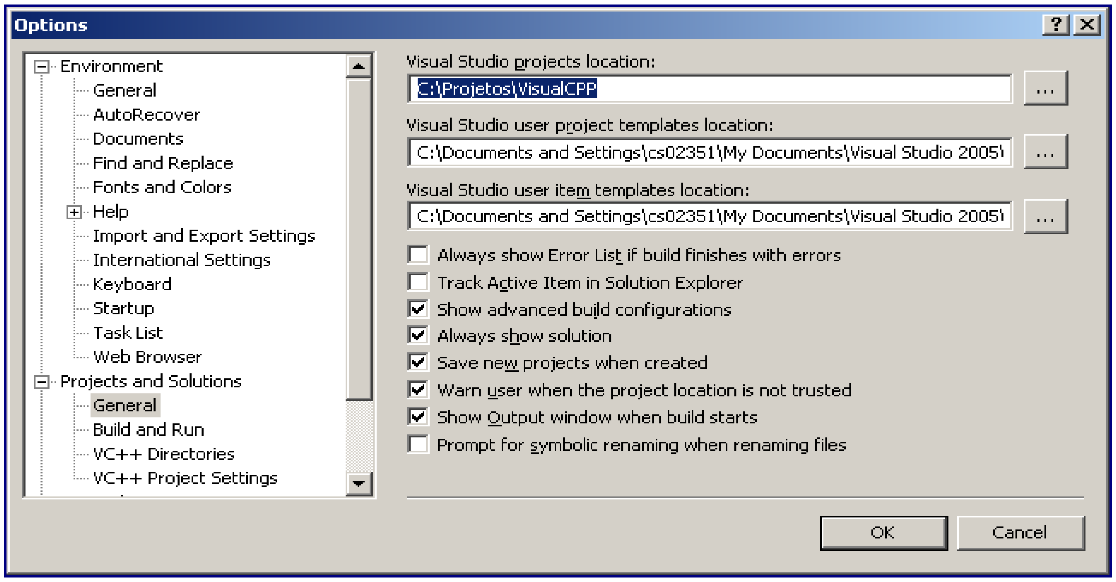
3. Copy “PALibAppWizard.ico” and “PALibAppWizard.vsz” to:
   - [Visual Studio]/VC/Express/VCProjects
4. Copy “PALibAppWizard.vsdir” to:
   - [Visual Studio]/VC/Express/VCProjects/NintendoDS
   - (You can create the “NintendoDS” folder manually)

That’s it, your AppWizard is installed. Now follow the rest of the guide to setup some more things in your Visual Studio.

Building the First Project

Open Visual C++. Before you rush on clicking “New Project” to see PALib there, let’s do one more thing that will be very helpful. Since the “make” tool we use to compile PALib project doesn’t like long filename and directories with spaces, I suggest you to change your default “Projects” folder inside Visual C++ so you won’t have to change it everytime you create a new project.

Go to “Tools”, then “Options”, then “Projects and Solutions”, then “General”. Change the field “Visual Studio Projects Location” to a folder with no spaces. I suggest something short, like I used in the picture below.



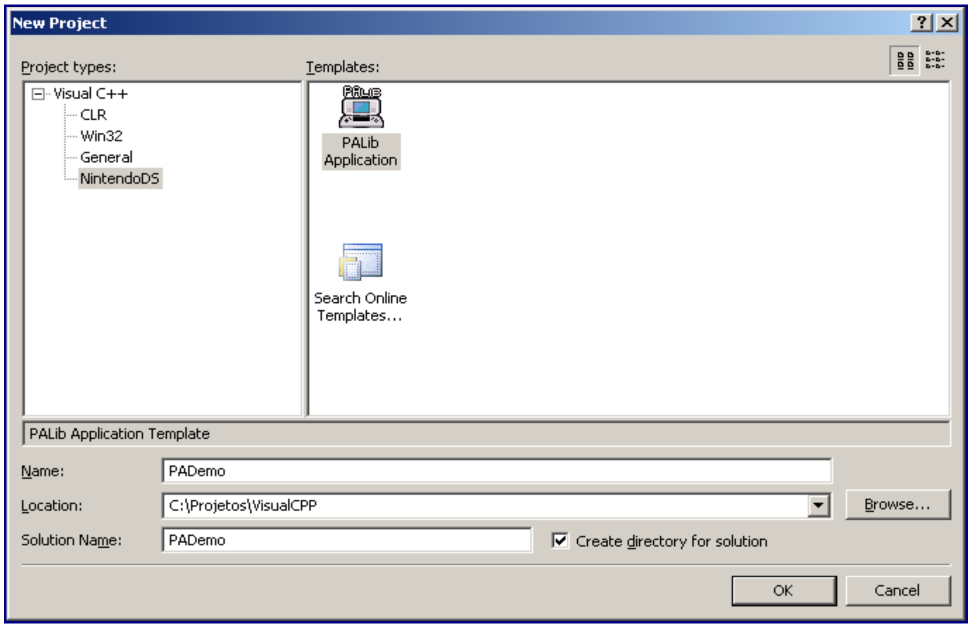
Visual Studio projects location: C:\Projetos\VisualCPP

Visual Studio user project templates location: C:\Documents and Settings\cs02351\My Documents\Visual Studio 2005\

Visual Studio user item templates location: C:\Documents and Settings\cs02351\My Documents\Visual Studio 2005\

- [ ] Always show Error List if build finishes with errors
- [ ] Track Active Item in Solution Explorer
- [x] Show advanced build configurations
- [x] Always show solution
- [x] Save new projects when created
- [x] Warn user when the project location is not trusted
- [x] Show Output window when build starts
- [ ] Prompt for symbolic renaming when renaming files

Now you can click on “File” → “New” and select “New Project”. If you installed the AppWizard correctly, you will see a new “NintendoDS” category, and inside it, you have the “PALib Application” wizard. This is a pre-configured project, with all the things you need to do in order to compile PALib project inside Visual C++ already configured for you. Type a Project Name, check the “Location” to be sure you don’t have a directory with spaces, and click “OK” to start the wizard.


New Project

Project types:
- Visual C++
  - CLR
  - Win32
  - General
  - NintendoDS

Templates:
- PALib Application
- Search Online Templates...

PALib Application Template

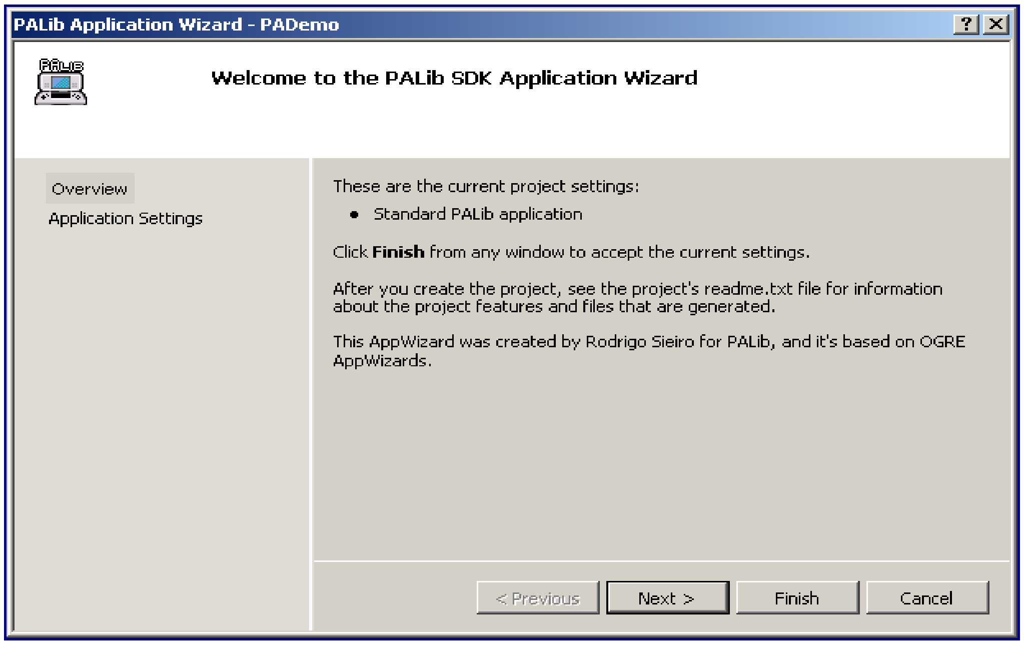
Name: PADemo

Location: C:\Projetos\VisualCPP

Solution Name: PADemo

[✓] Create directory for solution

There's not much stuff you can change in the wizard. I tried to think about some options I could put there but my creativity wasn't cooperating with me when I wrote the wizard. The first screen is an overview of the project you are creating. Click on "Next" when you are ready to proceed.


PALib Application Wizard - PADemo

Welcome to the PALib SDK Application Wizard

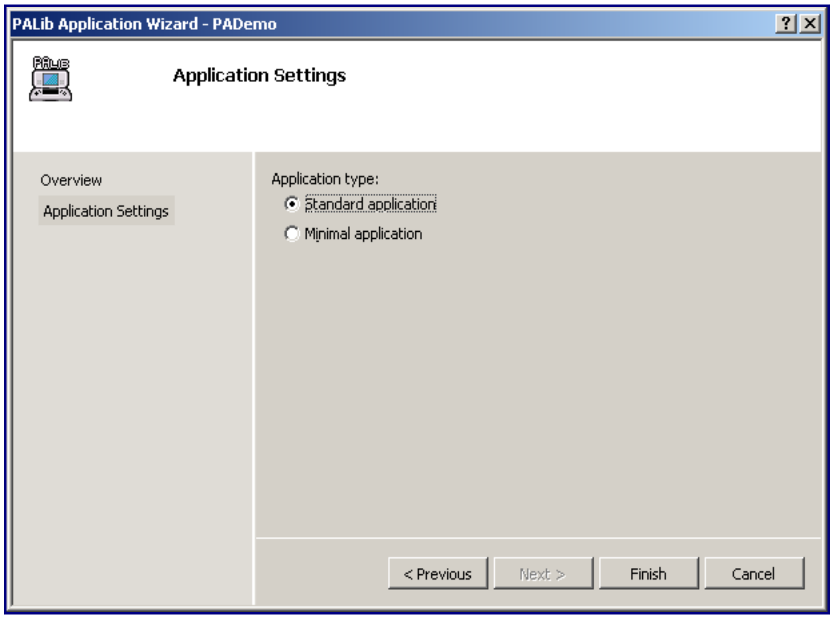
These are the current project settings:
- Standard PALib application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

This AppWizard was created by Rodrigo Sieiro for PALib, and it's based on OGRE AppWizards.

In the second page, you can choose between "Standard application" and "Minimal application". "What's the difference?", you may ask. Simple: the last one is the very minimal you need in order to build a PALib project. You can compile and run its generated application, but it won't do anything. The first is a fully functional "Hello World" application, with two more lines of code that actually do something. Now that's a huge difference! Click on "Finish" to generate your project.

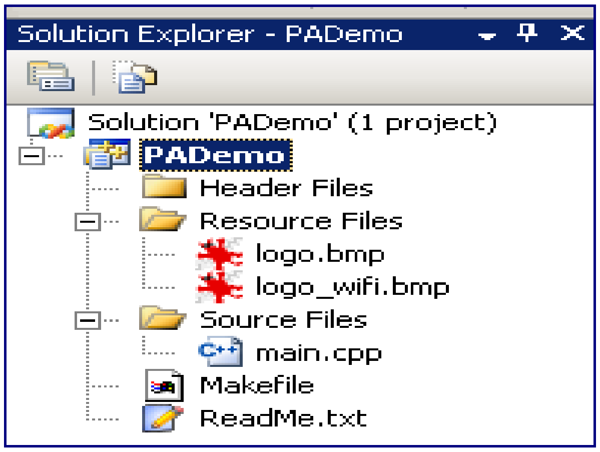


Application type:
(●) Standard application
( ) Minimal application

Now let’s check out the files that were generated by the wizard:

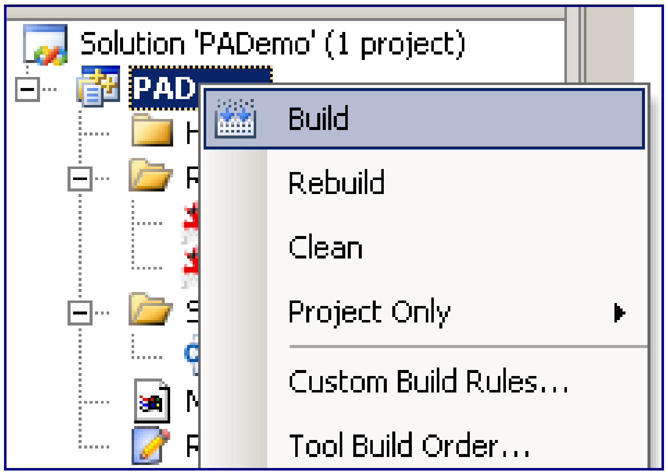
- logo.bmp → this is the image your app will show in DS Loaders.
- logo_wifi.bmp → the logo to use with WiFi.
- main.cpp → the main program file, with some code already written for you.
- Makefile → a PALib makefile optimized for Visual Studio.
- ReadMe.txt → some info.

Notice how Visual Studio can organize your project by content type... That’s even better when you have a lot of files inside your project!



```
Solution 'PADemo' (1 project)
  PADemo
    Header Files
    Resource Files
      logo.bmp
      logo_wifi.bmp
    Source Files
      main.cpp
    Makefile
    ReadMe.txt
```

I’ve tweaked the default PALib Makefile (found in the PALibTemplate) so it can work better with Visual Studio. This way, you can use all the build options Visual Studio offers: “Build”, “Rebuild” and “Clean”. Rebuild is actually the same as selecting “Clean” then “Build”.



Okay, we're set, let's try building our project for the first time. Click on "Build" then "Build Solution", or press F7. Visual C++ will start compiling your project using the Makefile. You can see what's it doing in the "Output" window. It will compile all your source files, link them in a binary file, and then call the tools that generate three files for you: a ".nds", a "ds.gba", and a ".sc.nds". These files will be generated in the "Release" subfolder of you Project. If the last line in the Output window says "Build: 1 succeeded", then everything went well. If you didn't touch the code and get an error, you're screwed. So you should delete DevKitPro and PaLib and then install it again and following the stpe from the start making sure to not miss out anything.

-Edit:

If you are receiving a compiling error of: c:/devkitPro/PAlib/include/nds/arm9/PA_Draw.h: In function 'u16 PA_GetBmpWidth(void*)': c:/devkitPro/PAlib/include/nds/arm9/PA_Draw.h:689: error: pointer of type 'void *' used in arithmetic c:/devkitPro/PAlib/include/nds/arm9/PA_Draw.h: In function 'u16 PA_GetBmpHeight(void*)': c:/devkitPro/PAlib/include/nds/arm9/PA_Draw.h:704: error: pointer of type 'void *' used in arithmetic

This can be fixed by opening PALib/include/nds/arm9, finding PA_Draw.h

'I Had to Change

extern inline u16 PA_GetBmpWidth(void *bmpdata){

```
 BMP_Headers *Bmpinfo = (BMP_Headers*)(temp+14);
 return Bmpinfo->Width;
```

}

extern inline u16 PA_GetBmpHeight(void *bmpdata){

```
 BMP_Headers *Bmpinfo = (BMP_Headers*)(temp+14);
 return Bmpinfo->Height;
```

}

to:

extern inline u16 PA_GetBmpWidth(void *bmpdata){

```
 u8 *temp = (u8*)bmpdata;
 BMP_Headers *Bmpinfo = (BMP_Headers*)(temp+14);
 return Bmpinfo->Width;
```

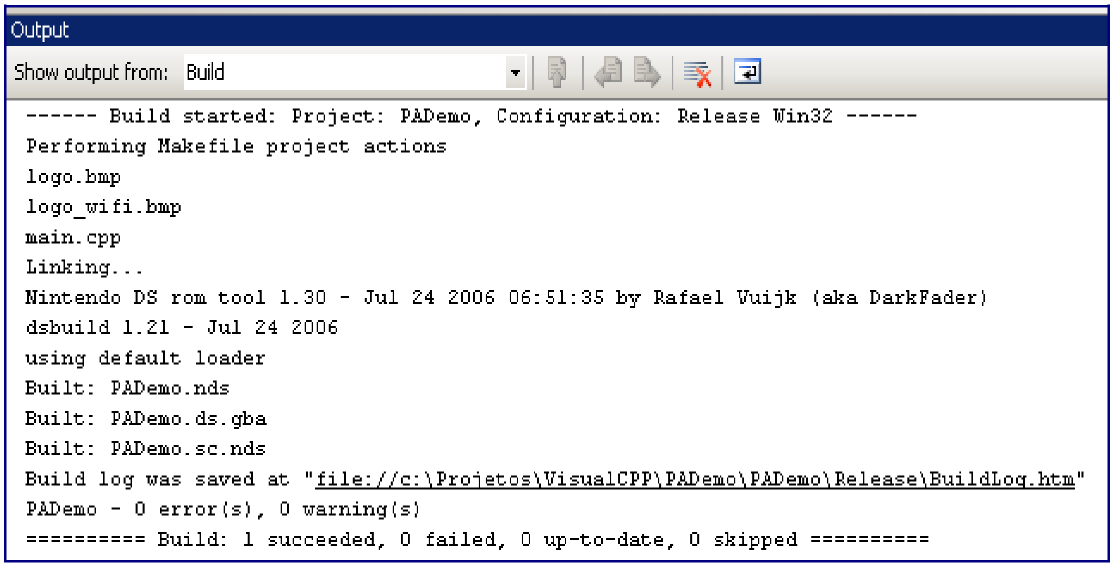
}

extern inline u16 PA_GetBmpHeight(void *bmpdata){

```
u8 *temp = (u8*)bmpdata;
BMP_Headers *Bmpinfo = (BMP_Headers*)(temp+14);
return Bmpinfo->Height;
```

} ' Credit goes to Frosty Chaotix for this helpful fix.

NOTE: For Windows Vista users, go to Project → [projname] Properties. Expand the "Configuration Properties" tree on the left bar and go to NMake. Click on the "Build Command Line" field and a little box with ... in it should appear on the right. Click that to bring up a text area to type out the commands. Insert a new line before what's already there and add "set PATH=C:\devkitPro\devkitARM\bin;C:\devkitPro\devkitARM\arm-eabi\bin;C:\devkitPro\devkitARM\libexec\gcc\arm-eabi\4.1.1;%PATH%" to it (without the quotes). Save it and you should be able to compile correctly. I got some errors in a couple sound headers from PALib about incorrect conversions from void pointers to u32 pointers. If you double click on the error, it will take you to the line. Add a (u32*) to the line so it's like "blahblah = (u32*)blaghblagangjklaglk;" on both lines that have the error. After that, the compile worked like a charm for me and NO$GBA read the file just fine. – Ari Velazquez (arivelz@gmail.com)



```
------ Build started: Project: PADemo, Configuration: Release Win32 ------
Performing Makefile project actions
logo.bmp
logo_wifi.bmp
main.cpp
Linking...
Nintendo DS rom tool 1.30 - Jul 24 2006 06:51:35 by Rafael Vuijk (aka DarkFader)
dsbuild 1.21 - Jul 24 2006
using default loader
Built: PADemo.nds
Built: PADemo.ds.gba
Built: PADemo.sc.nds
Build log was saved at "file://c:\Projetos\VisualCPP\PADemo\PADemo\Release\BuildLog.htm"
PADemo - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

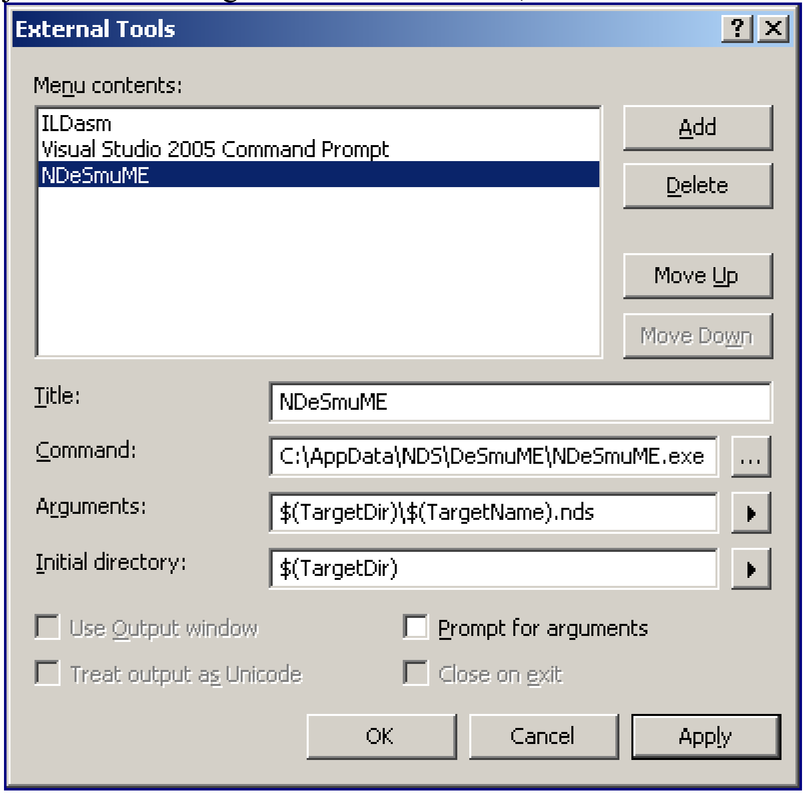
Lastly, don't forget to set up Visual C++ directory variabiles in order to get IntelliSense to work with the PALib! Go to Tools → Options → Project & Solutions → VC++ Directories, then select "Include files" from the top-right scroll menu, and add "<path to PALib>\include\nds" and "<path to PALib>\libnds\include" (without the quotes, anche replace <path to PALib> with the folder where you installed the PALib). Now by pressing Ctrl+Space VC++ will auto-complete for you PALib functions and classnames!

If you are having problems with compiling after following these steps, try reseting your computer.

Setting Up an Emulator inside Visual C++

This step is optional, but I highly recommend it. We are going to setup a NDS emulator as an External Tool, so you can test your project quickly. Click on "Tools", "External Tools". Then click on "Add" to create a new External Tool. You can create an External Tool for each emulator you use, but here I show you how to create one for NDeSmuME, the one I use.

Type the emulator name in the “Title” field, then the full path to the emulator EXE file in the “Command” field. In the “Arguments” field you can type “$(TargetDir)\$(TargetName).nds” (without the quotes), and in “Initial Directory” type “$(TargetDir)”. This way you’re passing the .nds file you just built as an argument to the emulator, so it will start automatically.

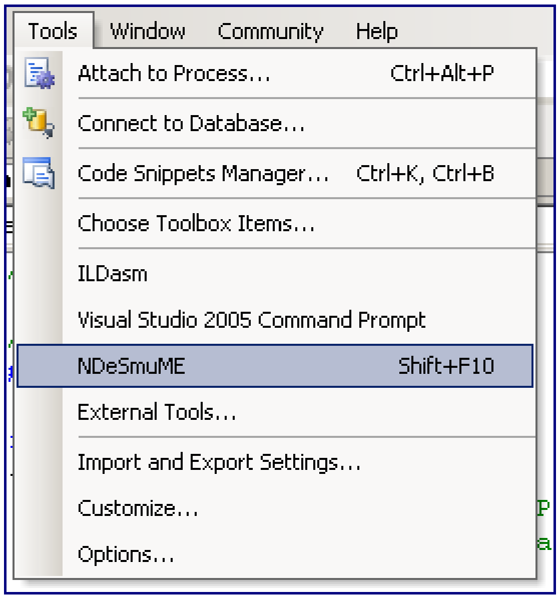


Title: NDeSmuME
Command: C:\AppData\NDS\DeSmuME\NDeSmuME.exe
Arguments: $(TargetDir)\$(TargetName).nds
Initial directory: $(TargetDir)

If you use an emulator such as No$GBA, you must change the “.nds” to “.ds.gba” without the quotes in the Arguments field.

You can even configure a keyboard shortcut (inside “Tools”, “Options”, “Environment”, “Keyboard”), so you can test you app quickly. To map a key to your emulator, select the command “Tools.ExternalCommand[x]” from the list, where [x] is the order of your tool in the list. For example, my emulator is the third tool in the list, so its command is “Tools.ExternalCommand3”. My shortcut is “Shift+F10”, so when I change something in my code I just press “F7” to build and “Shift+F10” to test. Easy, huh?

You can configure a script to copy the file to your card as an External Tool, so it’s easier to test you application in your NDS. To set up that script you follow the above to make a new external tool. Then put “C:\WINDOWS\system32\xcopy.exe” in the command box (this is for xp might be diffrent in vista) then put “$(ProjectDir)\Release\$(TargetName).nds X:\ /y” (were x is the letter of the drive) in the Arguments box and finally “$(TargetDir)” in the initial directory box.


Tools
Window
Community
Help
Attach to Process... Ctrl+Alt+P
Connect to Database...
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
ILDasm
Visual Studio 2005 Command Prompt
NDeSmuME Shift+F10
External Tools...
Import and Export Settings...
Customize...
Options...

Conclusion

That's it. Now you have a fully working IDE to develop your NDS games and applications. Obviously you can't debug your project using Visual C++ debugging environment (altough it would be great) but everything else should work flawlessly.

Update: I was getting a bug with the latest PALib in VCExpress, here is the fix: http://forum.palib.info/index.php?topic=4200.0

In Linux We Trust...

Original tutorial by KerneL, revised by ObsidianX :)

Alright, this is a nice step-by-step method for getting PAlib setup and compiling under Linux. I've got Ubuntu Linux running, but its pretty generic for the most part so just fix the distro-specific stuff yourself ;)

For Apple MacOSX users, the majority of this howto will work for you too. The only difference being your 'make' tool installation. For that you'll want XCode.

Quick Install

Script By Loumam, posted here by LynkW. Go here: http://ubuntuforums.org/showthread.php?t=750050

Requirements

These are the components needed to set up the development environment on Linux.

The lastest version of PAlib, available as a ZIP archive:

- http://www.palib.info/downloads/Stable/PAlib_080823_Archive.zip

devkitARM release 21 for Linx: (this has to be release 21, later releases are incompatible with PAlib 080823)

- http://sourceforge.net/project/showfiles.php?group_id=114505&package_id=124207

libnds 20071023

- http://downloads.sourceforge.net/devkitpro/libnds-20071023.tar.bz2?modtime=1193158736&big_mirror=0

dswifi 0.3.4

- http://downloads.sourceforge.net/devkitpro/dswifi-0.3.4.tar.bz2?modtime=1194419172&big_mirror=0

libfat-nds 20070127

- http://downloads.sourceforge.net/devkitpro/libfat-nds-20070127.tar.bz

Folders organization

First of all, we must organize this mess. Create a new directory named "devkitPro" and make sure you have read/write/execution rights on it. This guide will assume that this folder is in your home directory, but it doesn't have to be. Extract everything in the PAlib, devkitARM, and libnds archives to the devkitpro directory. Then extract dswifi and libfat-nds into the libnds folder.

Now, you should have something close to the following tree view:

```
/
|--home
       |--<login>
             |--devkitPro
                |-devkitARM
                |--libnds
                     |-dswifi
                     |-libfat-nds
                |-PAlib
```

Environment variables

The installation of PAlib requires three environment variables. The best way to do this is to open the "~/.bashrc" file (if you use something other than bash then use the appropriate rc file). Open your favorite text editor and add these three lines:

```
export DEVKITPRO=/home/yourUsername/devkitPro  #this will need to change if your
devkitPro folder is elsewhere. Also, replace "yourUsername" with your actual
username.
```

```
export DEVKITARM=$DEVKITPRO/devkitARM
export PAPATH=$DEVKITPRO/PAlib/lib
```

Feel free to adapt these lines according to your own tree view. Apply the changes with this command (only gotta do this once):

```
$ source ~/.bashrc
```

Check if everything went fine with the 'env' command (it lists every system variable). The following commands will help you to find the new variables:

```
$ env | grep DEVKITPRO
$ env | grep DEVKITARM
$ env | grep PAPATH
```

Note that if you run any of these and just get another $ prompt then the variable wasn't created. If this is the case go back and check your .bashrc file to check spelling. Also know that you won't see $HOME in the print out of 'env' but your actual home directory path. Example:

```
$ env | grep DEVKITPRO
DEVKITPRO=/home/<login>/devkitpro
$
```

The Make command

Right, you need to know if you can actually use PAlib. In order to do so, open up a new terminal. (get away from that mouse!) You will need to compile in peace so check this: type in "make", here, right away. If it returns you something like:

```
bash: make : unknown command
```

Then you've got a problem. However, if it returns anything else, congratulations, you are good to go! If you get this error, just open your favorite package manager (synaptic, aptitude or use apt-get...) and search for the package named "make". (If you're not on Ubuntu find the package that relates to your distro) Install it, the manager will take care of any dependencies. Now your terminal must return:

```
$ make
make: *** Pas de cibles spécifiées et aucun makefile n'a été trouvé. Arrêt.
```

Or for an English OS:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

Testing our new environment

We will now test our brand new installation. Go in the /PAlib/examples/Text/HelloWorld directory, for a good example to test first. Once you are there, just type:

```
$ make
```

If the compilation ends on:

```
Nintendo DS rom tool 1.30 - Jul 24 2006 06:34:31 by Rafael Vuijk (aka
DarkFader)built ... Keyboard.ds.gba
dsbuild 1.21 - Jul 24 2006
using default loader
```

then the job is done successfully. You have a fully functional PAlib! Congratulations!

If you get any weird errors check the forums to see if it's already been encountered and fixed. Otherwise if you get something like the following:

```
$ make
main.c
In file included from /home/ryan/devkitPro/PAlib/include/nds/arm9/PA_Sound.h:15,
                 from /home/ryan/devkitPro/PAlib/include/nds/PA9.h:37,
                 from
/home/ryan/devkitPro/PAlibExamples/Input/Keyboard/KeyboardCustom/source/main.c:2:
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_FS.h: In function 'PA_FSInit':
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_FS.h:66: error: 'REG_EXMEMCNT'
undeclared (first use in this function)
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_FS.h:66: error: (Each undeclared
identifier is reported only once
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_FS.h:66: error: for each function it
appears in.)
In file included from /home/ryan/devkitPro/PAlib/include/nds/PA9.h:42,
                 from
/home/ryan/devkitPro/PAlibExamples/Input/Keyboard/KeyboardCustom/source/main.c:2:
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_Motion.h: In function
'PA_MotionInit':
/home/ryan/devkitPro/PAlib/include/nds/arm9/PA_Motion.h:23: warning: implicit
declaration of function 'motion_init'
make[1]: *** [main.o] Error 1
make: *** [build] Error 2
```

...then you somehow managed to screw up your libnds package and should start over from the unpacking process, taking extra care not to mess up anything.

Linux Tools

DeSmuMe is an open source emulator and have a linux version which is included in the PAlib_080823_Archive.zip you downloaded already.

To compile it, let's just the same like previous in the extracted folder :

```
$ ./configure
$ make
$ sudo make install
```

You can also use the Dualis or NO$GBA windows executables with Wine. Install Wine using your package manager and then run the desired emulator through Wine.

```
$ wine Dualis.exe
etc...
```

You can use PAGfx in Linux (yes, yes, it IS an .exe that you can run in Linux) provided you have

Mono installed. Mono is a .NET runtime and is most likely in your package manager. Go on take a look.... You will also need to compile the Linux version of the tool since the binary PAGfx provided with PAlib is coded for the Microsoft.NET and includes code that won't run on Mono. You can find some binaries on the PAlib forums if you don't want to go through the trouble of compiling it yourself but I think you'll find it's quite easy to do ;) This is an easy process, so don't worry. All you need to do is install the Mono-GMCS compiler which, in Ubuntu, is as easy as typing:

```
$ sudo apt-get install mono-gmcs
```

After this navigate your way to the /tools/pagfx-linux/ directory in your PAPATH and type the following commands:

```
$ ./configure
$ make
```

Now, if you'd like to be able to run PAGfx.exe anywhere you can also run:

```
$ make install
```

but that's a matter of preference. If you don't want it globally accessible you can just copy the EXE out of the archive in /PAGfx/PAGfx/bin/Release. You'll find that it's already set to executable and you can run it right then and there. If this doesn't work you can always prefix it with 'mono ':

```
$ ./PAGfx.exe
or...
$ mono PAGfx.exe
```

The EFS_Lib archive in "Other Libs" of the PAlib archive you downloaded comes with a Linux version of the EFS builder.

Alright we did all the work of setting it up for you, now go make us some games, damnit!

I have created an executable file for building the nds files on linux. You can find it here.
-Prob_Caboose

FAQ - Installing and Compiling

Frequently Asked Questions - Installing and Compiling

Mac OS X

Mac section added 12/19/2006 by http://v8media.com, using Mac OS X 10.4.8

Install Apple's Developer Tools (Xcode)

Any recent machine will come with the Developer Tools on the installer discs, or sometimes as a separate disc. If you just bought your machine, install off the disc.

If you didn't just buy your computer, it's best to look to Apple's web site to get the most recent developer tools. The place to go for this is: http://developer.apple.com/tools/. It involves registration, but no money. Look around while you are in there, as there are some useful tools for all sorts of things you might do when scripting, programming, or even creating video on the Mac.

Once you have this installed you should have a folder at the root of your hard drive called "Developer". You will also have many command line tools added. Most importantly, "make".

Set Up Your Mac's Environment Paths

There seem to be many ways to do this, and it seems to depend partly on which "shell" you have installed or selected I use the Mac OS X default for 10.4, which is bash. To find out which shell you use, open the Terminal program (from Applications/Utilities) and then select the "Terminal" menu item and select "Window Settings". Make sure the top drop-down item "Shell" is selected. In that screen, you should see the 2nd line saying Shell: . If it's bash, here's how to set your "environment paths".

If you are using something other than bash, you either know what you are doing, or will need to find a tutorial to set up your environent paths for your shell.

Environment Paths are just paths in your operating system that you or a program define to let yourself or a program do something more easily. For example, say that everyday you have to log in to another machine using the terminal using ssh to check a university email. The first step would be to log in to the server, probably using a program called ssh. Here's what a command for that might look like: "ssh -i firstname.lastname@server.university.com". If you are doing this from your own machine, you can simplify this by making what is called an alias. your alias for this example might simply be "unimail" or something of the sort. Far easier to remember. We do this in a hidden file in your home directory called .bash_profile.

1) To create the file, or open the file if it already exists, open a new terminal window. 2) type in (without the quotes) "pico ~/.bash_profile". Now you should see a blank screen and some text a the bottom starting with: Get Help, Exit, WriteOut . . . You are in a program called Pico. All control is via the keyboard, your mouse won't help you here. 3) copy and paste these next two lines:

```
export DEVKITARM=/opt/local/devkitPro/devkitARM
export DEVKITPRO=/opt/local/devkitPro
export PAPATH=/opt/local/devkitPro/PAlib
```

4) Hold the "control" key, and then press the "x" key, this sends the Exit command to the program. It asks if you want to save, so press the "y" button. It then comes up with a screen asking what the name of the document should be, we gave it a name when we made the command to open Pico, so just hit the "return" key.

5) Either close this terminal window or type `source ~/.bash_profile` so that your changes will take effect in it.

6) There is no step 6.

Install DevKitARM, libnds, and dswifi

Disclaimer: I have not gotten a dswifi app to compile yet, I'm filling in what I know how to do so that someone who knows this part well will have an easy time of updating this with the correct information. Thanks ahead of time to whoever fills this information in!

Download the current version of devkitARM (It was at 21 as of 11/25/2007). This includes devkitARM and libnds. Here's the link: http://sourceforge.net/project/showfiles.php?group_id=114505&package_id=124207&release_id=549080. This is processor specfic. If you have an Apple PPC machine, download the osx version, if you have an Intel machine, download the Intel version. Both contain double-clickable package files. Go ahead and install these.

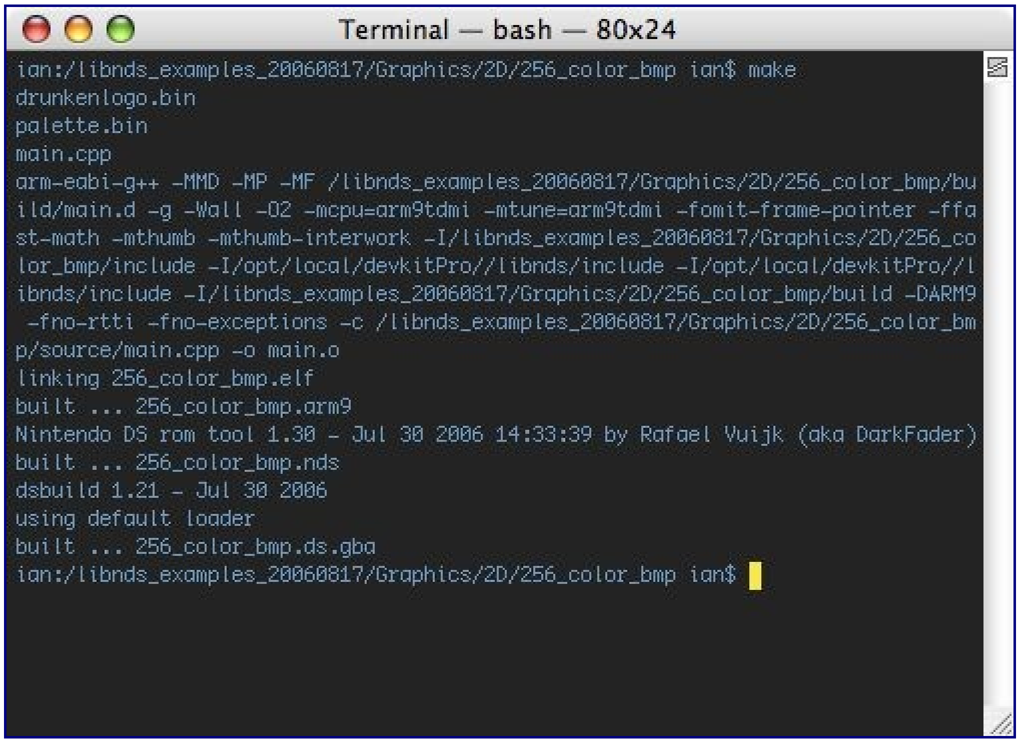
There are also examples listed "nds examples 20060817". Download those, or a newer version if one exists. Now we start getting to issues. It seems like the libs we just installed aren't fully aware of the capabilities of the OS, and don't seem to be able to deal with spaces in the names of files. As long as you put the example files and any others that you are working on directly into the "Users/MyUserName/Documents" or in a folder with no spaces and with project names with no spaces, you will have no issues. I found it easier to put the stuff in a folder in the root of my hard drive. So I have "ianDS", and "libnds_examples_20060817" as two folders in the root of my main drive.

After you place these files, we can test out some compiling. Here's a step-by-step easy way to do this.

1) Have one Terminal window open, and one Finder window where you can see the example file you want to try compiling. I'm using this file: /libnds_examples_20060817/Graphics/2D/256_color_bmp/. What you want to be able to see is the draggable folder name of the project.

2) Type "cd " into the terminal. That's a space after c, and d. 3) Drag the folder of the project you want to work on to the Terminal and let it go. It copied the whole path in there so you don't have to do it your self or find your way there using shell commands. 4) type "make"

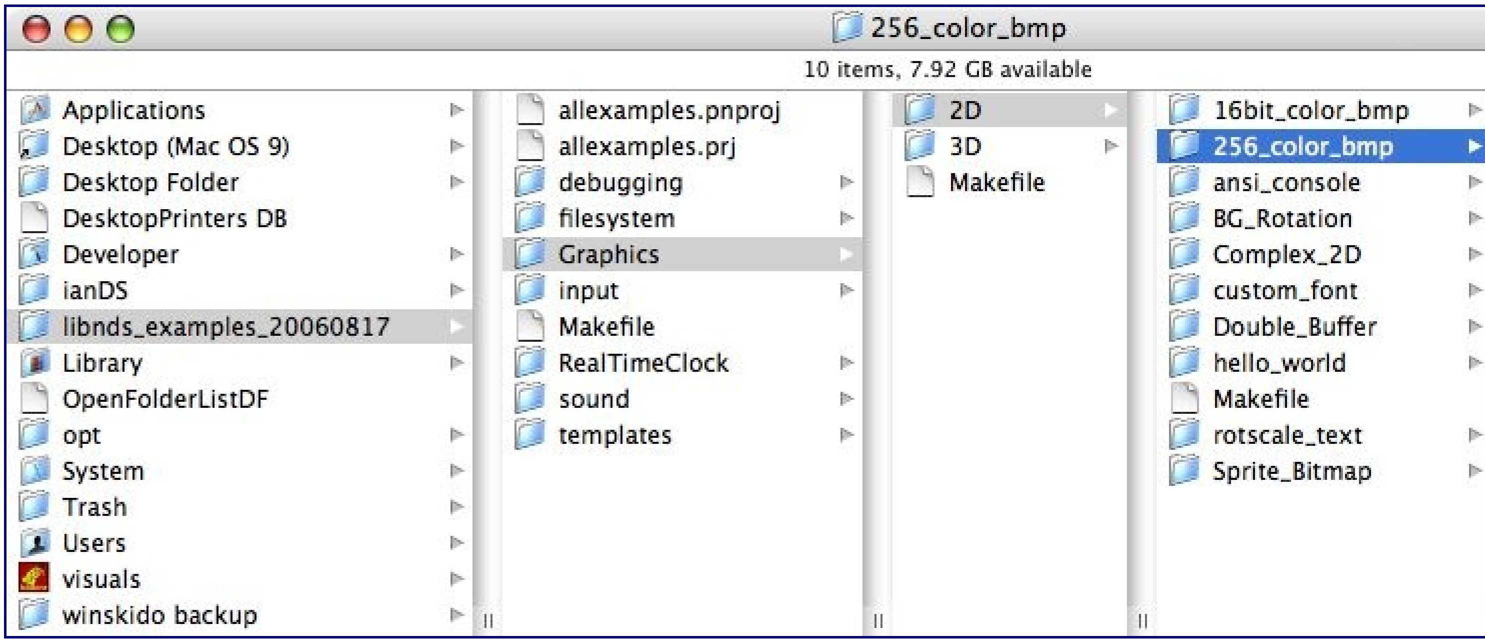
You should see something like this in your terminal window:



```
ian:/libnds_examples_20060817/Graphics/2D/256_color_bmp ian$ make
drunkenlogo.bin
palette.bin
main.cpp
arm-eabi-g++ -MMD -MP -MF /libnds_examples_20060817/Graphics/2D/256_color_bmp/bu
ild/main.d -g -Wall -O2 -mcpu=arm9tdmi -mtune=arm9tdmi -fomit-frame-pointer -ffa
st-math -mthumb -mthumb-interwork -I/libnds_examples_20060817/Graphics/2D/256_co
lor_bmp/include -I/opt/local/devkitPro//libnds/include -I/opt/local/devkitPro//l
ibnds/include -I/libnds_examples_20060817/Graphics/2D/256_color_bmp/build -DARM9
 -fno-rtti -fno-exceptions -c /libnds_examples_20060817/Graphics/2D/256_color_bm
p/source/main.cpp -o main.o
linking 256_color_bmp.elf
built ... 256_color_bmp.arm9
Nintendo DS rom tool 1.30 - Jul 30 2006 14:33:39 by Rafael Vuijk (aka DarkFader)
built ... 256_color_bmp.nds
dsbuild 1.21 - Jul 30 2006
using default loader
built ... 256_color_bmp.ds.gba
ian:/libnds_examples_20060817/Graphics/2D/256_color_bmp ian$
```

The important part there is where it says “built” near the bottom. As opposed to “Error”.

You should also see that your folder filled up a bit so it looks like this (if you are using column mode):



Woohoo! If you see this, you’ve just compiled multiple copies of a program. These can run on your flashed Nintendo DS or in emulators. For instructions on specific cards, look here, which is back up the page in the Windows section Putting the homebrew on the DS. If you are going to be doing much coding, I suggest using an emulator most of the time as this will be far faster than swapping cards, moving files, and unplugging and plugging cables every time you compile. Maybe just toss it onto actual hardware when you get something that you wonder how it looks for real or is complex enough that it may work differently than on an emulator.

One other thing you can do here while you still have the Terminal window open: type “make clean” and hit return. This should get rid of the files that were created during your “make” expedition. This can help you make sure that any errors you might get were not from extra files in your project.

PAlib

Next up is PAlib. http://palib.info/forum/modules/PDdownloads/. Take the latest stable release, in Zip format. Unzip it, and move the PAlib folder into `/opt/local/devkitPRO` alongside the other folders there. You may also need to replace `/opt/local/devkitPRO/libnds` with the version included with PAlib, but I would reccommend not doing this unless you get errors building PAlib examples **ever**. The rest of the folders and files in PAlib are examples and documentation, so they can go wherever you please.

Note: If you are having problems with testing, check that PA_Makefile is in the PALib directory (it may be in the PALib/lib directory). If it isn’t, move all the files from PALib/lib to PALib.

Testing PAlib

To test that everything has been installed properly, fire up Terminal.app and cd to the location where you installed PAlibExamples. Next type the following commands:

```
cd Demos/Bunny
make clean
make
```

With luck, you should see something like this after the last command:

```
arm-eabi-g++ -g -mthumb-interwork -mno-fpu -L/usr/local/devkitPRO/PAlib/lib
-specs=ds_arm9.specs main.o -L/usr/local/devkitPRO/PAlib/lib -lpa9
-L/usr/local/devkitPRO/libnds/lib -lfat -lnds9 -ldswifi9 -o build.elf
Nintendo DS rom tool 1.33 - Jan 31 2007 19:37:59
by Rafael Vuijk, Dave Murphy,  Alexei Karpenko
built ... Bunny.ds.gba
dsbuild 1.21 - Jan 31 2007
using default loader
```

Other Software

A piece of software that can be quite helpful on newer Apple Intel machines is WINE. WINE stands for WINE Is Not an Emulator. Since the newer Apple machines already have Intel chips, you don’t need an emulator to be able to run Windows programs, what you need is an environment to run those programs. Download from here: http://darwine.opendarwin.org/download.php. That’s an easy download and install, but you do need to have X11 installed, which you can find here: http://www.apple.com/downloads/macosx/apple/x11formacosx.html.

Some programs will need a Windows DLL file in order to run. If this is the case, you’ve got to find the DLL, maybe copy it from a Windows machine you have access to. Once you have it, just put it in the same folder with the program that you are trying to run. Start off simple and try with the sample programs that come with Darwine before you try something else.

Since the only programs you really need to run on the Windows side of things will most likely be the graphics conversion programs, being able to open them in a window instead of in a whole operating system is ideal. The program that comes with PAlib seems to work ok. You should find it here on your computer after installing PAlib: "/opt/local/devkitPro/PAlib/Tools/PAGfx/PAGC Frontend.exe".

If you are running a less recent Apple machine with some sort of PPC chip (G3-G5) what you will want is a copy of QEMU. The Mac OS X version is called just "Q". You can find this here: http://www.kju-app.org/kju/. If you are familiar with Virtual PC, you will find this to be pretty similar. Including that you will need a copy of Windows 98, 2000, or XP in order to use it.

Unfortunately, I have yet to get Darwine to run PAGfx, so I am currently using Parallels, which is a paid product. But worth not having to boot into Windows if all you are using is some tiny utility program. http://www.parallels.com/en/products/workstation/mac/. If you get Darwine to run some of these utilities, please post your success.

Another piece of useful software is GraphicConverter from LemkeSoft http://www.lemkesoft.com/. You can apparently use this to convert images to the format for the GBA and DS.

Here is how to do the preliminary steps: Make sure you have the most recent version of GraphicConverter. If you don't it might not work. After you check (or download it), open your file in GraphicConverter. Simply save the file as a "Byte-Array-Header" file.

After you do this, you can take your ".h" file and put it into a piece of software called hConvert http://www.patatersoft.info/hconvert.html. hConvert will take your .h file and turn it into a .bin file.

See the site for more info: http://www.patatersoft.info/manual.html

With the release of PAGfx-Linux, MacOSX users now have the oportunity to use the PAGfx command line version without a problem. All you have to do is to install 'mono' (http://www.mono-project.com/Main_Page), automake 1.10 and autoconf. Autoconf can be installed using apt-get tool (very useful, install it if you haven't yet), but automake has to be installed manually (apt-get only finds up to 1.9 version), however the instructions can be found easily googleing a bit. Then follow the instructions on the README file in PAGfx-Linux folder to install PAGfx. Once everything is installed you only have to put all your graphics on one folder, create the PAGfx.ini file (see sprites tutorial) and go to that folder on the terminal and type 'pagfx'.

Wireless Setup

You will be glad to see that Apple's Airport cards can be used to talk directly with the DS. Superb instructions here: http://www.macosxhints.com/article.php?story=20051119155606277.

Introduction to DS Programming

This introduction will cover the basic homebrew development toolchain as well as some of the most commonly used add-on libraries. It also includes a short discussion of available emulators and hardware (flash cards) for testing your projects.

Homebrew DS Development

Thanks to the excellent work of many coders in the DS development community, the Nintendo DS is certainly one of the easiest video game systems for homebrew developers to jump into. With the free development tools these coders have made available, even beginning coders can produce high quality games in a surprisingly short period of time. But remember, even with the best tools available (and I consider devkitARM and PAlib to be among the best for any system), producing a high qulaity game requires a lot of persistence and attention to detail.

The foundation that makes makes C/C++ development possible on the DS is the free devkitARM toolchain from devkitPro, including the libnds library. DevkitARM provides basic C libraries and a compiler targetted to the DS's ARM architecture while libnds includes many functions and macros for accessing the DS's specific hardware features.

PAlib adds to the devkitARM toolchain by giving DS developers a higher level entry point to the hardware, allowing them to focus on producing code rather than deciphering the complexities of the DS's hardware layout. PAlib is primarily intended to aide the development of original games on the DS. The bulk of its features are designed to make the end user interface better and easier to program. Graphics, sound (thanks to ASlib, which is incuded as part of PAlib) and user input are PAlib's strong points.

In addition to devkitARM, libnds and PAlib, many other libraries are available to make various aspects of homebrew DS development easier.

Nintendo DS Emulators

There are currently only 5 emulators. I highly recommend using exclusively No$GBA, it's the most accurate one.

- **No$GBA** contains almost full NDS emulation support including 3D and WIFI emulation. Runs a lot of commercial games. It may look as if this emulator corrupts the colors of your game, but it only simulates the display of the DS. Sadly, this isn't mentioned at the official homepage and often asked in the forums, so here's how to turn this off if you don't want it: Select Options→Emulation Setup and set GBA mode to "VGA (poppy bright)". Then select OK and Options→Save Options.
- Dualis, it works pretty well for homebrew, but isn't as accurate as No$GBA. Some code will run well on it, while it won't on DS !
- iDeaS is the other great emulator to watch ! It was the first to support 3d and sound, and... well, still remains the only one to really do it. The last version, 1.0.0.8c is out there, and we've seen screen shots of the WIP versions which led us to believe the next version will really rock, but still nothing comes...
- DeSmuMe is maybe the fastest emulator for DS. Even if it still doesn't support sound and zooming/rotation, it's worth a try. The original author of DeSmuME, YopYop156, has stopped working on it on the beginning of april 2006 and passed the torch to a new team of developers

on SourceForge.

- The last emulator is DSemu, and will probably do a great comeback with its new plug in features, but I don't use it yet. Maybe later

Ensata is another emulator out there. It's a crack/leak of an internal Nintendo emulator included with the official SDK. Due to the lack of legality, we don't use this.

Putting the homebrew on the DS

What you'll need first is something to actually hold all the homebrew programs that you have created on the DS, this is done by purchasing and using a "flash cart", most flash carts out there are for the GBA but now have been made to work for the DS. You can purchase many types and brands of flash carts, some with internal memory, some without. I'll review the most popular ones, although remember that you may not want to purchase these mentioned, or even from the store which I'll link to, but you will have to buy one if you ever intend to run your homebrew on an actual Nintendo DS. You'll also need a PassMe of some sort (if you get a slot 2 card), which I'll get to later. (Reader sidenote: Slot 1 cards are much cheaper and much easier to use then slot 2 cards, and don't require the use of a PassMe with the downfall that they can't directly use GBA games, but can with a relatively cheap expansion slot 2 card like the EZ-Flash 3in1)

- EFA Linker: In store description: "The EFA has become known as one of the markets most reliable GBA Flash Linkers to this day. Across the world, EFA Flash Cartridges are being chosen over the competition for their reliability, performance, and ease of use." These little bad boys have been around for quite some time and indeed do have some reliability attached to their name. They come in 256 megabit, 512 megabit, and 1 gigabit.
- EZ Flash: Mollusk's flash cart of choice. Careful, when flashing the rom, flash the .ds.gba (NOT the .nds), and *WITHOUT* the loader, or else it won't work... The in-store description goes a bit like this: "The EZ-Flash III is one of the latest flash kits with many advanced features that are not only for gamers. It has a built-in, PDA-like, icon based OS (operating system), and EZ-Disk function (works as a USB flash disk). The EZ-Flash III is also NDS, GBA(SP), and GBM compatible!". These EZ Flash IIIs come in 1 gigabit and 2 gigabit sizes respectively. There is also the EZ Flash and the EZ Flash II out for sale, please check around for stores that sell that product if you are interested.
- SuperCard: The Supercard isn't much like the EFA Linker or the EZ Flash. The key difference here is that the medium for storing DS games is actually your own CF or SD memory cards. This could save you quite a bit of money if you have a few of them laying around from your digital camera. The SuperCard is made to work with the SuperPass, which I'll explain later.
- M3 Perfect: While being the new kid on the block, the M3 Perfect would seem to work quite well in its infancy. Made out of China, the M3 is being marketed as a movie player but may do wonders for running homebrew. Many updates are released often out of their site M3adapter.com. The M3 Perfect available now works with CF cards and SD one.
- G6: From the makers of the M3 is the G6. These makers must love their product numbering. Unlike the SuperCard or the M3, the G6 has three variations, each having its own internal memory for storing homebrew files. The sizes come in 1 gigabit, 2 gigabit, and 4 gigabit.

Now, no matter what you do, you will always need to use one of these next methods to run your DS homebrew programs. It comes in just five variations, although the first three seem to be the most widely accepted.

- PassMe : That's what I currently use, and love it ! It's basically a big DS cart that goes in the DS slot. You put a real DS cart in it, which allows the passme to go around the DS's protections, and it uses the cart's save state too. You need a gba flash cart (I have an EZ Flash, for example) on which will be stored the rom. With newer versions of the DS however, a different PassMe is needed and thats where PassMe2/Passkey2/SuperPass is invented.
- FlashMe : Never tried it, but here's what it seems to be... Using a PassMe or WiFiMe, just once, you flash your DS (hacks its firmware), and then it works like the passme, with a gba flash cart, except that you don't need the PassMe anymore ! That cool, because if some friends have gba flash carts, you just flash their DS and they'll be able to have DS homebrew with nothing more. It also has the benefit of containing recovery code to protect your DS from malicious code.
- NoPass : After the DS cart encryption was cracked, commercial companies began producing a product that didn't require an original DS game inserted like the PassMe. This device works on every DS currently on the market and does not have any special requirements. Most work with any GBA (slot-2) media but some only work with the product they ship with, so make sure before purchasing one that it will work with your media.
- WiFiMe : If I got things right (thanks Aquarius :p), WiFiMe is an application you send through wifi, that boots the code from the gba cart... So basically, it's a virtual PassMe, using wifi. Can be used to boot FlashMe from flash cart or a GbaMoviePlayer. This method is obsolete as it only worked on early versions (through October of 2005) of the DS.
- Wireless Multi Boos (WMB) : Last method, this one works only for roms smaller than 4MB. It just sends the demo through wifi directly on the DS, but your ds should be able to run unsigned code (I used the FlashMe replacement firmware). But it has this size limit...
- DS-X : The DS Extreme is a DS 1 slot cartridge.You just connect the DS-X to your PC/MAC via USB (it has a tiny Mini-Usb slot on the cart) and just drag n drop your compiled NDS files. The easiest option so far. Currently supports a 512MB (4Gb) or 2048MB (16Gb) storage capacity. This device is one of the most expensive slot 1 devices compared to the M3 DS Simply/R4.
- M3 DS Simply/R4 : The M3 DS Simply and R4 are pretty much the same thing, aside from firmware differences. A quick flash can convert one to the other, as of June 2007. It is similar to the DS-X in that it is a slot 1 cartridge. Instead of using built-in storage, though, it accepts microSD cards. This drastically reduces the price to ~$40USD. The M3 DS Simply/R4 also include a PassMe2, in case you want to boot any slot 2 devices. These two devices have the largest fanbase by far.
- Cyclods : The Cyclods is a cart very similar to the M3 Simply and R4 with the exception that it is newer and at the moment (October/07) it is the flashcart with the most up to date flashcart.
- EZ-Flash V : The EZ-Flash V is the Slot-1 version of the EZ-Flash afore mentioned.
- Acekard RPG : An open source slot one solution with both a MicroSD slot and 1 Gigabite of internal memory.

It would be best to keep your flash cart and passme-device from the same maker to ensure compatibility so combine as follows: EFA Linker and EZ Flash with PassMe2, SuperCard with SuperPass/SuperKey, and M3 or G6 with PassKey2/PassCard 3.

— *Micheal Bach 30/11/2005 09:18*

Put the correct file into your flashcart

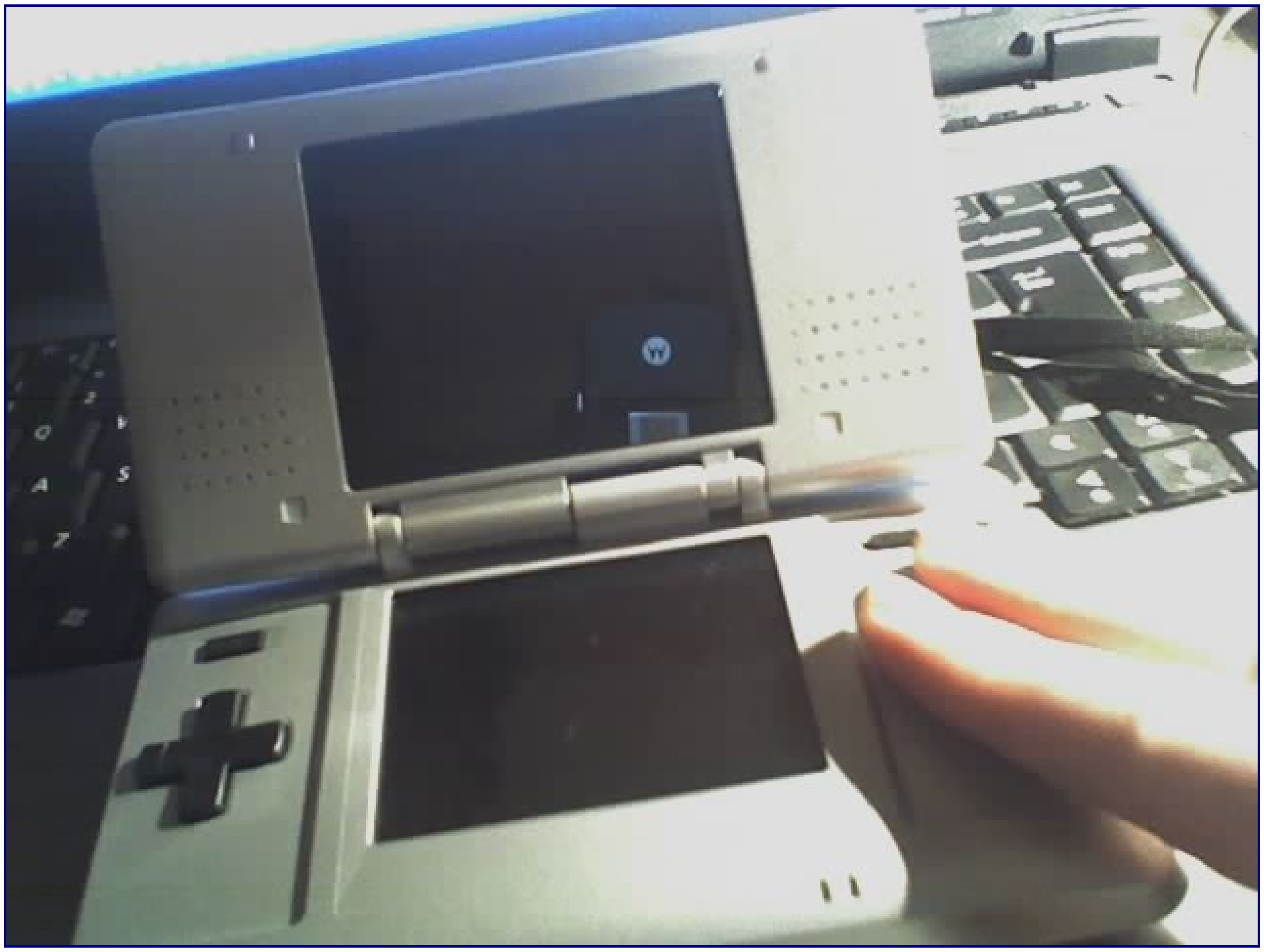
as soon as you compile your program/PAlibExample/homebrew, “magically” three files appears in your program root directory:

- <program_name>.ds.gba –> use this on GBA Flash Carts, and EZ (Some may require you rename it to .ds.nds)
- <program_name>.nds –> this is for Slot1 carts and most emulators, Chishm’d GBA Movie Player V2 and G6 and M3 (you must push START instead of A to load it).
- <program_name>.sc.nds –> this is for supercard, G6 and M3

just pick the file you need and put on your flashcart! well done! — *Lotti 01/09/2006 20:00*

Using the SuperCard with SuperPass
If you are using the SuperCard with a SuperPass (as shown in the picture below), the homebrew application you’ve created should be easy enough to be selected from the menu, as soon as you switch your DS on, a menu should appear on the top screen.

Using the SuperCard with a flashed DS
If you have flashed the DS Bios with the FlashMe method, after you have inserted the SuperCard into the slot, as soon as the DS is switched on, keep pressed the buttons A + B + X + Y, the SuperCard menu should now show up, select the file named “mygame.sc.nds” (remeber?) and launch it pressing A, if everything went well, your program should come up on the DS. Well Done.



BOOT_SC_EXFS_CF.nds
helloworld.nds
A/B FileOpen/FileClose
START ShowDebugLog
PlayControl...
Volume x100 MusicNextRandom
X/Y btn ChangeVolume
www.SUPERCARD.CN

<back to top>

<day2>

Day 2 - C/C++ Learning

Introduction

This tutorial will focus on the very basics, with more emphasis on C as it applies alrgely to C++ as well. Please refer to the 'External Links' and 'See Also' sections for links to further information relevant to the tutorial as well as some more advanced topics that you may find interesting. Please also note that though this tutorial is modeled in the form of one day's work, it may be in your best interest to spread it over several days. This tutorial will depend on libnds directly, as opposed to indirectly through PALib, this allows us to focus on C/C++ as opposed to the PALib API which the other tutorials cover; These are in effect, just normal progams.

Who this tutorial is for

This tutorial was written with the assumption that you have some programming experience, namely with concepts such as variables and functions(which you may also learn in mathematics). The aim is to make the tutorial in an incremental manner, therefore you can focus on one concept then build on it; As a side-effect, some more concepts may be introduced early on, then go into more detail later. As always try hard, programming is not easy, and if you need help or have any comments on the tutorial, make a post on the forum or correct it here.

Disclaimer

- This is not a comprehensive C/C++ tutorial(see external links or search the web).
- Some concepts are overly simplified, which may lead to confusion if you attempt to go too fast.
- This is not a C++ tutorial, it's a C tutorial that applies to C++.
- This is not a PALib tutorial.

Variables

At the very basic level, a variable is just a label for some value e.g. spritenumber, or spriteposition.

Declaration

In order to use a variable, it must first be declared. That is, the compiler (a program that translates your human-readable source-code into a form suitable for execution by a computer) is made aware of it. The compiler needs at least a variable type in order to determine how much storage to allocate for the variable, this information is also used to validate your uses of the variables as they are not always compatible with each other.

Types

At the lowest level, C supports only one type of variable, numbers. A step up, we begin to see abstractions of this ‘number’ into two categories, integers, i.e whole numbers and floats(those with a floating/decimal point) Within these two groups, we may categorize further, into the cocnept of **Signed** variables, those with a sign(negative numbers), It follows that, **Unsigned** variables are positive.

The mathetmatical concepts apply here; If the number is prefixed with a ‘-’(minus, dash, hyphen, ...) it is negative, otherwise it is implicitly positive when not prefixed, or explicitly positive if prefixed with a ‘+’ (plus); The latter method is optional, and is often less readable.

The signed/unsigned distinction typically applies only to integral(integer based) types.
Notice that float variables are **much** more CPU intensive on the NDS, so seek alternative methods where applicable, either integers or fixed point math

The fundamental types are further categorized by their sizes; Typical:

Integral Types

Type	Size	Unsigned	Signed
char	8 bits	0 → 255	-128 → 127
short	16 bits	0 → 65 535	-32 768 → 32 767
long	32 bits	0 → 4 294 967 295	-2 147 483 648 → 2 147 483 647

type *int*, is a special, and is typically depends on the archiecture. i.e 16 bits on systems with 16 bit CPUs and 32 bit on those with 32 bit CPUs(NDS). if an absolute range is required, for protability reason, use either short or long.

type *char* is also special in that, depending on the compiler it is signed or unsigned, so care must be taken when using them. If in doubt over which integral type to use, choose *int*.
libnds (and concequently PALib) provides clear names are these types.
e.g u8 for and unsigned 8 bit integer and s32 for a signed 32 bit integer.

Floats
(provide common ranges for float values as well)

Type	Size
float	32 bits
double	64 bits

Declaring a variable

Declaring a variable is actually very easy, it takes one line of code and is easy to remember, lets declare a float with the name “hello”:

```
float hello;
```

now that wasn’t hard huh?
float stands for the variable type.
hello is the name of the variable
the **semicolon** (’;’) tells the code that the line ends there, if would forget it the code will think the code will go on to the second line as if it was one.

now let’s declare a signed, 8 bit variable to lets say, store the age of the user

```
s8 age;
```

again, s8 is the variable type.
age is the name of the variable
and we end the line with a semicolon
I hope you get the idea now.

Here’s a quick view of the possible values each type has, depending on if it’s signed or not:

for example, if you want to store the amount of gold a player has, you would choose an unsigned 32 bit variable, as you can’t have a negative amount of gold, and would give the player a maximum of 4294967295 gold, plenty for the average homebrew game right?

Booleans

now lets talk about booleans, booleans are great things to know, you can see them like a switch, if you flip the switch one way the light turns out, and if you flip it the other way the light turns on. The best thing is that they are just as easy to use as any other variable! to declare a boolean use the following code:

```
bool NAME;
Usage
```

To use a variable, once it is declared, is easy, you just put its name where you want to use it. Here’s an example for the standard programming example **Hello World**:

```
//Include PAlib
#include <PA9.h>

//Declare 2 variables, signed 32 bits
s32 HorizontalTile = 0;
s32 VerticalTile = 0;

int main(void)
{

        //PAlib Initialization
        PA_Init();
        PA_InitVBL();

        //Initialize the text system on the top screen (1), background 2
        PA_InitText(1,2);

        //Print the text on the top screen, with coordinates 0,0 (very top left)
        //Instead of putting the coordinates 0,0 we put the variables
        //Which have been set to 0,0 at the top of the file
        //We could change their value now before the text is drawn for different
results (see next example)
        PA_OutputSimpleText(1,HorizontalTile ,VerticalTile ,"Hello World !");

        while(1)
        {
                //Loop forever
        }

        return 0;
}
```

Giving your variable a value

You can see in this example we're giving the variable a value when we declare it:

```
s32 HorizontalTile = 0;
```

But it's also acceptable to do something like this too:

```
s32 HorizontalTile;
HorizontalTile = 0;
```

Note: You have to assign a value to a variable before you can start using it, otherwise you'll get an error

Global and Local

Another thing to know about variables, which we haven't talked about... These can be either *global* or *local* variable... What's the difference ??

- A *global* variable is accessible to all functions in the file... It is therefor created outside any function. Notice how *s32 HorizontalTile = 0;* and *s32 VerticalTile = 0;* are outside of main. These are global variables, variables available to every part of the program.
- A *local* variable, on the contrary, is accessible only in a given function. It is created when the function starts, and disappears when the function ends... Here's an example :

```
void MyFunction(void)
{
   s32 variable = 0;
}
```

This variable is accessible ONLY in that function, and from no where else. Each time the function is called, it is recreated and reset...

In the end... if at any point you hesitate between using a local or global variable... you should probably use a local variable. As it is possible to pass a variable from one function to another (use &variable in the function declaration), global variables are almost always unnecessary and considered poor coding.

Why are global variables bad? Because anything can edit the information contained in them and if even one function messes with your variable that shouldn't, you can get some very wierd and _very_ hard to track down bugs. You have been warned.

Operations

You must admit that variables would be pretty useless if you couldn't do anything on them. You can, of course, use the basic operations (we won't cover much more). The 4 base operations are + - * and / (addition, subtraction, multiplication, and division). Be aware (like Jean-Claude Van Dam) that divisions are really slow... To give a variable a new value, you use the '=' symbol : variable = value;

There is another function besides +,-,*, and /! If you use %, this gives the remainder of a division. Posted by -Kirby64-(Now known as SuperKirby)

Yes, this is called “Modulus” and is essential with, among other things, the fizzbuzz program. Posted by Demosthenes2k8

Please take care to note, that mod(%) in c/c++ does not wrap-around as you’d expect on negative value. e.g

```
-2 % 10
```

does not result in 8, but -2; Therefore you must add 10 snippet(are we allowed to write functions in here(so early?))

```
if (lv < 0) {
        result = ((lv % rv) + rv);
} else {
        result = (lv % rv);
}
```

where lv is in place of -2 and rv is in place of 10

Here’s an example (this is not DS code, but you don’t care)

```
float PI = 3.14;
float radius = 10;
float perimeter;

perimeter = 2*PI*radius;
```

Ok, you have to admit this can’t be much easier...

On a side note, as this info isn’t too important, here are other operation syntaxes :

```
variable += 2;
```

(2, or another variable...) means variable = variable + 2; it’s just shorter The other operations are -=, *=, and /=, of course...

One last variant is the following one :

```
variable++;
variable--;
```

++ just means += 1, and -- means -= 1... It’s just the fastest way to write it You could also put the ++ or -- as a prefix. More important is explaining the actual difference between prefix and postfix, though, because they don’t do the same thing.

```
int x=0;
std::cout << x << ++x << x++ << x;  /* this prints 0, 1, 1, 2 */
```

Prefix (++x) adds one before the expression is evaluated; postfix (x++) adds one after the expression is evaluated. That’s why you bump one up the first time in the print, but not the second time.

For example, if you do say:

```
s16 var1 = 0;
s16 var2 = 20;

var1 = var2++;
```

the value of var1 = 20 and the value of var2 = 21

But if you do:

```
s16 var1 = 0;
s16 var2 = 20;

var1 = ++var2;
```

var2 = 21 but var1 is also 21, because the ++var2 first adds one and then returns the value of var2!

Bitwise Shift

This is faster than multiplication or division, but does not give the expected values for negative numbers and the actual optimization is tiny, so until you get the grasp of some basic coding, try not to think much about it.

An example is...

```
u8 num = 2 << 2;
```

In this case, num = 8.

Using bitwise shift like this (x « y) is like doing x * 2^y. Using bitwise shift like this (x » y) is like doing x/2^y.

Conditional branches

Even though all this is cool, it’s still pretty useless... Here comes the interesting part : if, while, and for instructions!

THIS is what you were waiting for...

if

Here is an example, which is just a chunk of code, nothing compilable, but to show you...

```
if(Stylus.Held == 1)
{
    PA_OutputSimpleText(0, 0, 0, "Stylus is held !!");
}
```

As you can see, this is easy. Stylus.Held is a variable that is by default at 0, and passes at 1 if the stylus touches the DS’s screen... This simple bit of code looks at Stylus.Held, compares it to 1 (the ‘==’ command is an equality check... do NOT put ‘=’ instead, because, as we have seen, variable = 1 gives the value 1 to the variable)

Now, just so you know, this code could have been written with if (Stylus.Held), because if you don’t put an equality check, it checks if the value is different from 0 and executes the code in the brackets according to that...

As you probably guessed, ‘==’ is not the only check you can do. The others are :

- >
- >= (> or =)
- <
- ⇐ (< or =)
- != (not equal, different)

Note that putting ! in front of a boolean turns it from 0 to 1, and from anything else to 0.

Next, here is another piece of code :

```
if (Stylus.Held)
   PA_OutputSimpleText(0, 0, 0, "Stylus is held !!");
if (Stylus.Held == 0)
   PA_OutputSimpleText(0, 0, 0, "Stylus is not held !!");
```

There are a few things to comment on that one. As you can see, here, I haven’t put the brackets... If you don’t put brackets, only the instruction right after the if is executed... Second, I could have done 2 things instead of the if (Stylus.Held == 0) :

- Put if (!Stylus.Held)... Why ? because !variable returns 1 if the variable is equal to 0, and 0 if not.
- Better : else ... yes, that simple :

```
if (Stylus.Held)
   PA_OutputSimpleText(0, 0, 0, "Stylus is held !!");
else
   PA_OutputSimpleText(0, 0, 0, "Stylus is not held !!");
```

If this condition is filled, do this, if not (else), do this other thing... pretty easy ! the same way as the if, the else can work with brackets (if{...} else{...}).

```
if (condition) {
  DoThis()
} else {
  DoThat()
}
```

Another way of doing this is:

```
condition? DoThis():DoThat();
```

switch...case

I am unsure of this notion, so please, if someone notices ANY mistakes, please fix it.

Switch is used when you require a whole bunch of conditional statements. Each *case* is like an *if* statement, which checks for a value. Use *switch* on a variable or value, and it will check each case for it. *Default* occurs when none of the other cases do. Put it at the end, in case none others are true. the syntax is:

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  .
  default:
     default group of statements
}
```

Here is an example...

```
int testNum = 3;

switch (testNum)
{
    case 1: //execute if testNum is 1
        PA_OutputSimpleText(1,0,0,"Test Num is 1");
        break;

    case 2: //execute if testNum is 2
        PA_OutputSimpleText(1,0,0,"Test Num is 2");
        break;

    case 3: //execute if testNum is 3
        PA_OutputSimpleText(1,0,0,"Test Num is 3");
        break;

    default: //execute if all cases return false.
        PA_OutputSimpleText(1,0,0,"None is true");
        break;
}
```

Note: When a case condition is true, then it will start there and execute every other case. That's why you need *break*. *Break* not only breaks out of loops, but out of switch statements as well. Not inserting break will result in a *fall-through*. There are some cases, though, where you could want to leave some of the *break*s out... Here's such an example :

```
int testNum = 3;

switch(testNum)
{
    case 1: //execute if testNum is 1
        PA_OutputSimpleText(1,0,0,"Case 1 executed");

    case 2: //execute if testNum is 2
        PA_OutputSimpleText(1,0,0,"Case 2 executed");
        break;
    case 3: //execute if testNum is 3
        PA_OutputSimpleText(1,0,0,"Case 3");
        break;

    default: //execute if all cases return false.
        PA_OutputSimpleText(1,0,0,"All other cases");
        break;
}
```

In this example, if case 1 is done, as there is no break, it'll execute cases 1 AND 2... case 2 only does case 2, 3 3, and any other value does the default case... You could even have some code to do cases 1, 2, and 3, etc...

while

while

while is another interesting conditionnal thingy... It works almost like the if condition, but loops and restarts as long as that condition is true...

```
while (!Stylus.Held)
{
    PA_OutputSimpleText(0, 0, 0, "Touch the screen to continue");
    PA_WaitForVBL();
}
PA_OutputSimpleText(0, 0, 0, "Starting !");
```

This code, which is pretty basic, is non-the-less usefull. What does it do ? Well, we can just read the code line by line... It first checks if the Stylus touchs the screen... if not (has ! in front), it does what in the loop, so writes on the screen that the stylus is pressed, then waits for the VBL (synchronises with the screen, 60 frames per second...). So, 60 times per second, it will check if the stylus touches the screen, and do so until you finally touch it ! Once you touch the screen, it stops looping on and on, and continues, writing that the program will now start...

This small code can be used as a waiting screen

do...while

A variant of the *while* keyword is do{...}while(...);

It works just the same, but instead of directly checking if it will loop, it first does once the code, then checks and loops :

```
do
{
    PA_OutputSimpleText(0, 0, 0, "Touch the screen to continue");
    PA_WaitForVBL();
}
while (!Stylus.Held);
PA_OutputSimpleText(0, 0, 0, "Starting !");
```

Here, even if the stylus was already touching the screen, it will display the touch screen message and wait 1 VBL...

When using a do{}while or a while{} loop, there's 1 thing you must really be aware of : in this loop, there must be a way of exiting... if not, the program will stay stuck in the loop forever, and will seem to hang. Here's an example of what not to do :

```
s32 variable = 1;
while (variable == 1)
{
    PA_OutputSimpleText(0, 0, 0, "Touch the screen to continue");
    PA_WaitForVBL();
}
PA_OutputSimpleText(0, 0, 0, "Starting !");
```

In this code, nothing will change the variable's value, so you will never be able to exit it !!!!

for

This last loop is in fact just a variant of the while{}, but is pretty nice. I use it all the time. Instead of just having the condition, it has 3 parts, for example :

```
s32 i;
for (i = 0; i < 120; i++){
    PA_WaitForVBL();
}
```

Now, let's analyse these 3 parts...

- The first part is the definition or initialiser, it's something to do BEFORE looping.
- The second part is the termination or condition, like the one in the while(...) or if(...)
- The third one is the incrementation - you can only put terms in there, not actual code.

Using while, you could 'translate' it this way :

```
s32 i = 0; // Initialiser
while(i < 120)  // Condition
{
    PA_WaitForVBL();
    i++;  // Incrementation
}
```

What's nice about the for command, is that, in the example I showed you, it gives you an easy way to have something done a given number of times... The first example will do the WaitForVBL 120 times before exiting, nothing more, nothing less... And what are 120 frames ? At 60 frames per second ? Well, 2 seconds !

With these notions, you already should be able to understand a little more stuff, and if you wish, you can move on to the sprite tutorial a bit before finishing this tutorial... That way you'll see a little more of DS dev, to motivate yourself

break

Sometimes, you wish to force-quit a loop. If you wish to do that, use break...

```
while(true)
{
    break;
}
```

At first glance, it may look like it loops infinitly, but *break* makes the program leave the loop immediately. You can put it in conditional statements, and it will still quit the loop.

continue

Sometimes, you wish to skip a part of a loop and go back to the condition. Then you need continue...

```
while(true)
{
    //some stuff
    continue;
    //Code which never gets executed
}
```

In this case you will never get to the *Code which never gets executed* part, because the *continue* statement jumps back to the while condition.

Functions

Here comes a rather important part of C programming : functions!

Basic Functions

Functions are pieces of code to execute.

That main you have seen earlier is oddly enough, a function. Functions only really need a name, but usually also have a return type and some arguments. Afterwards, there should be an opening brace, the code, and then a closing brace.

```
return-type functionname(arguments)
{
    //code
}
```

The *return-type* is what is returned by the function. If your function does not return anything, use *void*.

The return type can be any variable type. To return a variable, use *return*.

```
s32 getNumber()
{
    return 2;
}
```

getNumber is the function name. There are no *arguments* in this function.

It *returns* the number 2.

To use arguments, you put the variable type and a name for that variable.

```
s32 getNumber(s32 num)
{
    return num;
}
```

All these were basic functions, but you could have more complex one. Let's see if we can create a function that adds 2 given numbers, then returns it :

```
s32 addNumbers(s32 first, s32 second)
{
    return (first + second);
}
```

Here, you see that to seperate several arguments, you use a comma... To use this function in a program is simple :

```
number = addNumbers(3, 6);
```

I put 3 and 6 as arguments to pass, but I could as well have put variable1 and variable2, or combinations of numbers and variables.

Declaring a function

The only thing we haven't really talked about yet, concerning functions, is the correct way of declaring them. As you might have guessed, you can't create a new function in the middle of your main function, for example... So here's the correct way of doing so :

```
void myFunct(s32);

int main()
{
    myFunct();

    return 0;
}

void myFunct(s32 num)
{
}
```

This is just an example that doesn't do anything at all and wouldn't even compile, you don't really care. What you see first, the "void myFunct(s32 num);", is the prototype. It's just so that the other functions know your function exists, because if a function is created after the function it is called from, you get an error. Declaring functions lets you create your function later...

Then, you have your main code which you don't really care about for now.

And last, you can see that the actual code of your new function (myFunct) is after the main function, at the end of the code... It is declared exactly like the prototype, but this time around you have the brackets, and in them the code...

Arrays

You saw how to use variables, conditionnal loops, sprites, and backgrounds... All that was great, but now, we are going to see another C important part : arrays. What is an array ? An array is like a large list of similar variables.

Declaring an array is really simple. It's just like declaring a normal variable, but adding '[', the size, and ']' !

```
float MyArray[10000];
```

This declares a 10 000 long array of floats... that will immediately crash your DS! Since ten thousand floats take up about twice the size of the stack.

s16 MyArray[10000]; This works much better, since s16 uses much less space.

Anyway, now, what can arrays be used for... hmm... try to find some uses by yourself !

...

...

So, found some ?

Think about it, what if you need a list of numbers? Like 100 or more, would you make 100 variables? Ok, think about it again, what if you don´t know exactly how much numbers do you want?

This is when arrays comes to help you.

```
s32 numbers[1000]; // Table for numbers
s32 i; // Temporary variable to use with the for

for (i = 0; i < 1000; i++)
{
   numbers[i] = 0;  //Try to imagine that you are taking the number from something
else, not the same :P
}
```

Now, if you want to read its content you can do it like this.

```
s32 oneNumber = 0;

oneNumber = numbers[3];
```

This takes the number at the 4th position and copy it into the variable oneNumber.

Now I can hear you, 4th position? You are reading numbers[3]! Yes, but arrays, and pretty much everything in C starts at 0. numbers[0] it´s the first position, numbers[1] it´s the second, and so on.

Structures

Structures are something I find essential to organise your project and make it readable... They're a bit different from arrays, and can even be combined with them.

To make it simple, a structure is a group of variable under a single name. I'll make you see what's nice with it using a few examples.

Let's say I have a space ship game. In this game, my main ship (we'll call it bob) has a position (x and y variables), a horizontal and vertical speed on the screen (hspeed and vspeed), and that's about it for now. So, declaring it's variables would like :

```
float x, y, hspeed, vspeed;
```

(if several variables have the same type, you can declare them on the same line that way...) Ok, I have to admit, that's not too complicated, and wouldn't probably require structures...

Now, let's say we add an enemy, that has a position and speed exactly the same way... x and y being take, well do enemx, enemy, etc... The whole declaration becomes :

```
float x, y, hspeed, vspeed; // For the main ship
float enemx, enemy, enemhspeed, enemvspeed; // For the enemy ship
```

This is still ok, but kind of ugly... now, what happens if we have several ships ? we could do enem1x, enem2x, etc... but you have to admit it would become rather stupid and complicated...

Declaring Structures

How would this look using structures ? First, we'll create a structure using a typedef, which defines a new type (so you'll have as types float, s32, and now the structure's name)

```
typedef struct
{
    float x, y, hspeed, vspeed;
} shipinfo;
```

So this creates a new type, called shipinfo. Using it is simple : first, declaring a variable using that type, like mainship, and then using it :

```
shipinfo mainship;
mainship.x = 128;
mainship.y = 96;
...
```

As you can see, you created a new variable (mainship) which has a 'shipinfo' type. To access the different variables in it, you use the '.', like mainship.hspeed, etc...

HINT : Never forget to use "typedef" before struct : if you don't, then you will have to write :

```
struct shipinfo mainship;
```

Insteed of :

```
shipinfo mainship;
```

Array of Structures

Now, where this gets really nice, is combining structures and array. I'll post the code, and you should get it right away :

```
typedef struct
{
    float x, y, hspeed, vspeed;
} shipinfo;  // new type for ships...

shipinfo mainship;  // our main ship
shipinfo enemy[10]; // array of structures !
```

Here, you have your first array of structures ! These few lines create 10 different enemy ship informations, numbered 0 to 9, and accessible through

```
enemy[0].x = 128;
enemy[3].x = 12;
...
```

Cool !! this is way better than calling all your variables enem105x, etc... right ? But there is something else... what happens if we combine array of structures with a for loop ? This is something easy... Let's say that each frame, your enemy ships move by hpseed and vspeed (horizontally and vertically). If you wanted to move them, you would have to do

```
// same thing as enemy[0].x = enemy[0].x + enemy[0].hspeed; remember ?
enemy[0].x += enemy[0].hspeed;
enemy[0].y += enemy[0].vspeed;

enemy[1].x += enemy[1].hspeed;
enemy[1].y += enemy[1].vspeed;

enemy[2].x += enemy[2].hspeed;
enemy[2].y += enemy[2].vspeed;

enemy[3].x += enemy[3].hspeed;
enemy[3].y += enemy[3].vspeed;
```

Here, I did it only for... 4 ships (0-3), and it already takes quite a few lines... Do you imagine doing this for... 10 ships ? 100 ships ? I wouldn't !

For Loop and Structures

Using for, this is going to be much easier :

```
s32 numberofships = 10; // we have 10 ships, but we could put more
s32 i; // This is a temporary variable we will use for the for loop...
```

```
for (i = 0; i < numberofships; i++)
{
    // same thing as enemy[i].x = enemy[i].x + enemy[i].hspeed; remember ?
    enemy[i].x += enemy[i].hspeed;
    enemy[i].y += enemy[i].vspeed;
}
```

Now, isn't that sweet ? Just a few lines, and this will add the vertical and horizontal speeds to the 10 ships ! The for loop will give the values 0 to 9 (when 10 is given, it exits) to 'i', and we use that in the structure array ! If you have 100 ships, just change the numberofships to 100 and the for loop will obey

functions and Structures

Using a similar method as the 'for' one, we could also use functions... Let's do one that moves a given ship around by adding the speed to it's coordinates...

```
void MoveEnemy(s32 number)
{
    enemy[number].x += enemy[number].hspeed;
    enemy[number].y += enemy[number].vspeed;
}
```

This small function adds the speed of a given enemy to it's position.

Why is it declared as 'void' ? Because it doesn't return anything (do you see a 'return' in it ?) It executes codes on several variables, that's it... You can either use it as

```
MoveEnemy(0);

// or !

s32 i;
for (i = 0; i < 100; i++)
{
    MoveEnemy(i);
}
```

Parsing your code up like this can make it even easier to read, don't you think ? With this function, it's easier to move around one single enemy, or even all of them using the 'for' loop. Another nice thing is that if you want to add more code to the enemy movement (speed changes, position checking, or sprite position (which you have either seen already or will see soon enough)), it's easy, you just add that to the MoveEnemy function

Using Multiple Files : C, Headers

Once you start making more complicated projects, you'll need to organize your program across multiple files. To do this, you should group related parts of code into .c/.h files. For instance, say your program has a starfield, and you have a set of functions to update and render it. You'll probably want to put those in a separate file. You can organize it into a header file and source file. The header file, say stars.h, my contain something like this:

```
// in stars.h

// constants and variables
#define NUMSTARS              128

// function prototypes(the function declaration without the stuff in the { and } )
void update_star_field(int ms);

void render_star_field(int screen);

int some_helper_function(int a, int b);

// have some more stuff such as the above here
```

In the header file one would define any necessary constants as well as declare any functions that are defined in the accompanying .c file and need to be used by other parts of code. The source file would have something like this:

```
// in stars.c
#include "stars.h"

void update_star_field(int ms)
{
    int i;
    for (i = 0; i < NUMSTARS; ++i)
    {
        // some code to execute
    }
}

void render_star_field(int screen)
{
    /* more stuff here... */
}

int some_helper_function(int a, int b)
{
    // put some stuff here if you want to
    return a * b;
}
```

In the source file, we define any functions declared in the header file. Also, we can put in helper functions in the source file as well. Helper functions are just functions that may be needed by the functions in stars.c, for example, but shouldn't be called by other parts of your code. Notice that we included stars.h using quotes. This is because stars.h is a local header file, and using quotes indicates that.

Now, if another part of the code needs to use these functions, we can just include the header file.

```
// in main.c

#include <PA9.h>
#include "stars.h"
/*other includes.... */

int main()
{
   PA_Init();
   PA_InitVBL();

while(1){

   render_star_field();
   PA_WaitForVBL();
}
return 0;
}
```

— *masonium 22/02/2007 01:51*

See also

Common error messages and solutions.

External Links

Frequently asked questions in C - mostly advanced
Frequently asked questions in C++ - mostly advanced
Variables - Wikipedia
A more complete c/c++ tutorial - mostly applies to plain C as well
Thinking in C++ - book by Bruce Eckel (legally free in PDF format)

<back to top>

<day3>

Day 3 - Input/Output

The only Output system we'll talk about for now is the basic text system... Next, we'll see the different input methods : Pad, Stylus, and Keyboard etc....

Output : Text

Text is important for virtually EVERY single DS game. Imagine a game without speech or instructions. Creating your own text functions would be difficult, but that is what PAlib is for. PAlib ships with many handy text functions.

In order to initialize text, use *PA_InitText (bool screen, u8 bg_select)*

The first argument is the screen (1 = top, 0 = bottom), and the second is the background (0-3). Now, for every single text function using the same screen, it uses the same background.

Simple Text

Creating simple text is easy, or it wouldn't be called simple text. Use *PA_OutputSimpleText(bool screen, u16 x, u16 y, const char *text)*. The argument *screen* is the screen to use. *x* is the x-coordinate, in tiles(0-31). *y* is the y-coordinate, in tiles(0-23). *text* is, well, the text.

Here is some sample code...

```
void printText()
{
    PA_InitText(1,0);
    PA_SetTextCol(1,31,31,31);

    PA_OutputSimpleText(1,0,0,"This is an example");
}
```

If you want to erase some text, just overwrite it with spaces, like `PA_OutputSimpleText(0, 0, 0, " ");`

Box Text

Simple text may be useful, but often, it just doesn't cut it. Sometimes, you need wrapping text, confined to a rectangular portion of the screen. That's what *PA_BoxText()* is for. Its arguments are as followed.

- screen - the screen to use (0 or 1)
- basex - the top left corner x of the box, in tiles.
- basey - the top left corner y of the box, in tiles.
- maxx - the bottom right corner x of the box, in tiles.
- maxy - the bottom right corner y of the box, in tiles.
- text - text to output
- limit - the maximum letters to show at this time (could be used for slow-typing text).

This function returns the number of letters ACTUALLY outputed.

To have slow-typing text, just increase the limit by one. When the number returned does not change, it has stopped (either because there is no more text, or because the limit has been reached). At this point, you may stop increasing the limit.

If you have a long speech, you need to know the maximum number of characters that can be outputed. In that case, use this formula...

```
u8 maxChar = (maxx - basex)*(maxy - basey);
```

It should work.

Advanced Text

Now that you have seen how to output some basic text on the screen, we'll see what could be considered as the most important text function : *PA_OutputText(bool screen, u16 x, u16 y, const char *text, arguments...)*. As you see, it's not that different from the first simple text function, except for the arguments list, which is very important. It allows you to display the contents of variables or strings directly on the screen, which is really great for debugging !

You can take a look at the Text/Text example for some applications...

Displaying Integers

Here is some sample code displaying the Stylus's position... (you'll see how to use the stylus very soon...)

```
PA_OutputText(1,0,0,"Stylus X : %d   Stylus Y : %d", Stylus.X, Stylus.Y);
```

First thing you see, is the %d... it is used to display integers (s32...). If you put %d somewhere in the text, it'll search for an argument in the arguments you gave (Stylus.X and Stylus.Y, here). If you have %d several times, it'll take the arguments in the order of appearance... so the first '%d' is for Stylus.X, the second for Stylus.Y, etc....

Displaying Floats

Displaying floats isn't much different than displaying integers... There are just 2 things that differ :

- Use *%f* instead of *%d*
- Add how many numbers you want to be displayed after the period...

So you get something like this :

```
float test = 4.678;
PA_OutputText(1,0,0,"Float value : %f3", test);
```

See, not much harder ! This will display *Float value : 4.678*

Displaying Strings

The last thing to know is how to display strings... This is pretty useful, because you can for example integrate the player's name in your text, and thus have a personnalized message ! Just use *%s*, for string !

So you get something like this :

```
char name[10] = "Mollusk";
PA_OutputText(1,0,0,"Hi %s", name);
```

See, not much harder ! This will display *Hi Mollusk*. One important thing : your string, which is basically an array of *char*s, MUST have as last character value a 0... if not, the text output will continue until it falls on a 0, which could take some time... So if you add letters one by one in a string, with like ‘a’, then ‘b’, etc... always add *0* at the end.

Text Color

There are 3 different ways to choose a text’s color :

PA_SetTextCol

You can, at any point, change the font’s color by using a simple function : *PA_SetTextCol(screen, r, g, b)*. This can set the color as you want on 1 screen... ALL the text on that screen will have the same color, even if you wrote it before changing the color. What values are r, g, and b ??

- *r* stands for red... It’s the red quantity in the color, from 0 (no red) to 31 (maximum red).
- *g* is the same thing for green...
- *b* is for blue...

Here are a few examples :

- Blue : PA_SetTextCol(screen, 0, 0, 31)
- Red : PA_SetTextCol(screen, 31, 0, 0)
- White : PA_SetTextCol(screen, 31, 31, 31)
- Black : PA_SetTextCol(screen, 0, 0, 0)
- Grey : PA_SetTextCol(screen, 22, 22, 22)
- Magenta : PA_SetTextCol(screen, 31, 0, 31)

You can check the Text/Text example, it uses it

PA_SetTextTileCol

PA_SetTextTileCol is slightly different to PA_SetTextCol, because it does not change the existing text’s color... But it allows you to have different colors on the same screen (which is pretty nice). The PAlib example is Text/TextColors

```
PA_SetTextTileCol(1, i); // Change the color on the top screen, values 0 to 6
PA_OutputSimpleText(1, 2, i, "Color test...");  // Screen 1 has different colors
```

PA_SetTextTileCol(1, n); sets the text color for the new texts to n... nothing less, nothing more. n can take values from 0 to 9 (0 = white, 1 = red, 2 = green, 3 = blue, 4 = purple, 5 = cyan, 6 = yellow, 7 = light grey, 8 = dark grey, 9 = black).

09/04/2006: I believe Mollusk has just added color defines to the lib:

- Blue can now be accessed using TEXT_BLUE
- Red can now be accessed using TEXT_RED
- The same pattern follows

An example using PA_SetTextTileCol:

```
PA_SetTextTileCol(screen,TEXT_RED); //This will turn the text red
```

%cX

The last way to change a text’s color is to use the %cX input in your string, X being the color number (0-9, like right before). The nice part about this is that you can have multiple colors in the same string ! The examples comes from TextColors, once again :

```
PA_OutputText(0, 0, 0, "Color test...%c1another one, %c2again, %c3again...");
```

As you can see, just putting %cX changes the color (test on emulator if you don’t believe me . It’s simple, easy, and simple, lol.

Custom font & text border

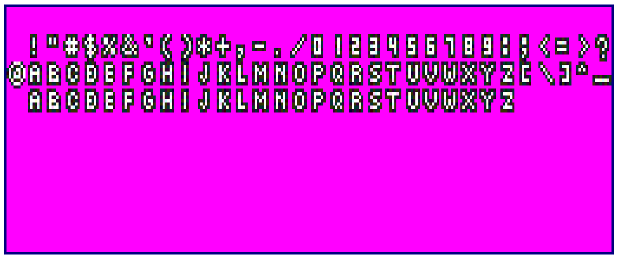
PA_lib allows you to customize the text font and the text border easily. You will need to import graphics to your project, and that’s where it becomes tricky, because it will involve some graphics conversion and if you are reading the wiki in the order, you should be a bit disappointed as you read this.

In fact, graphical data (like jpg images, gif, png...) are not directly compatible with the DS hardware. You have to convert those data into a raw format which can be understood by the DS.

Everything about converting data is well explained in the Sprites and Backgrounds tutorials. So, before you proceed, I suggest you to skip the *Custom font & text border* tutorial till you learned about Converting with PAGfx (day 4) and tiled backgrounds (day 5).

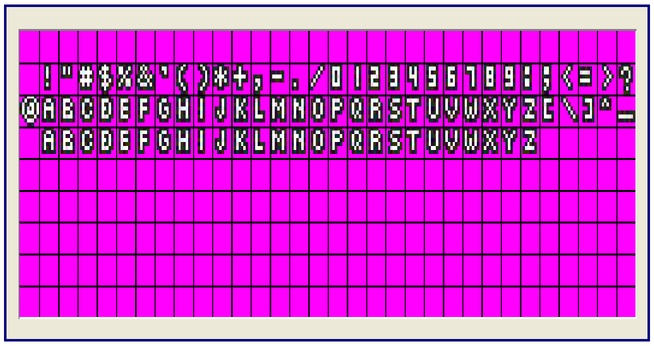
Custom font

Basically, a custom font is a tiled background, each tile representing a specific character. Here is a basic custom font layout:



The layout is very important, each character has its very own coordinate in this layout, so don’t mix them up (unless it’s voluntary). A character has the same size as a tile (8×8 pixels). You can edit this layout to make your own custom font.

For a more accurate editing, I suggest you to display a grid (8×8 pixels) in your favourite graphics editor, like this (zoomed):



One more thing to know about this layout, is how to differentiate capital from normal letters. In this example, capital and normal letters are the same. In fact, the first letter line is for capital letters and the second line is for normal letters.

you can use also PAFont to get you font ready to use without spend time and fatigue!

From now, you need to convert it into a tiled background. Here is the PAGfx settings:

```
#TranspColor Magenta

#Sprites :

#Backgrounds :
newfont.gif TileBg

#Textures :
```

Here we go for the code (assuming you have put your gfx in a directory named *font*):

```
// Includes
#include <PA9.h>       // Include for PA_Lib

#include "font/all_gfx.h"
#include "font/all_gfx.c"



// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL

        PA_InitText(0, 0);  // Initialise the normal text on the bottom screen

        // Load a custom text font on the top screen
        PA_InitCustomText(1,        //screen
                          0,        //background number
                          newfont); //font name

        // PA_OutputSimpleText is the fastest text function, and displays 'static'
text
        PA_OutputSimpleText(1,                  // screen
                            2,                  // X postion
                            2,                  // Y position
                            "Hello World !!"); // display a custom text on the top
screen
```

```
        PA_OutputSimpleText(0, 2, 2, "Hi there :p"); // and a normal one on the
bottom screen

        // Infinite loop to keep the program running
        while (1)
        {
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

As you can see, it's very simple to use a custom font. You just have to initiate the text of a screen with *PA_InitCustomText* instead of *PA_InitText*. That's all.

Tip: How to print out custom fonts by character.

Each slot on the font has an ascii id which is why there are a bunch of blanks before the default symbols and letters come in, well starting at 0 and working your way up you can reference each one by its ID like so:

```
PA_OutputText(screen, x, y, (char[]){65, 66, 67});
```

That will output ABC to the screen, and using this method you can call out any character from the possible 288 characters the PAlib font size allows

One thing that should be noted, however is that there are a lot of reserved special characters between 0-31 (the first row) so you should probably create your special characters 128 and so on.

-ObsidianX

Custom text border

Input : Pad, Stylus, Keyboard

There are basically 3 ways to input on the DS : using the pad (all the keys...), the stylus, or PAlib's keyboard... A last method would be using the microphone, I guess, but we'll talk about that later on, in the Sound tutorial...

Pad

Using the pad could hardly be easier ! It is automatically updated every frame, and a simple structure allows you to access it. The structure is named 'Pad', and it contains multiple structures, depending on the press type : Held, Released, and Newpress.

- Newpress is when you just pressed the keys, stays at 1 for only 1 frame.
- Held is by default to 0, passes to 1 when the key is held down

- Released passes to one when the key get released (was pressed and it no longer pressed...), stays at 1 for only 1 frame.

Each press structure then contains a list of all the keys :

- Left, Right, Up, Down for the arrow presses...
- A, B, X, Y for the ABXY keys
- L, R for the left and right triggers
- Start, Select for... start and select...

So, in the end, to check a key, you have Pad.Checktype.Keyyoucheck Checking if the A button is held down would mean looking at Pad.Held.A ! Pad.Released.A is when it gets released, and Pad.Newpress.A for a new press...

As you can see, this is easy. Here is a quick and sample code that doesn't work but just to show how to use the keys :

```
if(Pad.Held.Up)
{
    MoveUp();
}
if(Pad.Held.Down)
{
    MoveDown();
}
```

Simple, isn't it ?

Check out the Input/Pad example if you want...

— *Mollusk 29/11/2005 22:44*

Stylus

The stylus works pretty much the same as the pad... It is automatically updated every frame, and uses a structure named Stylus (simple...) The different stylus variables are :

- Held, Released, and Newpress, just like the pad ones...
- X and Y, used for the stylus's position

So here's a basic code example :

```
if (Stylus.Held)
{
  PA_SetSpriteXY(screen, sprite, Stylus.X, Stylus.Y);
}
```

As you may guess, this simple code checks for the stylus press. If the stylus is held on the screen, it will position the top left corner of the sprite at the stylus's position... Nothing much more to say

Check out the Input/Stylusexample if you want...

Note by Phaezon: If you're using PA_OutputSimpleText() (or any other related function), make sure you put a few spaces after %d, like so:

```
PA_OutputText(screen, tileX, tileY, "Stylus X: %d   ", Stylus.X);
```

Not doing this could lead to bugs which make you think that the stylus is COMPLETELY off (it said something like 964 pixels on mine) when it gets to double-digit and single-digit numbers. If I had 135 once and then I tapped where 34 is, I'd get 345, since the last number in the three-digit coordinate (the 135) is not overwritten (34 is, obviously, a two-digit number, so the third digit (5) has nothing to be overwritten by. The spaces will overwrite it and make it much more precise.

Keyboard

Using PAlib's keyboard is really easy and simple, as you can see in the Input/Keyboard example...

To work, the keyboard must be loaded on a given background (numbers 0 to 3), different from the text background and any other one...

```
PA_Init();    // Initializes PA_Lib
PA_InitVBL(); // Initializes a standard VBL

PA_InitText(1, 0);  // Initialise the text system

PA_InitKeyboard(2); // Load the keyboard on background 2...

PA_KeyboardIn(20, 100); // This scrolls the keyboard from the bottom, until it's at
the right position

PA_OutputSimpleText(1, 7, 10, "Text : ");

s32 nletter = 0; // Next letter to right. 0 since no letters are there yet
char letter = 0; // New letter to write.
char text[200];  // This will be our text.

// Infinite loop to keep the program running
while (1)
{
        // We'll check first for color changes, with A, B, and X
        if (Pad.Newpress.A) PA_SetKeyboardColor(0, 1); // Blue and Red
        if (Pad.Newpress.B) PA_SetKeyboardColor(1, 0); // Red and Blue
        if (Pad.Newpress.X) PA_SetKeyboardColor(2, 1); // Green and Red
        if (Pad.Newpress.Y) PA_SetKeyboardColor(0, 2); // Blue and Green

        letter = PA_CheckKeyboard();

        if (letter > 31) { // there is a new letter
                text[nletter] = letter;
                nletter++;
        }
        else if ((letter == PA_BACKSPACE)&&nletter) { // Backspace pressed
                nletter--;
                text[nletter] = ' '; // Erase the last letter
        }
        else if (letter == '\n'){ // Enter pressed
                text[nletter] = letter;
                nletter++;
        }

        PA_OutputSimpleText(1, 8, 11, text); // Write the text
        PA_WaitForVBL();
}
```

I guess it's a pretty complexe example (compared to the other ones...), so we'll see that step by step.

It starts by a text init, because... we'll need to display some text !

Then comes

```
PA_InitKeyboard(2); // Load the keyboard on background 2...
PA_KeyboardIn(20, 100); // This scrolls the keyboard from the bottom, until it's at
the right position
```

- The first function loads the keyboard on the bottom screen, on background 2... you can use any background from 0-3 that doesn't have anything on it yet... The second one scrolls the keyboard in, just because it looks nice that way It scrolls it in to position (20, 100), but you can set any position you want...
- *PA_SetBgPalCol(Screen, BG, PA_RGB(31,31,31));* is a function to change the background color. Screen should be 0 or 1 (0 for bottom), BG is for Background number (of the keyboard) (0-3) and PA_RGB is the color (0-31). This isnt used for the keyboard persé, but its the quickest way to change the background color.
- *PA_SetKeyboardColor(0, 1);* is a function to change the keyboard's color... just check the different values and use the one you like most...
- *letter = PA_CheckKeyboard();* is the most important function ! *CheckKeyboard* tests to see if you pressed a key with the stylus, and returns the keyboard key pressed in the form of a character, or 0 if not pressed...

```
if (letter > 31) { // there is a new letter
        text[nletter] = letter;
        nletter++;
}
```

- Another important block... If the key was a letter, we'll add it to the current text... and that's one more letter in it, so nletter++ !
- *PA_OutputSimpleText(1, 8, 11, text);* is the function to display the text we are outputting...

I skipped a few lines of code (the backspace and line return (\n is line return) because even though it's in this code example, you'll have to figure it out by yourself

Compile and test the demo, even works on DualiS ! Enjoy !!

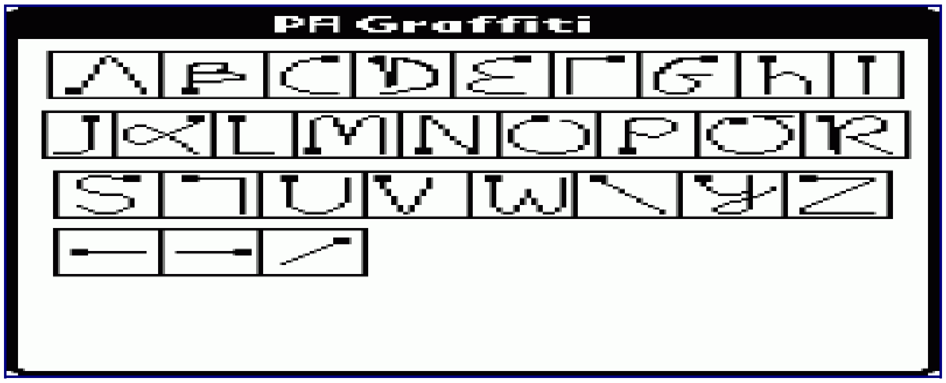
Shape Recognition

This last feature is pretty interesting... You can use it in 2 ways :

- Using PAlib's Graffiti-like recognition
- Using your own custom shapes, to do something like Lost Magic or Pac-Pix !

PA Graffiti

This code will allow you to collect letters using the Graffiti-like system... It will be improved over time. The code is based heavily on the keyboard example's one, so it shouldn't be too hard to figure it out. There are some obscure 8bit stuff you'll see later on in backgrounds



The code comes from Input/RecoGraffiti.

```
PA_InitText(1, 0);  // Initialise the text system on the top screen
PA_Init8bitBg(0, 0); // We'll draw the characters on the screen...

u16 *pal = (u16*)PAL_BG0;
pal[1] = PA_RGB(31, 31, 31); // colors...

PA_WaitForVBL();

u8 nletter = 0;
char text[200];  // This will be our text.

// Infinite loop to keep the program running
while (1)
{

        if(Stylus.Newpress) PA_Clear8bitBg(0); // Reset the screen when we start a
new character

        PA_8bitDraw(0, 1);

        char letter = PA_CheckLetter(); // Returns the letter !!!

        if (letter > 31) { // there is a new letter
                text[nletter] = letter;
                nletter++;
        }
        else if ((letter == PA_BACKSPACE)&&nletter) { // Backspace pressed
                nletter--;
                text[nletter] = ' '; // Erase the last letter
        }
        else if (letter == '\n'){ // Enter pressed
                text[nletter] = letter;
                nletter++;
        }

        PA_OutputText(1, 2, 2, text); // Write the text

        PA_OutputSimpleText(1, 0, 10, "Draw a PAGraffiti letter to have it
recognized by the system...");

        PA_WaitForVBL();
}
```

Now, let’s it part by part...

```
PA_InitText(1, 0);  // Initialise the text system on the top screen
PA_Init8bitBg(0, 0); // We'll draw the characters on the screen...

u16 *pal = (u16*)PAL_BG0;
pal[1] = PA_RGB(31, 31, 31); // colors...

PA_WaitForVBL();

u8 nletter = 0;
char text[200];  // This will be our text.
```

This part just initialises the text and a drawable background, you’ll learn more about that later on...

- `nletter = 0;` Will be used to count the number of letters

```
if(Stylus.Newpress) PA_Clear8bitBg(0); // Reset the screen when we start a new
character
PA_8bitDraw(0, 1);
```

This is some code used to draw on the screen and erase it if you press the stylus again... Later on, later on...

```
char letter = PA_CheckLetter(); // Returns the letter !!!
```

Now we’re getting to it ! This code checks the letter !!! Must be called every frame in order to work (like the keyboard, like the keyboard...) Returns 0 if the letter isn’t finished yet...

And the rest of the code is IDENTICAL to the keyboard example, I will not talk about it again.

.

Custom Shapes

What’s really nice with this system is that you can have custom shapes ! It’s just a matter of seconds to add them, and this example is there to show you...

The code comes from Input/RecoAddShape

```
PA_UsePAGraffiti(0); // Do not use the provided Graffiti shapes...

PA_RecoAddShape('a', "AAAAAAAAAAAAAAA"); // Straight right-going line
PA_RecoAddShape('b', "111111111111111"); // Straight left-going line
PA_RecoAddShape('c', "IIIIIIIIIIIIIII"); // Straight up-going line
PA_RecoAddShape('d', "999999999999999"); // Straight down-going line

PA_WaitForVBL();


// Infinite loop to keep the program running
```

```
while (1)
{

        if(Stylus.Newpress) PA_Clear8bitBg(0); // Reset the screen when we start a
new character

        PA_8bitDraw(0, 1);


        char letter[2]; letter[1] = 0;
        letter[0] = PA_CheckLetter(); // Returns the letter !!!

        if (Stylus.Released){ // only show if released
                PA_OutputText(1, 2, 7, "Shape Recognized : %s", letter); // Letter
recognized
                PA_OutputText(1, 0, 1, "A for right, B for left, C for up, D for
down");
        }

        PA_OutputText(1, 4, 22, "Shape : %s", PA_RecoShape); // Write the shape
string

        PA_OutputSimpleText(1, 0, 10, "Draw a PAGraffiti letter to have it
recognized by the system...");

        PA_WaitForVBL();
}
```

Now the comments :

```
//PA_UsePAGraffiti(0);// Removes the graffiti recognition, so only our shapes will
be taken...

PA_RecoAddShape('a', "AAAAAAAAAAAAAAA"); // Straight right-going line
PA_RecoAddShape('b', "111111111111111"); // Straight left-going line
PA_RecoAddShape('c', "IIIIIIIIIIIIIII"); // Straight up-going line
PA_RecoAddShape('d', "999999999999999"); // Straight down-going line
```

This is where we give PAlib the new shapes to recognize... It might not seem very intuitive, given... How did I obtain them ? By running this rom, lol. You'll see a little bit further. All you need to understand is that a shape is defined by 2 parameters :

- The letter (or number 1-255) which will be returned when it is called
- Its string, given by this rom... When drawing a shape using this code, the string is shown, allowing you to copy and use it ! Nice !!

The rest of the code is actually very similar to the previous example, so I won't comment it more. Only 1 line remains important :

```
PA_OutputText(1, 4, 22, "Shape : %s", PA_RecoShape); // Write the shape string
```

PA_RecoShape is the string in which the shape was stored... That's how I got the 4 shapes we put right before !

And that's it, the program will recognize the shapes we asked

Shape Infos

I lately added a third example to go with the latest version of the lib... It adds informations to shape recognition. This information is basic but can be usefull in certain contexts. The information is stored on Stylus.Released in a structure called PA_RecoInfo. It contains :

- *Length*, the total length of the drawing in pixels
- *startX* and *startY*, the position of the first point. This can be used in a game to determine if a given object/sprite is activated, for instance.
- *endX* and *endY*, the position of the last point.
- *minX*, *minY*, *maxX*, and *maxY* which delimit the shape. This can be used to know how big the shape is, and find it's center.
- *Angle*, the global angle between the first point and the last one... Not always usefull.

Here's the code using it, which is just a copy/paste of the Custom Shape one, so I'll just post the modified stuff... All this is in the infinite loop...

```
if (Stylus.Released){ // only show if released
        PA_OutputText(1, 2, 7, "Shape Recognized : %s", letter); // Letter
recognized
        PA_OutputText(1, 0, 1, "A for right, B for left, C for up, D for down");

        // Use all the info to draw the shape outline, start and end...
        // Rectangle around the drawing zone
        PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.minY, PA_RecoInfo.maxX,
PA_RecoInfo.minY, 2);
        PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.maxY, PA_RecoInfo.maxX,
PA_RecoInfo.maxY, 2);
        PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.minY, PA_RecoInfo.minX,
PA_RecoInfo.maxY, 2);
        PA_Draw8bitLine(0, PA_RecoInfo.maxX, PA_RecoInfo.minY, PA_RecoInfo.maxX,
PA_RecoInfo.maxY, 2);

        // Plot points on the start and end :
        PA_Put8bitPixel(0, PA_RecoInfo.startX, PA_RecoInfo.startY, 2);
        PA_Put8bitPixel(0, PA_RecoInfo.endX, PA_RecoInfo.endY, 2);
}

PA_OutputText(1, 4, 22, "Shape : %s", PA_RecoShape); // Write the shape string
PA_OutputText(1, 4, 23, "Length : %d pixels  ", PA_RecoInfo.Length); // Length in
pixels
```

I won't re-explain what I just said 😛 I just don't like repeating myself, lol. Basicaly, the code which can be used to draw a box around the shape :

```
PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.minY, PA_RecoInfo.maxX,
PA_RecoInfo.minY, 2);
PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.maxY, PA_RecoInfo.maxX,
PA_RecoInfo.maxY, 2);
PA_Draw8bitLine(0, PA_RecoInfo.minX, PA_RecoInfo.minY, PA_RecoInfo.minX,
PA_RecoInfo.maxY, 2);
PA_Draw8bitLine(0, PA_RecoInfo.maxX, PA_RecoInfo.minY, PA_RecoInfo.maxX,
PA_RecoInfo.maxY, 2);
```

It just draws 4 lines using the min/max X/Y positions, pretty standard...

Then, 2 pixels are plotted at the start and end position :

```
PA_Put8bitPixel(0, PA_RecoInfo.startX, PA_RecoInfo.startY, 2);
PA_Put8bitPixel(0, PA_RecoInfo.endX, PA_RecoInfo.endY, 2);
```

...

And the length is displayed with `PA_OutputText(1, 4, 23, "Length : %d pixels ", PA_RecoInfo.Length);`

Nothing more, nothing less... If you see anything missing, some info I forgot which seems vital, please send me a mail (*Mollusk*) or drop on the forum...

Oh, and I didn't put *Angle* in that example because it wasn't that usefull...

Output : 8bit and 16c modes

This section is for advanced users... If you are just starting, I highly recommend skipping this part and going on with Sprites for now, coming back later on... These text modes require a bit more knowledge to master (though it still is pretty easy ^^), and are generally not needed when starting coding games. If you plan on coding applications, though, this could be usefull...

Understanding the basics

The first part of this tutorial talked about simple text output, as well as displaying variable values on the screen and all... So you're probably asking yourself why on earth 2 other different text systems are there too, as the first one allowed you to do pretty much anything you needed...

The default text system uses the DS's hardware tile system, just like backgrounds, to display text. This means it uses 8×8 pixels tiles, and it is impossible to change that... 8bit and 16c text modes, however, use backgrounds you can draw on, plotting pixel per pixel... Starting to see the difference ?

Advantages of the 16c/8bit

Plotting pixel per pixel (or actually 2, 4, or 8 by 8 for speed reasons) has a huge advantage over the tiled text system : you can have different font sizes, and especially letters of different sizes inside a same font ! Yes, you heard it right, this means variable width fonts !

Font Sizes

By default, PAlib offers 5 different font sizes, numbered 0 to 4... The smallest font is like 4 or 5 pixels wide and tall, which means you can put nearly twice as much text per line ! And as you can display more lines of text than the normal output system, this means even more than twice as much text on the same screen ! Nice :p

Ok, that's a cool thing, but it's not all. And actually that size can be a bit of a pain to read. Using sizes 1 or 2 feels better for the user (something like 7 and 9 pixels high fonts), and you still get another advantage from the system... read on :p

Variable Width

In this text system, different letters can have different sizes... Ever wondered why on earth both ‘i’ and ‘w’ would take the same space on the screen ???? Yeah, kind of stupid, so now you can actually use 2-3 pixels wide for ‘i’, and more for ‘w’ :)

This means even more text per line, as you don’t lose any space anymore on small letters. Plus, it looks much better.

If you look at NDS games out there that have text output ingame, like RPGs, Ace Atourney and all, you’ll see that that’s the kind of text they use :)

Disadvantages

If there were only advantages, there wouldn’t be any need for the plain text output system...

Low Speed...

Speed is the first and major issue when using these new text modes... Why ? Well, the normal output system uses tiles. To display a letter, it takes only 1 line of code : changing a number in the map to display the tile corresponding to the letter... However, in 8bit and 16c modes, you have to plot all the pixels ! To save time, pixels are plotted 4 by 4 or 8 by 8, but in all cases you have to know precisely where to plot them, and you have several lines to plot (8 for 8 pixel fonts), so it takes MUCH MORE time...

No Auto-Erase

When you write some text in normal text mode, it automatically overwrites the preceding text, simply because it changed which tiles were used in the map...

When you play with pixels, plotting a pixel doesn’t erase the pixels all over the place (or else you wouldn’t know what you’re doing ^^), so when you want to change a line of text in 8bit/16c, you first need to erase it... Either the whole screen or just a corner, but you’ll have to erase it. This makes 8bit/16c text good for display text files, for example, as it doesn’t require updating the text every VBL, but horrible for debugging, when each variable is updated on the screen every frame...

No Variable to Letters...

Last flaw... These functions do not take the %d, %f, and %s commands for displaying the variables’ content... This is really painfull, and I did it for a specific reason : these text modes are already pretty slow, and adding these features would have made them even slower...

There is a work-around, though (of course !) : using stdio to convert your stuff into another text array, then displaying that array.... (if someone could complete that please ^^) *(done - davido2)*

Here’s an example:

```
char temp[100];
sprintf(temp,"%s %d", "Here's a number:", 10);
PA_SmartText(0,0,0,256,192,temp,1,1,1,999);
```

8bit vs 16c...

Ok, here comes the last thing to know before seeing the actual code... Which should you use between 8bit and 16c text...

16c : Speed

The main reason to use 16c text rather than 8bit is that it’s 2-4 times faster to draw on the screen, or to erase the whole screen...

Second reason ? It takes almost half less space in the Vram, which is nice :)

8bit : Functionnalities

8bit is slower, yeah... But it can also do things you can’t achieve with 16c text... To start off, 8bit can have 256 colors, while 16c only has... 16... That’s nice, can always help...

Plus, the major thing is that 8bit text can be displayed rotated (left or right), meaning you can actually write text to be read holding the DS the other way... This proves to be really good for applications (ebooks, etc...) and some games (just see Brain Academy !), even though it might not be used most of the time...

Conclusion ?

In the end, you’ll want to use 16c rather than 8bit if you don’t have any special usage to it and just want a nice-looking font... 16c will be both faster and smaller in VRAM... If you need to have tons of colors and the ability to rotate the screen, you’ll definately want 8bit :)

16c Text

Because it’s the one I like best, we’ll start out by seeing this one...

Due to lack of time, I’ll just post the link to the example for now ^^

16c Example

```
        PA_Init16cBg(0 // screen (top/bottom)
                   , 3); // bg (0-4)
        PA_Init16cBg(1, 3);  // 16 color background init with default colors for
text

        PA_16cText(1, //screen
                                        10, 10, 255, 20, //x1, y1, x2, y2 position
                                        "Hello World", //text
                                        1, //color (1-10)
                                        0, // text size (0-4)
                                        100); // maximum number of characters (use
like 10000 if you don't know)
```

FAQ - Pad and Stylus

Q1. The stylus positions seem to be completely off ! Can you do something to change that fact ?!?

- Hmm... No, sorry ! Ok, I'll explain. But first, if you have a screen protector, remove it at check if it works better or not (the stylus was like 20 pixels off with the film on, for me, and only 2 off without it !). The current stylus code is known to not be perfect, but is a code that works best for some people (some other codes I tried where even better for me, but 20 pixels off for some people !) ; this comes from the fact that there are tons of different touch screens, and the code we use for the stylus just isn't fit for all of them yet. Also, the stylus position being like 5-6 pixels off on the corners is normal, the hardware precision is much lower there...

Note: For the people with around 20 pixels off (Example, you try to press 'Q' and you get 'A') Go into your touch screen calibration and touch directly OUTSIDE of the mark it tells you to, which every direction you may be off, this will, for the most part fix it. You will need to calibrate it again for other things, however.

Also remember that when writing text and using the stylus coordinates to place it, you will need to divide the stylus X/Y coordinates by about 8. this is because when the OutputText(); function uses tiles and not full coordinates. MC

Q2. When I use Dualis and simulate the stylus by a mouse click... it hangs !

- This is a known issue for Dualis r12, with the latest stylus code. It can be 'fixed' by replacing the arm9.dll by the arm9.dll found in Dualis r11 (though this will add the bugs that were corrected from r11 to r12, for other stuff, and notably break the large backgrounds)

— *Mollusk 29/11/2005 22:44*

<back to top>

<day4>

Day 4 - Sprites

Here is the tutorial to learn about sprites.

DS Sprite Specs

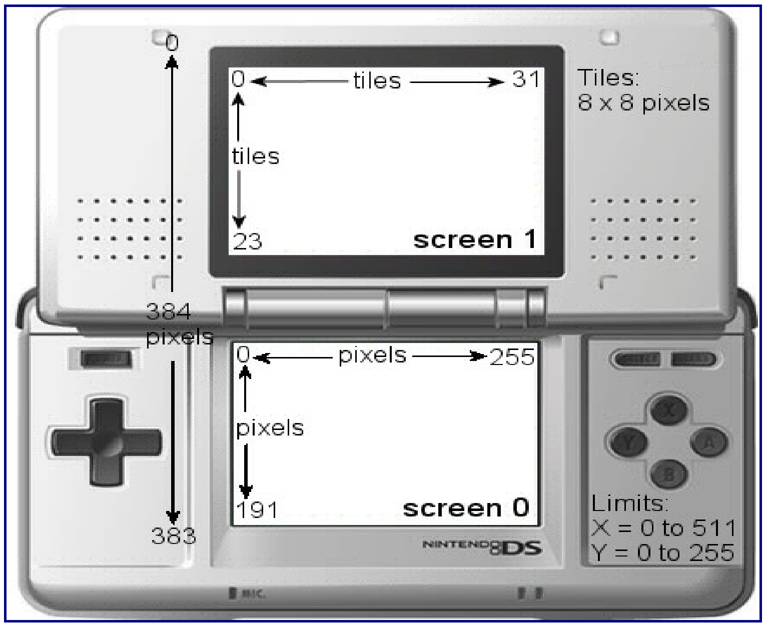
Before starting we will see exactly what the DS is capable of ... 128 sprites can shown on each screen or engine, so a total of 256 different sprites on both.

Each sprite can be horizontal or vertical, can be moved across the screen, can be animated (by updating the image used), can become partly transparent, or even become a patchwork!

Also, the sprites can be rotated and zoomed! However, there is a limit to this point ... You cannot set a rotation / zoom setting for each sprite, but there are only 32 different locations to the screen (called rotsets). And then assign each a sprite rotset (or not). Therefore, all the sprites can rotate, zoom, or both, but only in 32 different ways at once. Several sprites can share the same rotset, which is not a problem, and will be rotated and enlarged in the same way. So only 32 sprites can be affine transformed, which shouldn't bother most people.

DS Screen Sizes

This is a good schema by Bennyboo, which could be useful ... not only for the sprites, but for many other things...



As for the sprites, you just consider about the pixel size ... So the max is 256 pixels wide and 192 pixels high. Given that the first pixel is the number 0, not 1, it means 0-255 and 0-191 ... As shown in the diagram.

In addition, the position of a sprite is limited to certain values. X can only be between 0 and 511, and Y between 0 and 255. That means putting a sprite at the position X = 512 is equivalent to putting it in position X = 0! That is what is called sprite wrapping. When it reaches the limit, becomes the other and vice versa.

Color Modes

Now, with regards to the colors ... The Sprites can have 3 different color modes:

16 color palettes, with a total of 16 different screen palettes. This is widely used in GBA, but disappeared in DS. 16 colour sprites can use all 16 palettes, meaning if you decide to, you can choose different colour palettes for sprites, so you create different looking sprites by only using one sprite, just by changing the palette. Each tile of a 16 colour takes up 32 bytes.

256 color palettes (much better, but uses twice as much memory), with a total of 16 types of palettes per screen (the GBA only had 1). This is what we will use mostly... You can only use the same palette.

16-bit sprites, (that is to say, they don't have a palette). However, these sprites are not used to often, as they use alot more of video RAM, and takes up far too much space to be used.

In the end, it is best to get sprites that are 256 colors:) You'll see that even if this seems a bit limited, it will have large enough colors for each sprite.

anandjones

Sprite Sizes

The DS can handle quite a few different sprite sizes, but you have to always remember that only specific sizes are allowed. The only available width and heights are 8, 16, 32, and 64 pixels. And as a few specific sizes aren't there, here are the sizes you can use (the width is put horizontally, height vertically...) :

	8	**16**	**32**	**64**
8	8×8	16×8	32×8	
16	8×16	16×16	32×16	
32	8×32	16×32	32×32	64×32
64			32×64	64×64

Basically, you can use any sprite size beside 64×8, 64×16, 16×64, or 8×64...

Now, what happens if you sprite isn't exactly the same size than the authorized sizes ? You'll get a pretty ugly and unviewable sprite... There is a simple way to go around this, however : using any paint program, just set your sprite size to a DS size in which it fits. For example, if you made a 48×48 pixels sprite, you can fit it in a 64×64 box ! Same thing would apply for like 10×64, but the best size would be 32×64...

Transparent color

Last thing to take into account is that, I can guess, you don't want the background color of your sprite to be visible... If you have a round sprite, like a frisbee, you wouldn't want the square border to show. In order for the background to be 'removed', or rather set as the transparent color (only 1 transparent color per sprite), you have to choose your transparent color : the best is magenta, as you don't use it very often (red : 255, green : 0, blue : 255), or some use black, but as I often use black in my sprites, I don't find that to be such a good choice...

Here are some common colors used for transparency:
**Color name | R,G,B
Magenta | 255,0,255
"Blackread" | 1,0,0
Off Black | 1,1,1
Off white | 254,254,254

Converting with PAGfx

You know what would have been cool ? If sprites could be used as they are, without any modification, directly on the DS... sorry to disappoint you, stop dreaming ! :'(That's just not how the things are, but you'll see that converting a sprite to the DS format isn't THAT complicated...

The tool we are going to use is PAGfx, a converter created by Mollusk and Kleevah, which could hardly be easier to use. There are 2 ways to use it, one for beginners, using the Visual Interface, one for advanced users, with just the PAGfx.ini file. Both ways offer the same possibilities, don't worry.

PAgfx is now available on Linux! Download latest version here — *Josheat 14/2/2007*

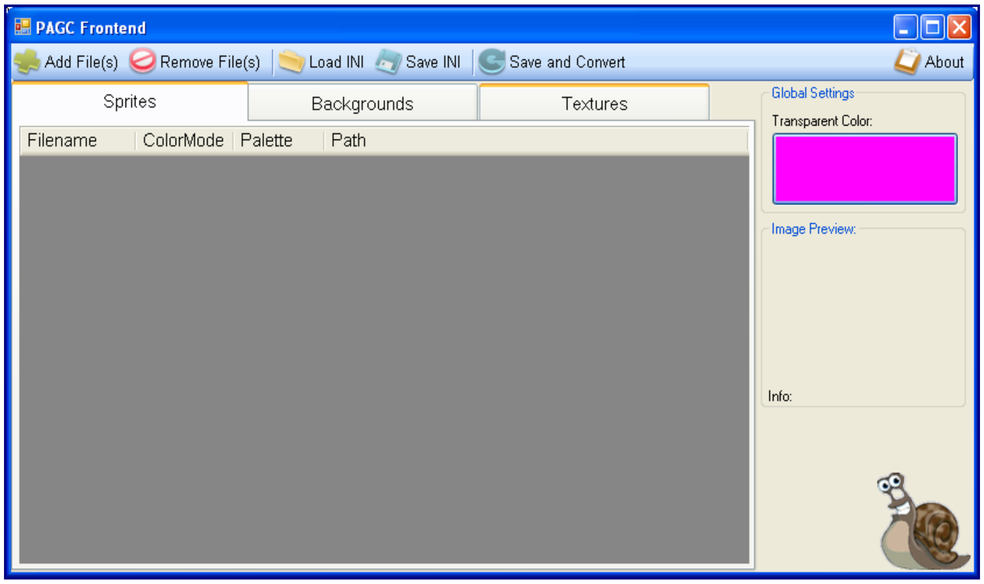
===== PAGfx Visual Interface =====...

First things first... Go into your PAlib/Tools directory, and you should see a PAGfx folder. Open it, and inside you should have 4 files.

- PAGfx.txt is just a quick help and a changelog. You shouldn't need it really, as I'll explain everything here...
- PAGfx.ini is the ini file used for converting sprites, backgrounds, textures... I'll explain it in the second part
- PAGfx.exe is the exe file that converts your sprites, based on the ini file... You'll see that later on too...
- PAGC Frontend.exe... is the frontend ! Now, that's the one you'll want to open right now.

If you get an error message, it's because you don't have the .Net framework installed (yeah, sometimes life sucks...). You can just download and install it

Now, when opening the frontend, here's what you get :



Buttons

I'll explain the buttons, but it's pretty self-explanatory...

- Add Files adds a file to either sprites, backgrounds, or textures, depending on where you are...
- Remove Files removes the selected files
- Load INI loads the PAGfx.ini file in the current directory
- Save INI saves all the sprite/background/texture infos to PAGfx.ini for later conversion with PAGfxConverter.exe
- Save and Convert just saves to the INI file, and loads PAGfxConverter.exe to convert it immediately, which is what you should use the most...

Next, you see that there are 3 possible tabs : Sprites, Backgrounds, and Textures... nothing too complicated to understand...

Setting the transparency

As I said before, you can choose which color will be the transparent color for you sprite background, and that's rather important. In PAGC Frontend, you have a Transparent color part, and in it you can choose between 4 colors : black, white, magenta, and green. As said before, I recommend Magenta, but Black is often used.

Adding a Sprite

Ok, to see how it works, we'll start off by adding a file... just add any image file (make sure it's in the same directory as the PAGC Frontend!). Let's see what the options are for the sprites...

- Filename : the filename ! Duh...
- ColorMode : This is an important one... you can choose between 16colors mode, 256colors, or

16bit. Basically, what you'll want the most is 256 color mode.

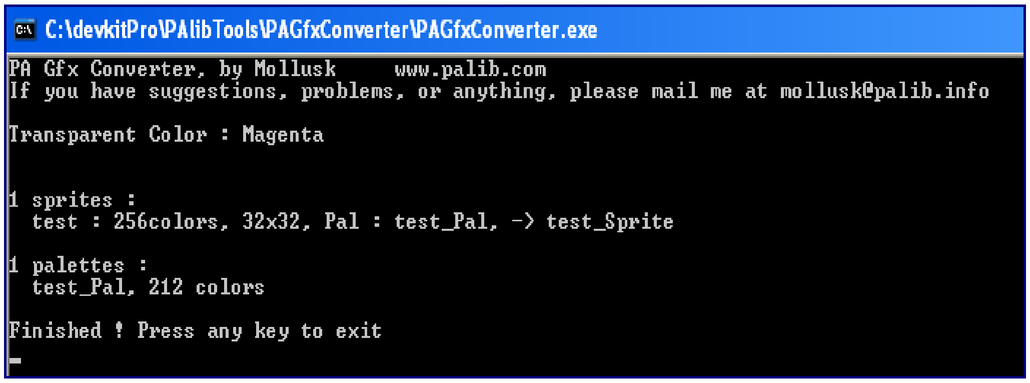
- PaletteName : This is the 2nd important one... as seen before, the DS can have 16 palettes per screen for the sprites. By default, the program puts as palette name the sprite name. You can change this to have like sprite0, and for other sprites sprite1, etc... If you want several sprites to have the same palette, just put them the same palette name Can't make it much easier...

  The palette's real name will be PaletteName_Pal, but we'll see that very soon...
- Path : just the path... nothing much...

And that's basically all you need to know ! Oh, and sprites must have width/height multiples of 8, if not it won't work ;)

Converting

Now, click Save and Convert, it should output some weird info that you might already understand, and if not it's ok, you don't care for now, and then end up telling it worked (if your sprites had less than 256 colors... if not, you're in trouble ! just kidding, if not, use a sprite from the PAlib Sprite examples). If it worked, you should get something like this :



C:\devkitPro\PAlibTools\PAGfxConverter\PAGfxConverter.exe

```
PA Gfx Converter, by Mollusk     www.palib.com
If you have suggestions, problems, or anything, please mail me at mollusk@palib.info

Transparent Color : Magenta


1 sprites :
  test : 256colors, 32x32, Pal : test_Pal, -> test_Sprite

1 palettes :
  test_Pal, 212 colors

Finished ! Press any key to exit
_
```

NOTE!!: For some computers, it will not say "Press any key to exit." This, however, shouldn't affect the created file(s).

You'll see that the converter outputed a few files :

- The SpriteName.c is the file containing all your sprite's data, and will be used to load the sprite
- The PaletteName.pal.c contains... the palette ! So you'll need that too to load your sprite with correct colors...
- all_gfx.c is a pretty cool file, it actually links to all the sprite, background, and palette files... Why ? because instead of having to add each image.c file to your project, you'll just have to add this one, and it'll add everything for you
- all_gfx.h is pretty much the same as the .c, but contains the sprites' data names and all, you'll see that later on...

If you want to, you don't have to include the all_gfx.c file, just only the header .h. Just have your binary .bin files (outputted in bin folder from PAGfx), and put those in your template data folder. However, at the cost of a longer compile time. But if you want to, what I do to save a lot of time, is just edit my batch .bat file in Notepad, and then just remove the make clean line. Saves a lot of clean time. And then if you want to, just add the line back in. Easy :)

AJ

PAGfx.ini

For those out there who are curious, you can now check the PAGfx.ini that was created by the Frontend... If you just want to use the frontend, don't bother, and skip this part...

It should be really basic, something like

```
#TranspColor Magenta

#Sprites :
C:\test.png 256colors test

#Backgrounds :

#Textures :
```

It's not hard to understand what all this stands for...

- #TranspColor Magenta is the transparent color you want to use for sprites and backgrounds. It can be White, Black, Green, or Magenta...
- #Sprites : will list all the sprites to convert... Each line is like C:\test.png 256colors test
    - C:\test.png is the path towards the image. You can put a relative path, like just test.png if PAGfxConverter.exe is in the same folder...
    - 256colors is the color mode, so 16colors, 256colors, or 16bit, for sprites... (we'll see backgrounds later on).
    - test is the palette name, I left it by default, so the palette will be test_Pal...

And the rest doesn't matter much yet, as it's for backgrounds and textures Simple, isn't it ? You can convert PAGfx.ini by using PAGfx.exe... It's the converter called from the frontend

PAlib Sprites

In the first sprite tutorials, we will concentrate on plain, basic sprites, and having them run up and down the screen. Later tutorials will cover sprite rotation/zooms, sprite transparency, and sprite animation. Be patient !

Displaying

The time has finally come to actually use the DS's hardware ! Yahoo !! Here comes the first sprite displaying tutorial...

The following examples are taken from PAlib v0.72a, because I update the sprite stuff in that version... Please install this version (or above) before continuing, or else you will not have the exact same examples in the PAlibExamples folder...

Open PAlibExamples/Sprites/Basics/CreateSprite, and check out the main.c code in it :

```
#include <PA9.h>

// PAGfxConverter Include
#include "gfx/all_gfx.h"
#include "gfx/all_gfx.c"

int main(void){

        PA_Init(); //PAlib inits
        PA_InitVBL();

        PA_LoadSpritePal(0, // Screen
                        0, // Palette number
                        (void*)sprite0_Pal);    // Palette name

        PA_CreateSprite(0, // Screen
                        0, // Sprite number
                        (void*)vaisseau_Sprite, // Sprite name
                        OBJ_SIZE_32X32, // Sprite size
                        1, // 256 color mode
                        0, // Sprite palette number
                        50, 50); // X and Y position on the screen

        while(1) // Infinite loops
        {
                PA_WaitForVBL();
        }
return 0;
}
```

As you may see, everything is explained already ! I won't come back on the stuff explained in Day1 (check out the template decoding part...), assuming you did that already...

```
// PAGfxConverter Include
#include "gfx/all_gfx.c"
#include "gfx/all_gfx.h"
```

This part includes all the graphics you have converted and put in source/gfx. Nothing more to say about it (you can check that folder, that's where the sprite was converted, with its palette and all...)

```
PA_LoadSpritePal(0, // Screen
                0, // Palette number
                (void*)sprite0_Pal);    // Palette name
```

This is important : it's the palette loading ! It loads a palette on the bottom screen (screen 0 is the bottom screen, 1 the top screen...), in the first palette (palette 0), and the palette loaded is sprite0_Pal... Don't worry about the '(void*)' in front of the palette name, it's just to indicated that it gives the palette's location... The top first reason for a sprite not showing is... forgetting to load a palette for it ! and be carefull, load it with the correct number (since there are 16 different palettes for the sprites...)

```
PA_CreateSprite(0, // Screen
                0, // Sprite number
                (void*)vaisseau_Sprite, // Sprite name
                OBJ_SIZE_32X32, // Sprite size
                1, // 256 color mode
                0, // Sprite palette number
                50, 50); // X and Y position on the screen
```

Here, everything is explained again, this example really is easy !

- Screen 0 for bottom screen, again... If you load on the top screen, the sprite will not show. Why ? because the palette was loaded for the bottom screen !
- Sprite number comes next... As said before, the DS can have up to 128 sprites on each screen, numbered from 0 to 127 (included). For each sprite you create, you therefore have to choose a sprite number, which will be used later on to modify that sprite (update it, move it around, etc...). Furthermore, the sprite number determines which sprites are in front of which (sprite 0 is in front of all sprites, 1 in front of all besides 0, etc...) The lower the number, the higher the priority it has.
- The obj size is composed of 2 variables which aren't actually easy to use, so the macros OBJ_SIZE are there to help... If you are unsure what sizes are available, check the beginning of this tutorial...
- The color mode comes next. It can be only 2 things : 0 for 16 colors, 1 for 256 colors. You'll almost always use 256 colors... (The code to display a 16 color sprite is not the same as the 256 color one.)
- Then comes the palette number. Here, it's 0... If you put the wrong number, you will either get a sprite with odd colors (if it uses another palette), or no sprite (no palette loaded...). So be carefull when using that one.
- Then comes the X and Y coordinates of the sprite, in pixels, nothing much. This coordinate does not use the sprite's center, but the upper left corner.

And that's basically all there is to understand ! Once you got that, just double-click the build.bat, and open the rom in the emulator or on DS... As the infinite loop (while(1)...) does not contain any code (other than the VBL), your sprite will just sit there and not do anything.

Before going on to the next part, try modifying this example to have it display the sprite on the top screen, then on both screens...

For a 16color sprite, you have to use the following:

The palette:

```
PA_LoadSprite16cPal(0, // Screen
                     0, // Palette number
                    (void*)sprite0_Pal); // Palette name
```

And for creating the sprite:

```
PA_CreateSprite(0, // Screen
                                0, // Sprite number
                                (void*)vaisseau_Sprite, // Sprite name
                                OBJ_SIZE_32X32, // Sprite size
                                0, // 16 color mode
                                0, // Sprite palette number
                                50, 50); // X and Y position on the screen
```

Moving

Here comes what could be the most important aspect of sprites... moving them around ! We'll see 2 different methods, the first using the stylus, and the second being the universal method you'll use all the time...

PA_MoveSprite

Ok, first, open the MoveSprites example in PAExamples/Sprites. We won't paste all the code this time, as tons of it are similar to what you've already seen...

```
for (i = 0; i < 16; i++) PA_CreateSprite(0, i,(void*)vaisseau_Sprite, 
OBJ_SIZE_32X32,1, 0, i << 4, i << 3);
// This loads sprites a bit everywhere
```

Ok, now, why is there a *for* before the the CreateSprite function ? Simply because we would like to create 16 sprites (just for testing), similar... As you can see, the sprite number is... *i* ! That means that we will create sprites with numbers 0 to 15. Last, you see that the x and y positions are *(i«4)* and *(i«3)*. I won't explain that too much just yet, but it means i*16 and i*8. There'll be a tutorial later on, in the math part, concerning these stuff... Why *i*16* ? Because the first sprite (i = 0) will be at 0, the second at 16, etc... So we'll have sprites all over the screen! Cool!

Next comes, in the main loop, the important code you want to memorise :

```
while(1)
{

        // Use the MoveSprite function on all sprites...
        for (i = 0; i < 16; i++) PA_MoveSprite(i);
        // The MoveSprite function checks if you are touching a sprite, and moves 
it around if*
        // *you are... Pretty nice if you have multiple sprites around

        PA_WaitForVBL();
}
```

This is a basic main loop, with just a single function in it : *PA_MoveSprite*... now, this function couldn't get any easier; This function checks if the given sprite number, on the bottom screen, is a sprite which is touched by the stylus... If that's the case, it'll *bind* with it, and they become linked. As long as you keep the stylus held, the sprite will move to its position 😃 So why is the *for* loop here again ? Because we want to test sprites 0 to 15! Yup, that simple...

You can now compile the code and test it on your DS, because it crashes Dualis r12 (not r11)... You'll see this simple code can give a pretty good result, and wasn't much effort !

PA_SetSpriteXY

Here comes the mighty most important function concerning sprites !! The SetSprite function !! It's use is pretty simple and basic :

```
PA_SetSpriteXY(screen, sprite, x, y);
```

Could it be easier to understand ? You just have to give it which sprite to move (screen (0 for bottom, 1 for top screen) and sprite number), and then the new position (x and y), and the sprite is assigned this new position. There is only one thing you really need to know : the position corresponds to the sprite's upper left corner, not the center. So if your sprite is like 32×32, you'll want to put as coordinates *x-16* and *y-16* to have it centered on point *x, y.*

On another note, you can also use PA_SetSpriteX and PA_SetSpriteY if you do not want to set the x and y positions at the same time, it goes faster...

X and Y limits

There are 2 more functions available : *PA_GetSpriteX(screen, sprite)* and *PA_GetSpriteY(screen, sprite)* which return the sprite's coordinates in PAlib's sprite system... The first thing you'll notice is that these values will always range between 0 and 511 for X, 0 and 255 for Y, because these are hardware limits... Considering this, if you use values above or below these, the sprite will wrap around... A sprite at position x = 512 will be given the position x = 0, and will therefor be on the screen ! If the value of X is set to a negative number, the sprite wraps the other way. That is, a sprite of width 10 with an X value of -10 will be invisible as its placed ot position (512-10= 502) which is off screen. Same thing applies for Y, but with more limited values...

Keys

Now that you know how to use this function, the first thing we'll see is how to move a sprite with the DS's cross ! Now, let's check the code in PAExamples/Sprites/MoveSpritewithKeys...

I will only paste the important lines :

```
s32 x = 0;    s32 y = 0; // sprite position...
```

These are 2 variables (we could have used structures now that you know how to use them)

that will store the sprite's position...

```
x += Pad.Held.Right - Pad.Held.Left;
y += Pad.Held.Down - Pad.Held.Up;
```

What's this ??? I know that it's not really what comes to mind concerning sprite movement. The easiest way to do it would have been like

```
if (Pad.Held.Right) x = x + 1;
```

(to move the sprite 1 pixel...)

Here, what happens if you press Right ? Pad.Held.Right gets value 1, and Pad.Held.Left gets value 0. So

```
x += Pad.Held.Right - Pad.Held.Left;   -> x += 1 - 0;
```

So it moves the X position by 1 pixel, exactly like the if stuff did, except that this works for Left (-1 pixels) and does not use the *if*, which is slow !

Same thing for up and down...

Now, we're getting to the last part of the code :

```
PA_SetSpriteXY(0, // screen
                0, // sprite
                x, // x position
                y); // y...
```

Pretty easy, already explained... Now that X and Y positions have been updated according to the Pad, we move the sprite to that position !

You are now free to compile this code and test (works even in dualis...)

For those who want to go a little further, how can you change the movement's speed ? Say you want to move by 2 pixels instead of 1...

...

..

.

Time up ! The correct answer is :

```
x += (Pad.Held.Right - Pad.Held.Left) * speed;
y += (Pad.Held.Down - Pad.Held.Up) * speed;
```

(speed can be any value, such as 2 for 2 pixels...) If you don't believe me, just test it, replace speed by the number of pixels to move per frame and you'll see the sprite moving around faster

Stylus

To check this one out, just open the MoveSpriteWithStylus example, and you'll see it's even easier to understand than the previous one ! Basically, there's just 1 important line of code (the rest is sprite loading and displaying the stylus's position on the screen...)

```
PA_SetSpriteXY(0,0,Stylus.X,Stylus.Y);
```

It puts sprite 0, (on the bottom screen) to the stylus's X and Y coordinates... nothing more to say about it I guess

With all this, you have enough knowledge to begin programming on your own. If you just started DS dev, I would recommend moving on to the next tutorial (backgrounds) before finishing this one... Or you could just read the rotation part and then move on. If you're an experienced programmer, you're welcome to continue on to the end of this tutorial...

Moving without wrapping

With a little help from Mollusk, I've finally devised a way to move a sprite without wrapping it. This is useful especially in large maps where you have for example character sprites or something that has to scroll out of view and return when you come visit it again. There are several much more complex ways to do this, but you're probably going to use something like this as a beginner. Also note than in the Platform Game 5, the coins are deleted when they exit the screen and then created again once you stumble upon them. This is useful if you have lots of spare processing time (meaning you don't have too much going on) and not enough memory to store every sprite. But sometimes, you have lots of memory, so the sprites can just stay there. You want this, for example, in shared multiplayer because one player can be within range to see the sprite, and another player could be too far away from it. This just keeps the sprite there for everyone to see. Keeping the sprite there also makes sense if you don't have a lot of sprites, and you could easily spare those few kilobytes of VRAM. Lets take a look at a classic no wrapping function:

```
void SetSpriteNoWrapX(u8 screen, u8 sprite,,u8 size, s16 x) {
    if ((x < -(size-1)) || (x >= 256)) {
        PA_SetSpriteX(screen, sprite, 256); //If the sprite is out of bounds, move
it offscreen
    }
    else {
        PA_SetSpriteX(screen, sprite, x); // Otherwise, move it to the desired
position
    }
}
```

If you’re as smart as me (lol), you’d care to point out that if you’re scrolling around, the sprite can’t have a position in terms of screen pixels, because otherwise it would just stay there and not move. Therefore, you must have your sprite positioned on a *scroll grid*. Scroll grids, as I call them, automatically change the positions of objects as you scroll around. Although the scroll grid also uses pixel-based values, it is much more flexible as it can scroll the object in question off the screen. More about scroll grids in Day 6.

-Phaezon (martin.hanzel@live.ca)

Stylus Touch

PAlib offers a function or 2 to know if a given sprite is touched by the stylus. This can be used to activate some functions in a game or application. Note, however, that doing such a function yourself would be faster, as it would be perfectly adapted to your game, whereas PAlib’s function must be general and work for everyone...

The example in PAlib is SpriteTouched.

```
u8 i = 0;
for (i = 0; i < 8; i++) PA_CreateSprite(0, i,(void*)mollusk_Sprite,
OBJ_SIZE_32X32,1, 0, i << 5, i << 4);

PA_OutputSimpleText(1, 0, 10, "Please touch a sprite");

while(1)
{
        // Now we'll test every sprite to see if we touch it...
        for (i = 0; i < 8; i++) {
                if (PA_SpriteTouched(i)) PA_OutputText(1, 0, 15, "Sprite %d  ", i);
                // If we touch the sprite, returns 1...
        }

        PA_WaitForVBL();
}
```

The first 2 lines just create sprites at different positions... That’s the same as the Sprite Moving example, nothing much to say about it...

But in the infinite loop comes the interesting code : *PA_SpriteTouched(sprite)*. This simple function returns 0 if the sprite isn’t touched by the stylus, and 1 if it’s touched... This is enough to test one sprite at a time, but here we had 8 different sprites. That’s why I added the *for* loop, which will here test for sprites from 0 to 7, and display on the screen which sprite was touched. You could replace the text output by something like *spritetouched = i;*, which would store the touched sprite’s number in the spritetouched variable...

Nothing more to say about this example, it’s just pretty simple and basic Hope you understood everything !

Rotation and Zoom

Here comes another important tutorial, this time concerning rotations and zooms ! As I may have said before, sprites can be rotated and/or zoomed at will, but there is a small limitation : even though all the sprites be rotated, a rotated sprite needs to have a *rotset* attributed to him, and only 32 rotsets are available per screen or only 32 affine sprites... Then how can all the sprites be rotated, if only 32 rotsets are available ????? Well, you can have several sprites on the same rotset, and, in that case, all these sprites will be rotated/scaled with in the exact same way ! Imagine you have a space ship, which can turn around, and is made up of several parts (let’s say 2 sprites for the wings, 1 for the body, 1 for the cockpit, and 1 for the guns...). They all need to be rotated and scaled in the exact same way, so instead of having each a different rotset, it’s just easier to give them all the same one

There are 3 examples for rotations and zooms in the PAlib Sprite Examples : Rotation, Zoom, and RotZoom, which use 3 different functions, one for only rotating, one for only zooming, and one which does both at the same time...

Rotation

We’ll start with Sprite Rotations... The example is *Sprite_Rotation*, and I’ll paste here the interesting part of the code :

```
// Activate rotations for that sprite
PA_SetSpriteRotEnable(0,// screen
                      0,// sprite number
                      0);// rotset number. You have 32 rotsets (0-31) per screen. 2
sprites with*
                         // *the same rotset will be zoomed/rotated the same way...

u16 angle = 0; // Rotation angle...

while(1)
{
        ++angle; // change the angle
        angle &= 511; // limit the range to 0-511. works only with 1, 3, 7, 15, 31,
etc... (2^n  - 1)

        // Fast function for rotations without zoom...
        PA_SetRotsetNoZoom(0, //screen
                           0, // rotset
                           angle); // angle, from 0 to 511

        PA_WaitForVBL(); // Synch
}
```

First off, you create a sprite, just as usual, with its palette and all... Then, you need to activate the sprite in rotation/zoom mode and give him a rotset (from 0 to 31, remember ?), which is done with *PA_SetSpriteRotEnable(screen, sprite, rotset)*. Once this is done, the sprite is ready to rotate ! When you want to stop rotating it, you can use *PA_SetSpriteRotDisable(screen, sprite)*.

The few lines of code directly following aren't too important, it's just declaring a variable for the angle, and adding 1 'degree' to it each turn... There is, though, a major thing to know about the angle. It's not a 360° angle, but ranges from 0 to 511, and it's counterclockwise... Why ? because that's what is best for the DS, and it's much faster that way than with normal angles... You'll get used to it So what's the *angle &= 511;* all about ? You'll have a more complete tutorial on the *&* operation in the math tutorial coming up. All you need to know is that it limits the variables range to that value, but only takes 2^n - 1 numbers : 1, 3? 7, 15, 31, 63, 127, 255, 511, etc... So it limits the variable's range to 0-511, which is exactly our angle's limits ! Hehehe... What happens if the value goes over 511 ? It goes back to 0, and so on. So 512 will give 0, 513 will give -1, etc... And that's true for negative values, -1 giving... 511 ! This is cool (and one of the reasons we don't use 360° angles).

Next comes the important function, *PA_SetRotsetNoZoom(screen, rotset, angle);* . Never forget that here, you are manipulating the rotset (0-31) and not the sprite, that's why you don't use the sprite number anymore... You just give the screen, the rotset, and the angle (0-511) as arguments, and *PA_SetRotsetNoZoom* will rotate the sprite just like you want !

As this example has an angle to which 1 is added every frame, you can compile the example and see on hardware how it turns... It *can* work on DualiS... but doesn't always, sorry.

Last thing : the sprite is rotated from it's central point

Zoom

Now that we have seen how to rotate a sprite, we'll see how to zoom it. It's fairly easy, once again, and the example, *Sprite_Zoom*, is really similar to the one we've just seen... I won't paste the Rotset Enabling part of the code this time, you already saw it , right ?

```
u16 zoom = 256; // Zoom. 256 means no zoom, 512 is twice as small, 128 is twice as 
big....

while(1)
{
        zoom -= Pad.Held.Up - Pad.Held.Down; // Change the zoom according to the 
keys...

        // Fast function for zoom without rotations...
        PA_SetRotsetNoAngle(0, //screen
                            0, // rotset
                            zoom, zoom); // Horizontal and vertical zoom. You can 
have a sprite*
                            // *streched out if you want, with the zoom only for x 
or y axis....
```

First off, you have the zoom variable... Why is it set to 256 ? Because... 256 means NO zoom. The DS zoom works the following way :

- 256 is no zoom, or rather 100% size...
- 512 is half zoom, 50% size...
- 128 is double zoom, or 200% size...

I know this is a bit painful at first, but you'll get use to it. To make it simple, the base zoom is 256, smaller values give a bigger sprite, and bigger values give a smaller sprite. Here, the zoom is modified by the Pad Up and Down keys, so you can check for yourself the result...

Now, let's check the important zoom function : *PA_SetRotsetNoAngle(screen, rotset, zoomx, zoomy);*. Why are there 2 zoom values ??? Because you can zoom your sprite independently on the X and Y axis if you want ! Even though most of the time you'll want to zoom it the same way on both axis, it can be good to have different zooms... I used it in a space ship demo, to stretch the ships vertically and flatten them horizontally, which made a nice squeeze effect during hyperspace.

Just compile and test ! You'll see how this looks like The sprite is zoomed from the central point, just like for the rotations...

If you're smart (or if you were careful when reading this tutorial and the Zoom example in PAlib), you should have 2 questions, which are in fact pretty much the same :

1. What happens if a sprite is zoomed too big ? After all, when creating the sprite, you set its size, so having a too big sprite should go beyond this size, and... not work ??
2. Why didn't I talk about *PA_SetSpriteDblsize(0, 0, 1);*, which is in the PAlib example ??

Well, first : if your sprite is too big, everything beyond the sprite frame (like 32×32, for example), will just be cut off ! Ok, this is very limiting, but hey, Nintendo thought of everything ! And that's where question 2, *PA_SetSpriteDblsize(screen, sprite, enable/disable)* comes in ! If you double-size a sprite, it won't actually zoom it to 200%, but double's it's frame size ! So a 32×32 sprite becomes a 64×64 (with the 32×32 image in its center), and a 64×64 becomes... 128×128 ! Wow, that's a big sprite

Now that you know this, you see how to go beyond the sprite size limit if you want to zoom your sprite a little more... You have to consider one last thing, though : if you double the canva size, and since the sprite position is the top left corner... it changes the sprite's center on the screen... So if you place 2 sprites at the exact same coordinates, but one double-sized and one normal, you'll see they're not placed the same way... Never forget that !

Some helpful functions: By Prob_Caboose

```
float NDSRotToAng(float input){
        return ((input*100)/142);}

float AngToNDSRot(float input){
        return ((input*142)/100);}
```

Both Rotating and Zooming

These 2 examples just taught you how to rotate or zoom, but in fact you can do both at the same time if you need to... The PAlib example is the Sprite_RotZoom one, and it's very much like the other ones, once again, but contains both the angle and zoom variables... I will not post any code this time, as it's useless, just look at the example

The only function, this time around, is *PA_SetRotset(screen, rotset, angle, zoomx, zoomy);*. It's just like a combination of the 2 preceding functions, nothing terrible !

I guess you are now set and know all you need to know about sprite rotations and zooms ! Off to another lesson...

Sprite Flipping

Sprite flipping is something really simple on DS, so even though it could seem like a minor subject, I'll show how easy and important it can be... I'll add an example in the next release (Sprite/Flips).

There are just 2 functions to flip a sprite :

- *PA_SetSpriteHflip(screen, sprite, 1/0 for yes/no)*, for horizontal flips (left/right)
- *PA_SetSpriteVflip(screen, sprite, 1/0 for yes/no)*, for verticalflips (up/down)

This is really too easy for you ! 1 sets the flip on, 0 removes the flip, that's it. You can flip horizontally, vertically, or both at the same time.

Now, where/when can this be used ? Imagine you're doing a fighting game... Like Street Fighter 24532 AlphaBetaGamme XX. You have one player looking towards the right, and one towards the left. There would be 2 ways of acheiving this :

- Having twice as many graphics, one for left and one for right... This is a lot of unnecessary work and takes up tons of space. It is also slow to update...
- Using Horizontal Flip to invert left/right ! Fast and Simple...

What seems like the best solution ?

Mosaic Effect

The Mosaic is an easy effect to use, but I don't like it that much It was used in Super Mario Bros, for example, and changes a bit the way the sprite looks by replacing single pixels by squares of a given size (from 1×1 blocks to 15×15 pixel blocks). What's so good about that ? It can be used for transition effect on sprites, a sort of blurring effect, etc... But as I said, I don't like it that much and never use.

Check the Mosaic sprite example in the next version of PAlib coming out...

Anyways, the DS and PAlib offer you a very simple control over the Mosaic effect. Once you have created a sprite, you can activate the effect for a given sprite using *PA_SetSpriteMosaic(Screen, Sprite, Mosaic on/off (1/0));*.

Once this is done, you can easily change the mosaic values, horizontally and vertically (so you could have like 1×8 pixel blocs if you want !) using the following function : *PA_SetSpriteMosaicXY(screen, horizontal block size, vertical size...);*

Have you seen the BIG limitation of this ? Well... You can control each sprite to activate or deactivate the mosaic effect, but the mosaic block level is the same for all activated sprites on a given screen ! So you can't have one sprite with a certain mosaic, and another with a different one... that's the second reason why I don't like this effect that much...

Just compile the example and test on DS, should work (though it doesn't on DualiS).

Transparency

Transparency, or Alpha-Blending, can be nice in a game or application. It looks great and professional, and in fact isn't hard to do !

PAlib uses the DS's hardware to achieve it, but this has 1 important limitation : even though you can set any sprite you like to be alpha-blended, ALL the alpha-blended sprites will have the same transparency level... So you can't have like one sprite half-transparent and another almost completely... They'll all be the same...

The code is fairly easy to understand, and you have an AlphaBlending example in the sprite examples...

```
PA_SetSpriteMode(0, // Screen
                 0, // Sprite
                 1); // Alphablending


s16 alpha = 7; // Transparency level

// Enable the alpha-blending
PA_EnableSpecialFx(0, // Screen
                   SFX_ALPHA, // Alpha blending mode
                   0, // Nothing
                   SFX_BG0 | SFX_BG1 | SFX_BG2 | SFX_BG3 | SFX_BD); // Everything
normal

while(1) // Infinite loops
{
        alpha += Pad.Newpress.Up - Pad.Newpress.Down;
        PA_SetSFXAlpha(0, // Screen
                       alpha, // Alpha level, 0-15
                       15); // Leave this to 15

        PA_WaitForVBL();

}
```

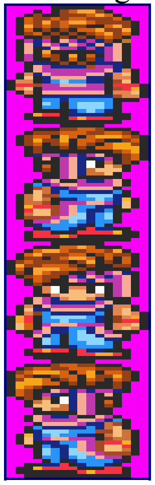
The sprite is created like any normal sprite, but has it's alphablending option activated with *PA_SetSpriteMode(screen, sprite, mode);* There are different modes, you can check the documentation, but for now just remember that 0 is nothing special, and 1 is alphablending...

Once activated, the sprite isn't alphablended just yet. You first need to activate the DS *special effect* system, setting it to alphablending mode too... *PA_EnableSpecialFx(Screen, SFX_ALPHA (Alpha blending mode), 0 (leave to 0 for now), SFX_BG0 | SFX_BG1 | SFX_BG2 | SFX_BG3 | SFX_BD);* The big list is a list of everything besides the sprites, don't worry, you'll understand that some other day...

PA_SetSFXAlpha(screen, alpha level (0-15), normal level (leave to 15)); is the last important function (the alpha variable I put is modified by the Up and Down keys...), and is used to set the transparency level... Just compile and test different values by pressing Up and Down to see what the sprite looks like...

Frames

What are frames for ? They can be used to animate a sprite in complex ways, to bring it to life ! I'll talk about sprite animations really soon, because frames are in some way easier to use, but also more flexible... Mastering them fully, however, can take some time.

First thing : if you want to use frames, you must put all the sprite frames in a single image when converting, and these frames must be one on top of the other, like this : 

That's the secret to nice animations... This example will not show any animation, but the most common and usefull use of frames : having the image change depending on the key presses... In this example, the sprite will look in the direction pressed, which is what you need to have it come to life !

```
PA_CreateSprite(0, 0,(void*)frames_Sprite, OBJ_SIZE_16X32,1, 0, 128-16, 64);

while(1)
{
        if (Pad.Held.Up) PA_SetSpriteAnim(0, 0, 0); // screen, sprite, frame
        if (Pad.Held.Down) PA_SetSpriteAnim(0, 0, 2); // screen, sprite, frame
        if (Pad.Held.Left) PA_SetSpriteAnim(0, 0, 3); // screen, sprite, frame
        if (Pad.Held.Right) PA_SetSpriteAnim(0, 0, 1); // screen, sprite, frame

        PA_WaitForVBL();
}
```

As you can see, the sprite is created like any normal sprite ! That's what's cool... Then, you see the code to change the frame, *PA_SetSpriteAnim(screen, sprite, frame number);*. And that's it ! In this code, that frame is changed depending on the keypress, in order to have the sprite look at the right direction...

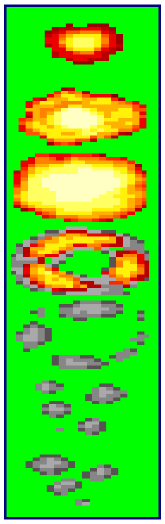
Nothing more to say, so we'll move on to sprite animations !

Oh, one thing you need to know before continuing... Changing the sprite's frame means copying the new image over the old one. Which means that it takes time to update. So if you have too many frames updated each turn, the game will slow down...

Animations

Simple Animations

Sprite animation has just become really easy... Now, you can just load a sprite, and tell PAlib to animate it, from one frame to another, and it will automatically loop. Just like the frame choice, you must *always* put all the images in the same sprite image file, one on top of the other. Here's the image used in this example :



I'll just post the interesting part of the code :

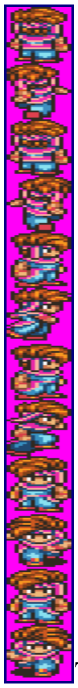
```
// Load the sprite palette,
PA_LoadSpritePal(0, // Screen
                0, // Palette number
                (void*)explosion_Pal);  // Palette name

// Here, we'll load a few similar sprites sprite to animate... at different speed
PA_CreateSprite(0, 0,(void*)explosion_Sprite, OBJ_SIZE_64X64,1, 0, 0, 64);
PA_CreateSprite(0, 1,(void*)explosion_Sprite, OBJ_SIZE_64X64,1, 0, 64, 64);
PA_CreateSprite(0, 2,(void*)explosion_Sprite, OBJ_SIZE_64X64,1, 0, 128, 64);
PA_CreateSprite(0, 3,(void*)explosion_Sprite, OBJ_SIZE_64X64,1, 0, 196, 64);

// Start the animation. Once started, it works on its own !
PA_StartSpriteAnim(0, // screen
                0, // sprite number
                0, // first frame is 0
                6, // last frame is 6, since we have 7 frames...
                5); // Speed, set to 5 frames per second
PA_StartSpriteAnim(0, 1, 0, 6, 15); // for the second one, speed of 15 fps...
PA_StartSpriteAnim(0, 2, 0, 6, 30); // for the third one, speed of 30 fps...
PA_StartSpriteAnim(0, 3, 0, 6, 60); // for the last one, speed of 60 fps...
```

Why does it load 4 sprites ? Just because I wanted to show how it does with 4 different animation speeds, that it... As you can see, the sprite loading is just a plain usual one, nothing special about it, as always... Next, the *PA_StartSpriteAnim(screen, sprite, first frame, last frame (included !), speed (in fps))* function. Once loaded, it animates the sprite just like you asked, at the given speed, and indefinitaly. You can use *PA_StopSpriteAnim(screen, sprite)* to stop it, or *PA_PauseSpriteAnim(screen, sprite, pause (1 to pause, 0 to unpause))* to pause/unpause the animation...

You can compile this example and test on DS !

Complex Animations

This second animation example and tutorial doesn't use more functions than the first one, but rather shows you how to use them better... It's the SpriteAnim2 example...


The image is similar to the frame tutorial one, but with more frames, to have the movements ! You can check it out in the source/gfx folder of the example. You'll notice that the sprite has some animations to move up, down, and right... but not left ! Why ? Because the left animation is a copy, flipped, of the right animation... So adding it to the image would take up more space for nothing, it's just easier to flip the sprite using the DS hardware... Saves up 25%, which is actually alot, considering that the graphics take up the major part of a DS rom...

On to the code we go !

```
while(1)
{
        // Animation code...
        if(Pad.Newpress.Up) PA_StartSpriteAnim(0, 0, 0, 3, 6);
        if(Pad.Newpress.Down) PA_StartSpriteAnim(0, 0, 8, 11, 6);

        if(Pad.Newpress.Right) {
                PA_StartSpriteAnim(0, 0, 4, 7, 6);
                PA_SetSpriteHflip(0, 0, 0);
        }
        if(Pad.Newpress.Left) {
                PA_StartSpriteAnim(0, 0, 4, 7, 6);
                PA_SetSpriteHflip(0, 0, 1);
        }

        if(!((Pad.Held.Left)||(Pad.Held.Up)||(Pad.Held.Down)||(Pad.Held.Right)))
PA_SpriteAnimPause(0, 0, 1);


        // Moving Code
        y += Pad.Held.Down - Pad.Held.Up;
        x += Pad.Held.Right - Pad.Held.Left;
        PA_SetSpriteXY(0, 0, x, y);
        PA_WaitForVBL();
}
```

The sprite is loaded like any sprite, so I didn't show it here...

if(Pad.Newpress.Up) PA_StartSpriteAnim(0, 0, 0, 3, 6); This means to start playing the animation from frame 0 to 3 when the key up is pressed... The fps is really slow : 6 frames per second ! Why ? Because the animation has very few frames, so it would be way too fast with a higher frame rate. This code is absolutely not perfect, as it doesn't give good results if you hold different directions at the same time, but it's just the simplest code I could do... if you need something better, you should now be able to do it by yourself

```
if(Pad.Newpress.Right) {
        PA_StartSpriteAnim(0, 0, 4, 7, 6);
        PA_SetSpriteHflip(0, 0, 0);
}
```

This part is different from the Pad Up/Down, because it checks if left or right to Flip the sprite horizontaly or not (remember, to save space in the rom...). That's why it has the sprite animation and the Hflip function in it...

I won't detail the moving code, it's the one we used before in the movement tutorial...

Compile and test !

Extended Animations

Latest addition to PAlib's animation system : extended animations ! What is that ??? It allows you to controle easily how the animations are played. When loading them with the AnimEx function instead of the plain Anim one, you have 2 extra options :

- *Animation Type*, ANIM_LOOP (0) being normal, and ANIM_UPDOWN (1) being a back and forth animation
- *Cycle Number*, being how many times you want it to be played. Setting it to -1 will make it play indefinitly... Concerning the ANIM_UPDOWN, a complete back and forth animation counts as 2 cycles...

As you can guess, the normal animation is just like the extended one, but set to a default value of infinite looping...

The code is a copy/paste of the explosion code, only the starting function changed (and not for all, lol).

```
// First animation will be normal
PA_StartSpriteAnim(0, // screen
                0, // sprite number
                0, // first frame is 0
                3, // last frame is 3, since we have 4 frames...
                5); // Speed, set to 5 frames per second

// Extended animations for the rest
PA_StartSpriteAnimEx(0, 1, 0, 3, 5, ANIM_ONESHOT); // just play it once...
PA_StartSpriteAnimEx(0, 2, 0, 3, 5, ANIM_UPDOWN, -1); // back and forth, infinite
number of times
PA_StartSpriteAnimEx(0, 3, 0, 3, 5, ANIM_LOOP, 5); // Play it 5 times
```

The ANIM_ONESHOT macro is there to say you want to play the animation just once. When it ends, it returns to the first frame... If you would like it to stay at the last frame, use the ANIM_UPDOWN type and 1 cycle...

Compile and enjoy !

Depth

Often, in 3D like games, you need sprite overlapping. People need to go in front other people, etcetera. In order to do this, you have to set priorities. There are 2 kinds of priorities on the DS :

- Sprite Priority : a sprite with a lower sprite number will be in front of a sprite with a higher sprite number
- Background Priority : To override the sprite priority, you can change the background priority. By default, all sprites are in front of background 0. You can set a sprite to be in front of a different background, 1, 2, or 3. A sprite on background 2 will be behind all sprites with a background priority of 0 or 1, and in front of all sprites with a priority of 3, all this regardless of their sprite numbers...

I won't post any code regarding this, just the function : *PA_SetSpritePrio(screen, sprite, priority);* That's pretty much all there is to know

— *Mollusk 29/11/2005 17:42*

Dual Sprites

One of the latest additions to PAlib has been the creation of the Dual Sprites functions. I must admit I was a bit against it, but it was so requested I ended up doing it anyways... It took me quite a long and boring time, because it was mainly just copy/paste and minor modifications to all the sprite functions.

Enough talk about me... What are these... Dual Sprites... ??? Well, they're a modified version of the sprites, which display can be displayed on both screens as though it was just one ! This doesn't mean a sprite will be displayed on the top and bottom screens at the same time, but that instead of having a vertical range of 0-191, it becomes 0-383, by using both screens as a single ! So you can have your sprites moving back and forth from one screen to another very easily. This is nice for games like pong, shoot'em up, etc...

The y = 0 point is the top-left point on the top screen. Pretty much all the functions we have already seen are available using PAlib's Dual Sprite system, just by adding the prefix *Dual* : *PA_DualCreateSprite(...)*, *PA_DualSetSpriteXY(sprite, x, y)*, etc... The only difference, besides the *Dual* prefix, is that these functions do NOT require you to set a screen number anymore... seems logical, since you consider them like just a single screen...

An example exists now. See %PALIBEXAMPLES%\Sprites\Basics\DualSprite\

Last thing, but it's important : you probably noticed that the DS's screens don't touch... they have a space in between them, and so I wondered how to manage that space in the Dual functions... My final descision was to go for a more flexible system, so I added a last function : *PA_SetScreenSpace(space in pixels)*. By default, this value is 48 pixels, it's the value I felt the most comfortable with... This means that when moving a sprite from one screen to another, it moves as though there were 48 pixels in between, during which you DO NOT see the sprite if it's too small... Of course, a 64×64 sprite will always show, as it's bigger than the 48 pixel space... If you want to do a pong-like game, you can leave that value to 48, or even put 64 to have a little more space. But if you are doing a shoot'em up, you might like to not have any space between the screens, so that NO objects can be *hidden* from the player. In that case, go for a 0, or 16, pixel space ; 0 will act as though the screens touched themselves...

DS Sprites Explained

I guess there's a small part that was missing here : how the sprite system works... Knowing this will help you decide how to organize your graphics and optimize your code...

What is a sprite ? It's not just a single entity, but a mixture of several things :

In 2D programming, it is usually referred to an 2D image or animation. The Nintendo DS has dedicated 2D hardware which makes it very easy to use these.

1. The image, in 'raw format', which is your Name_Sprite array, converted using PAGfx or gfx2gba, grit or anything else... That's basically the graphics, which will be copied in the VRAM (Video Ram), in a specific part reserved for sprites (1 part for top screen, 1 part for bottom screen). Several sprites can share the same graphic, but as it has only 1 copy in that case, animating 1 sprite will change the graphics in the VRAM and thus animate all sprites the same way...

2. The sprite palette, which is shared for all the sprites. PAlib, by default, has 16 palettes for the sprite, so you could have 16 sprites with different palettes, or 10 with the first and other with the 15 others, etc... But in lots of cases (especially when you start out), you just use a single palette for all the sprites...

3. The DS knows all about the sprites thanks to the OAM (Object Attribute Memory), which is a place in memory to handle all the sprites. This is a huge array containing the basic info for all sprites, one by one... (numbered 0 to 127, 2 copies existing, one for each screen...). The infos are like : X and Y position, shape/size (square/rectangle, 8 pixels wide or more...), horizontal or vertical flips, etc... One important info : rotation/zoom on/off, and Rotset number (0 to 31).

4. The rotset contains how the sprite using that rotset will be zoomed/rotated. As there are only 32, you cannot have 128 sprites rotated 128 different ways :/ So only 32 sprites are available to be affine transformed. Seems like a limit, but in fact it doesn't matter that much, you can most of the time find a decent solution : several sprites can share the same rotset if needed to be rotated/zoomed identically... And you won't find that you would have more than 32 sprites anyway for most projects.

AJ

FAQ - Sprites

Q1. My sprite doesn't show up !!

- Ok, a few things to check...
    - Have you loaded a palette to go with it ?
    - Is this palette on the same screen as your sprite ?

        - Do the palette and the sprite have the same palette number ?

Q2. My sprite has a background behind it!

- Check these couple of things
    - Look at question 1's answers
    - Make sure that the transparency colors are correct. If you're using Photoshop or any other 'professional' photo editing tool, the background color in some places can be off just by a minuscule amount. This is because these tools blend color in some places to reduce jagged edges, so you might get a tiny shade of the background color where you don't want it. The image converter is very picky about colors, so even if a pixel is just a tiny bit off, it'll judge is as a non-background color. This color difference might be totally unnoticeable to human eyes, so I prefer to go over the edges in a program like Paint (lol) which doesn't have anti-alias (edge blending). However if you still feel like using a professional tool, get Paint.NET and turn off Anti-alias for the tools.

<back to top>

<Day5>

Day 5 - Backgrounds

Here will be a tutorial for displaying backgrounds using PAlib

DS Background Specs

In the same way as we described the DS’s possibilities for sprites, we will now see what the DS can do concerning backgrounds... The possibilities are even more varied, maybe a bit too much :p I guess PAlib doesn’t really cover all the possible combinations, but rather the most common and useful ones...

- The DS can display 4 different backgrounds on each screen, each background having its own 256 colors palette. Every tiled background is composed of 2 parts :
    - A tileset, which is a set of 8×8 pixel ‘tiles’, blocks, that compose the global image
    - A tile map, which is a small map ranging from 256×256 pixels to 512×512, which describes how the tiles are aligned on the screen.
- Now, that’s true for tiled maps... But you can also replace a tiled map by an 8bit or 16bit drawable background, on which you can load a bitmap, a bmp, a gif, or even a jpeg, and be able to draw, plot pixels... However, always remember that this type of background is limited to 1 per screen, and takes up a lot of RAM (3/8th for 8bit, 6/8th for 16bit !!)

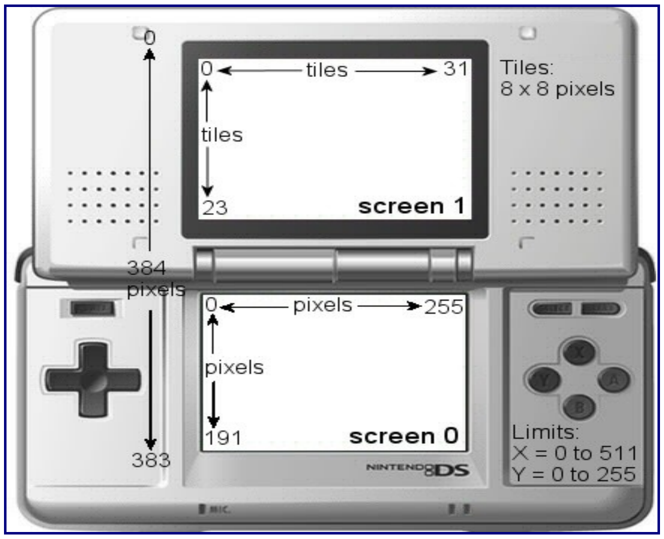
The DS can also display up to 2 rotating/scaling backgrounds per screens. These aren’t used that much, but they’re here for your pleasure only 😃

In this first background tutorial, we will just see how to use normal, plain tiled backgrounds, and you’ll see that it offers already pretty good possibilities...

Here’s a little scheme showing the DS screen sizes, with tiles and all :

DS Screen Sizes

Here’s the nice little scheme made by Bennyboo, which, once again, could be useful...



Just remember though what when you create your backgrounds, they are not 255 pixels or 191 pixels in size. They are 256×192, because 0 counts as the first pixel.

DS VRAM Sizes

In addition to making sure your backgrounds are the perfect size, you'll also have to keep an eye on the size of your backgrounds. The DS doesn't have an unlimited memory, and you'll probably have to manage your memory more than once. Very likely, you won't be able to have more than 3 backgrounds at once on each screen, so you'll have to do some fancy tricks in order to avoid problems like backgrounds not appearing right or being all corrupt. Here are some tips:

- Don't use all 256 colors. Even if your background uses less than 256 colors, the remaining colors are still in the palette and still take up valuable space in the Video RAM (VRAM). However, it is not guaranteed that PagFX won't yell at you for not having 256 colors. Furthermore, it can also lead to more problems in your program, because PAlib also doesn't guarantee non-256 color background support. Long story short, it's a risk to decrease the amount of colors in your palette, altough you might want to try it anyway, in case it works.
- Merge! I once had one BG and a second BG displayed at once on the same screen and then I noticed that they would do the same job if I merged them both in another image. I had bgHUD and bgStars, and now I have bgStarHUD, since I put them both in one background to save space.
- Backgrounds must be as small as possible. I guess that speaks for itself.

According to some tests I made, smartly conserving space in your VRAM could free up to 50% for the backgrounds, which means you can have twice the amount of backgrounds, or have them twice as large to use the same amount of space!

If these tips didn't work, here are some things to ask yourself:

- Is my background on the same BG number as the text?
- Is there already another BG and am I just overwriting it?
- Is my transparency correct?
- Is it just a bug in my emulator?
- Have I saved and converted the damn things?

Converting

PAGfx Frontend

Using backgrounds in the frontend is pretty much the same as the sprite usage... Just check the sprite tutorial. The only difference is that you need to click the Background tab to access the backgrounds... For the different available background modes, just refer to the following PAGfx.ini explanations...

PAGfx.ini

It is similar to sprites. The syntax is *imagefile BgMode palette*. If you do not put a palette name, it will use the background's name as palette and allow you to load it even more easily...

```
#TranspColor Magenta

#Sprites :

#Backgrounds :
C:\test.bmp EasyBg

#Textures :
```

The available background modes are :

- *8bit* for 8 bit bitmap image...
- *16bit* for 16 bit bitmap image...
- *TileBg* for a plain, normal, background, and what you'll most often use... Sizes are 256×256, 512×256, 256×512, 512×512.
- *LargeMap*, similar to TileBg, but using PAlib's LargeMap functions. It allows any background size you need, like 1024×2048, etc...
- *RotBg* for rotating backgrounds, with sizes 128×128, 256×256, 512×512, and 1024×1024
- *EasyBg* automatically selects the best format between TiledBg and LargeMap, so this is what we'll be using all the time :)

When converting, PAGfx should tell you how many tiles it uses, tell you which palette it uses, etc... Everything *should* be ok... On we go to background loading using PAlib !

Old TiledBg/LargeMap info, I'll redo that in an advanced user info

EasyBg : No Worries...

We'll use the easiest PAlib/PAGfx options to load our backgrounds... This is what you'll be using most in your code, so watch closely...

To convert in PAGfx, you’ll have to select the type as **EasyBg** (ok, true, TiledBg will work too ;))

Loading

I’m not sure it’s possible to make this much easier... If you have any ideas, tell me :)

PA_EasyBgLoad

Ok, this is easy, and uses the EasyBg option from PAGfx... So you won’t be able to use that with gfx2gba or other background converters, sorry :/ It’s arguments are as follows : *PA_EasyBgLoad (screen, background number, background name);*

The DS has 4 possible backgrounds per screen, so use a background number from 0 to 3... The smaller the number, the higher the priority (like for sprites...) → Background 0 is above Background 1, etc...

I’ll paste the complete code from PAlibExamples/Backgrounds/LoadBg, as it’s the first background tutorial...

```
// Load a simple background, very easy and simple...

// Includes
#include <PA9.h>       // Include for PA_Lib

// Converted using PAGfx
#include "gfx/all_gfx.c"
#include "gfx/all_gfx.h"


// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL

        // Load Backgrounds with their palettes !
        PA_EasyBgLoad(0, // screen
                        3, // background number (0-3)
                        bg0); // Background name, used by PAGfx...
        PA_EasyBgLoad(1, 0, bg0); // Same thing, but on the top screen...

        // Infinite loop to keep the program running
        while (1)
        {
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

This is really simple, loads the background with its palette on both screens... Nothing much to say about it, I guess. You can also check the LoadBgs example, which loads 4 backgrounds on a single screen...

Check the examples for PA_DualEasyBgLoad if you want a background that scrolls across both screens!

Unloading

If you want to get rid of a map (without putting another one in its place, as it will be unloaded automatically), use the following code:

```
PA_DeleteBg(screen, bg_number);
```

Where, of course, screen is the screen that the background is loaded to, and bg_number is the 0-3 background number on that screen for the map you're trying to get rid of. Pretty simple, eh?

Background Scrolling

Normal Scrolling

Scrolling a background is as easy as sprite moving was ! It uses a simple function, and the code example can be found in PAlibExamples/ScrollBg :

```
PA_InitText(1, 0); // Init text on the top screen, background 0...

// Load Backgrounds and their palettes...
PA_EasyBgLoad(0, 3, BG3);
PA_EasyBgLoad(1, 3, BG3);

s32 scrollx = 0; // No X scroll by default...
s32 scrolly = 0; // No Y scroll by default...

// Infinite loop to keep the program running
while (1)
{
        // We'll modify scrollx and scrolly according to the keys pressed
        scrollx += (Pad.Held.Left - Pad.Held.Right) * 4; // Move 4 pixels per press
        scrolly += (Pad.Held.Up - Pad.Held.Down) * 4; // Move 4 pixels per press

        // Scroll the background to scrollx, scrolly...
        PA_EasyBgScrollXY(0, // Screen
                        3, // Background number
                        scrollx, // X scroll
                        scrolly); // Y scroll

        // Display the X and Y scrolls :
        PA_OutputText(1, 0, 0, "x : %d   \ny : %d   ", scrollx, scrolly);

        PA_WaitForVBL();
}
```

The background is loaded as usual, and in this example text is initialised... We'll use it to display the background scrolling information. By the way, quick review of the text init function : you choose a background number on which to load the text... so that’s 0-3... If you load a background on a background used for text, it’ll screw up your text... and if you load a text over a background I used, it’ll erase your background... You cannot mix your own backgrounds and text...

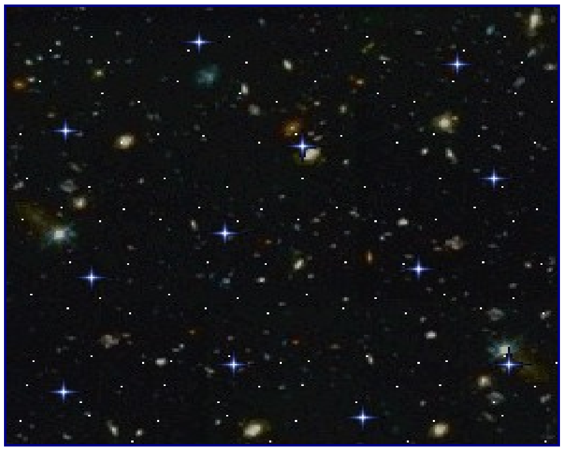
- *scrollx += (Pad.Held.Left - Pad.Held.Right) * 4;* This is pretty much like we did for the sprite, 4 being the speed ! (remember ?)
- *PA_EasyBgScrollXY(Screen, Background number, X scroll, Y scroll);* This is pretty self-explanatory, I guess... It scrolls the background TO a given position... So put 128, 128, and it’ll scroll the background to that position.
- The text output just displays the background current position...

Note that the following functions are also available : PA_EasyBgScrollX and PA_EasyBgScrollY, when you don’t need to scroll X and Y at the same time.

What happens when a background gets *off* the screen ? Depending on it’s size, it will wrap around or not... If your background is 256 wide, it’ll wrap around horizontally... If it’s between 256 and 512, it will not, though, because the DS default sizes are 256 or 512 large (using more selects a different scrolling mode, you can have more than 512 and it’ll wrap around nicely). When it doesn’t wrap around, it actually displays a few blank spaces until it falls back onto the background. Same thing vertically, it all depends on your background size...

Parallax Scrolling

Parallax Scrolling is a great technique that does a different scrolling speed on all background, giving an impression of depth and a sort of 3d effect... It looks good and is so easy to put in place it would be a shame not to use it ! It’s used in many games, from shoot’em up (scrolling the close stars at one speed, and the further ones at another) to plateform games (mario has the blocks at one speed, the moutains at another speed, and the clouds at still another one, and it looks perfect !), etc... This can be used in tons of situations. But I guess seeing the demo running will explain better than any words...



PAlib has some nice functions for parallax scrolling, one to initiate it, and one to use it... Check the Backgrounds/Parallax example for a space-like example...

```
// Load the 4 Backgrounds on the bottom screen...
PA_EasyBgLoad(0, 1, BG1);       //screen, background number, background name
PA_EasyBgLoad(0, 2, BG2);
PA_EasyBgLoad(0, 3, BG3);
```

```
// Initialise parallax vertically (Y axis) for both backgrounds
// 256 is normal speed, 128 half speed, 512 twice as fast...
PA_InitParallaxY(0, //screen
                                0, //Parallax speed for Background 0. 0 is no
parallax (will scroll independently with PA_EasyBgScrollXY)
                                256, // Normal speed for Bg1
                                192, //   3/4 speed
                                128); // Half speed
PA_InitParallaxY(1, 0, 256, 192, 128);  // Same thing, but for Screen 1...

s32 scroll = 0;


// Infinite loop to keep the program running
while (1)
{
        scroll += 1; // Scroll by one pixel...
        // Backgrounds with a parallax speed of 256 will scroll 1 pixel, 192 will
scroll 0.75, and 128 0.5
        // We could also have put a negative parallax speed to have some
backgrounds scroll in different directions

        PA_ParallaxScrollY(0, -scroll);  // Scroll the screen 0 backgrounds.

        PA_WaitForVBL();
}
```

First, you need to load a few backgrounds, at least 2... You just can't scroll at different speeds if you have only one !

Then, you have to init the parallax scroll : *PA_InitParallaxY(screen, bg0 speed, bg1 speed, bg2 speed, bg3 speed);*. You'll give a different speed to each background. Carefull, 256 means default 1 pixel speed, 512 means twice as fast, 128 half as fast, etc... You'll to do a little trial and error to find the speed that looks best in your game, I guess, but it's not hard. Normally, you put your main background, the closest one, to a speed of 256, and the further you go, the lower the value should be. If you do not want to parallax scroll a given background, just set its speed to 0... here, we did it for background 0... (you'll want to do the same if you use a background to display text, scores, or other stuff...)

scroll += 1; is just because it scrolls on its own, but we could have used the Pad keys to scroll it.

PA_ParallaxScrollY(screen, scroll); is the interesting function that does the actual parallax scrolling... here, we put -scroll because we wanted to scroll it on that specific direction...

Now, compile and test, it even works on DualiS ! You should see a nice parallax scrolling applied to a space background What do you think ? Nice, isn't it ?

Ok, here it scrolls vertically, while mario has a horizontal scrolling... This is easy to change. Had you noticed that all the parallax functions had as last letter *Y* ? Just remplace *Y* by *X* and you'll get a horizontal scrolling ! Yup, that easy... You could even combine X and Y scrolls if you want.

Rotating Backgrounds

Rotating backgrounds are cool but harder to use than normal backgrounds... To convert them, use the RotBg background mode...

The example I'll use is the one taken from Backgrounds/RotBackgrounds... I'll post almost all the code here

```
PA_SetVideoMode(0, 2);  //screen, mode

PA_LoadPAGfxRotBg(0, //screen
                  3, // background number
                  Rot, // background name in PAGfx
                  1); // wraparound !

PA_InitText(1, 0);

// Infinite loop to keep the program running
s32 scrollx = 0;
s32 scrolly = 0;
s32 rotcenterx = 0;
s32 rotcentery = 0;
s16 angle = 0;
s32 zoom = 256;

PA_OutputSimpleText(1, 2, 2, "Zoom       : Start/Select");
PA_OutputSimpleText(1, 2, 3, "ScrollX    : Left/Right");
PA_OutputSimpleText(1, 2, 4, "Scrolly    : Up/Down");
PA_OutputSimpleText(1, 2, 5, "RotCenterX : A/Y");
PA_OutputSimpleText(1, 2, 6, "RotCenterY : B/X");
PA_OutputSimpleText(1, 2, 7, "Angle      : R/L");

while (1){
        zoom += Pad.Held.Start - Pad.Held.Select;
        scrollx += Pad.Held.Right - Pad.Held.Left;
        scrolly += Pad.Held.Down - Pad.Held.Up;
        rotcenterx += Pad.Held.A - Pad.Held.Y;
        rotcentery += Pad.Held.B - Pad.Held.X;
        angle += Pad.Held.R - Pad.Held.L;

        PA_SetBgRot(0, 3, scrollx, scrolly, rotcenterx, rotcentery, angle, zoom);

        PA_WaitForVBL();
}
```

As you can see, there are 2 particular differences between Tiled Background loading and RotBackgrounds :

- *PA_SetVideoMode(screen, video);*, without which you wouldn't have the rotating background...
    - Video mode 0 sets the 4 backgrounds to normal tiled mode (or large map)... that's the default mode.
    - Video mode 1 sets the first 3 backgrounds to tiled mode (or large map...), and background 3 becomes a rotating background...
    - Video mode 2 sets the first 2 backgrounds to normal tiled mode... and you h ave 2 backgrounds for rotations (2 and 3).

That's pretty much all you need to know for video modes for now... Here, I set it to mode 2, but 1 could have done the trick. As you should have understood by now, rotating backgrounds cannot be put using any backgrounds, but only on 2 or 3, depending on the video mode...

- *PA_LoadPAGfxRotBg(screen, background number, background name in PAGfx, wraparound (0/1 for off/on));* is pretty similar to the other background loading functions... It loads the background and its palette, if converted with PAGfx...

Then come all the rotating/scalling/moving variables, they're pretty selfexplanatory, so I won't detail that... The last interesting part is :

- *PA_SetBgRot(screen, background, scrollx, scrolly, rotcenterx, rotcentery, angle, zoom);* It's kind of similar to sprite rotation/scaling, as it takes the same values for angles and zoom (256 zoom is 100%, etc...). You have more options though :
  - *scrollx* and *scrolly*, The X,Y offset for the rotation on the image. 0,0 is the top left corner of the image.
  - *rotcenterx* and *rotcentery* for the rotation center, which can be changed! This is where the image will be placed, NOT where the image rotation happens.
  - *angle*, just like sprites.
  - *zoom*, here doing both X and Y zooms, though it could be possible to seperate those if someone needed it... but I'll leave this as is for now.

I recommend just compiling and testing (on DS, won't work on emulator), because the keys are indicated on the screen to see how it moves arounds and rotates and all... Also, this background is set to wraparound, so it's *infinite*, you could try to put 0 to wraparound and see how it looks...

Some helpful functions:

```
float NDSRotToAng(float input){
        return ((input*100)/142);}

float AngToNDSRot(float input){
        return ((input*142)/100);}
```

8/16 bit Backgrounds

I'll cover 8bit and 16bit backgrounds in the same tutorial, as they are very similar, both in functionning and PAlib functions... What's the main difference between the 8/16bit backgrounds and the other background we just talked about ? Remember how the other backgrounds where made up of 8×8 pixel tiles ? That's not the case for these bitmaps backgrounds ! Nope, these are pixel by pixel drawable backgrounds...

8/16bit vs Tiles

I guess the best, before starting, would be to see the pros and cons of using these bitmap backgrounds... And *when* they can actually be very useful :

Cons

- They take up MUCH more space in video RAM than other background types... An 8bit one takes up 3/8th of the ram for a given screen, and a 16bit one 6/8th ! So if you use 16bit, you can put, at most, 2 other plain backgrounds (if they're not too complex...). Kind of limited...
- They are VERY slow to draw on... remember, I said that tiles were 8×8 blocks, while bitmaps are pixel per pixel... So it takes like 64 more time to change a 8×8 block in a bitmap background than in a tiled one... Plus, 8bit backgrounds have specific limitations that make it even slower than 16bit drawing pixel by pixel (but faster if you copy 2-4 pixels at a time, though)...
- Kind of the same, but erasing the whole background takes a long time, whereas it's pretty much instantanious for other backgrounds
- They can only be placed on background 3... It's not really a problem, as you can change the background priority, but remember that you can't load a background as background 3 after that, or else it'll bug (backgrounds 0-2 are available, though)

Pros

- You can actually draw directly on the screen ! PAlib provides function to draw using the stylus, for example (used in nDoS's paint app, for example), and other stuff could be done that way !
- You can load certain image formats directly on an 8bit or 16bit (depending on the format) background, without having to convert it ! Even though this is slow, it has the very cool fact that these images are often pretty small to include in your rom, so it's ideal for splash screens or static backgrounds that just need to be loaded once, and for which speed doesn't matter... The supported formats are *jpeg* (16bit only), *gif* (8/16bit), *bmp* (16bit), and *raw* (8/16bit).
- 16bit backgrounds are.. 16 bit... this means no palette limitation, and all colors possible on the screen ! That's pretty nice, used in conjunction with jpegs, for example

Final Thoughts

Let's see in what types of applications, or in which game phases, a 8/16bit background can be used...

- Splash screens are really a great use... You can use a jpeg and load it on a 16bit background, it'll look great and won't take much space in the rom...
- Menu screens can also have an 8 or 16bit background as a back screen (with jpeg, once again), and then you can use sprites for the menu items...
- Small 'video' shoots can be done with animated gifs on an 8 or 16bit screen ! This is really cool, as it doesn't take lots of space, but will only look good for cartoon-like animations, not movie-like ones...
- Ingame, there's no actual great use for bitmap backgrounds, unless you have a screen you want to draw on using the stylus...
- You could use the stylus drawing for signing your high scores, a bit like polarium does ! can be a nice little replacement to basic three letter names...

8 and 16 bit Background

Creating 8 bit and 16 bit backgrounds in PA_lib requires only a few lines of code. 8 bit backgrounds are a bit more complicated because they require the extra step of loading the palette but other then that 16 and 8 bit backgrounds are incredibly similar.

In order to create either an 8 bit or 16 bit background you first create the image in whatever drawing program you usually use. Make sure the image is 256 by 192 or else the image will appear skewed onscreen. Save the file. If it is a 16 bit background put the file name then16bit and no palette name. If it is 8 bit put the file name 8bit and then the palette name. Use the PAGfx converter and the backgrounds have been converted.

In order to use 8 bit backgrounds in your program you only need the lines…

```
 PA_Init8bitBg(0,3);
 PA_Load8bitBgPal(0,(void*)bmp_Pal);
 PA_Load8bitBitmap(0,background_Bitmap);
```

`PA_Init8bitBg(0,3);` initializes screen 0 to put an 8 bit background with a priority of 3. The priority can be anywhere from 0 – 3.
`PA_Load8bitBgPal(0,(void*)bmp_Pal);` loads the palette for the 8 bit bitmap you will be using.
Finally `PA_Load8bitBitmap(0,background_Bitmap);` puts the picture on the screen. 0 is the screen number and background_Bitmap is the file name followed by _Bitmap (assuming you used a bitmap for your image).

16 bit backgrounds are even easier to do and only require two lines…

```
PA_Init16bitBg(0, 3);
PA_Load16bitBitmap(0,background_Bitmap);
```

These lines are almost identical to those above and so their meaning should be pretty obvious. You do not need to load a palette because 16 bit images have no palette. — *dustin rhodes* *07/03/2006 00:34*

Loading a Jpeg

It is very easy. We have to use “PA_LoadJpeg()”

Syntax: PA_LoadJpeg(u8 screen, void *jpeg)

Is the same as PA_LoadGif. The only difference is you can’t put a jpg image in specific x,y. Example code:

#include <PA9.h> #include “intro.h” *Name of jpg image. Must be inside the project directory. int main () { PA_Init();* Inits Palib

```
PA_InitVBL(); //Inits VBL
PA_Init16bitBg(0, 3);   // Inits 16 bits background in bottom screen
PA_LoadJpeg(0, //bottom screen
            (void*)intro); //Jpg to show
 while(1)
  {
     PA_WaitForVBL();
  }
return 0;

}
```

It’s easy and practical.

Loading a Gif

Plain Gifs

I eat babies. Mmmm.

Loading a plain gif really couldn’t be easier. Take a look at the LoadGif example which comes with PALib.

You want to place your .gif file in the ‘data’ folder of your project, then you’re ready to go. Heres the example code:

```
// Include any GIF  and output it on an 8bit or 16bit screen !

// Includes
#include <PA9.h>       // Include for PA_Lib

#include "Mollusk.h" // gif to include
#include "trans.h" // gif to include

// Function: main()
int main(int argc, char ** argv){

        PA_Init();    // PA Init...
        PA_InitVBL();   // VBL Init...

        PA_Init8bitBg(0, 3);    // Init a 16 bit Bg on screen 0
        PA_Init8bitBg(1, 3);    // Init a 8 bit Bg on screen 1

        PA_LoadGif(     1, // Screen, which is 8 bit...
                                (void*)Mollusk); // Gif File

        PA_LoadGifXY(   0, 100, 60, // Screen, which is 16 bit, and at position 
100, 60
                                (void*)trans); // Gif File



  while(1)  {
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

I’ll walk through the code bit by bit.

```
#include "Mollusk.h" // gif to include
#include "trans.h" // gif to include
```

The files “Mollusk.h” and “trans.h” are created when you build the project, and are placed in the “build” folder automatically. Even though they aren’t there yet, make sure you include these files in your source code or it won’t work!

```
PA_Init8bitBg(0, 3);     // Init a 16 bit Bg on screen 0
PA_Init8bitBg(1, 3);     // Init a 8 bit Bg on screen 1
```

I’m pretty sure the first line is a typo and should instead be *PA_Init16bitBg(0, 3)* so you can ahead and change that line if you wish. The main point is you load a gif to a 16 bit background just as you would on an 8 bit background.

```
PA_LoadGif(     1, // Screen, which is 8 bit...
                                (void*)Mollusk); // Gif File

PA_LoadGifXY(   0, 100, 60, // Screen, which is 16 bit, and at position 100, 60
                                (void*)trans); // Gif File
```

Pretty self explanatory, just shows you can either load a gif as it is (in which case it will automatically be positioned at x,y coordinates 0,0) with *PA_LoadGif*, or you can give it custom coordinates by loading with *PA_LoadGifXY*.

One other thing to note is the *(void*)Mollusk* and *(void*)trans*, these are just the names of the gif files, which are in this case called *Mollusk.gif* and *trans.gif*, easy huh?

I hope this bit is ok, feel free to edit me if I have missed anything or gotten anything wrong.

<back to top>

<Day6>

Day 6 - Dev-related Math Stuff

All these have a 2 aims :

- Produce faster code
- Manage a specific situation (trajectories, gravity, collisions...)

Random Numbers

Random numbers are often required. They could be used for lotteries, AI, and power increases for level ups. I know what you are thinking, “How do I get random numbers?”.

PA_Rand()

Luckily, there is a method called *PA_Rand()*. You use it like this.

```
u32 num = PA_Rand();
```

This generates a random number alright, but with a major downside: it returns a HUGE number (definitely useless for level-up (unless your character is supposed to get super strong!)).

PA_RandMax(max)

Let’s say that you want a number between 0 and 4, inclusive.

Use *PA_RandMax*.

```
u32 max = 4;
u32 num = PA_RandMax(max);
```

PA_RandMinMax(min,max)

To take a number from 1-4, inclusive...

```
u32 min = 1;
u32 max = 4;
u32 num = PA_RandMinMax(min,max);
```

Fixed Point Math

If you try to use float’s or double’s on the DS you will see that they are very slow. But, you actually need decimal values in your game/app, so what are you going to do? The answer: using fixed point values! This part will show you how to manage positions, etc, using fixed point math, which should give a great speed boost to your games/applications.

Theory

Using a 32-bit variable (*s32*), we’ll ‘reserve’ the last 8 bits for an equivalent of the decimal part. So the first 24 bits will be for the integer part, and the last 8 bits for the decimal part.

As 8-bit values can go from 0 to 255, it means you have a precision of 1/256th, which is largely enough in most cases. 24-bits for the integer part will give you way enough values to manipulate large numbers.

In the end, in fixed point, a value of 256 means 1, 512 means 2, 1024 means 4, etc... So to do 0.5, you use... 128! You can use any intermediate value, like 196, which would correspond to 3/4th, etc... See, it’s not too hard!

Now, let’s see how to use it

Basic Example

Here’s the code, I’ll explain most of it. Its purpose is to show how you can used fixed point math to move a sprite by less than a pixel every frame.

```
s32 speed1 = 256;  // first speed...
PA_OutputText(0, 18, 2,  " 1   pixel/frame");

s32 speed2 = 128;
PA_OutputText(0, 18, 10, "0.5  pixel/frame");

s32 speed3 = 64;
PA_OutputText(0, 18, 18, "0.25 pixel/frame");

PA_LoadSpritePal(0, 0, (void*)sprite0_Pal);     // Palette name

PA_CreateSprite(0, 0, (void*)vaisseau_Sprite, OBJ_SIZE_32X32, 1, 0, 0, 0);
PA_CreateSprite(0, 1, (void*)vaisseau_Sprite, OBJ_SIZE_32X32, 1, 0, 0, 64);
PA_CreateSprite(0, 2, (void*)vaisseau_Sprite, OBJ_SIZE_32X32, 1, 0, 0, 128);

// all sprites stick to the left
s32 spritex1 = 0;       s32 spritex2 = 0;       s32 spritex3 = 0;


while(1) // Infinite loops
{
        // Move all the sprites by their corresponding speed
        spritex1 += speed1; spritex2 += speed2; spritex3 += speed3;

        // Position all the sprites, >>8 to return to normal position
        PA_SetSpriteX(0, 0, spritex1>>8);
        PA_SetSpriteX(0, 1, spritex2>>8);
        PA_SetSpriteX(0, 2, spritex3>>8);

        PA_WaitForVBL();
}
```

I will only comment the important parts of this code, and won’t go into repeating them when there 3 times the same.

- `s32 speed1 = 256;` is the first step in using fixed point math. As I pointed out earlier, our fixed point values will be normal values multiplied by 256... So to get a speed of 1 pixel per frame, you just set a speed of 1*256 = 256! That simple!
- `s32 speed2 = 128;` Ok, now, try to guess what speed, in pixels per frame, that would be...time up! 128 = 256/2, so, as 256 is 1 pixel per frame, 128 would be 1/2 pixel per frame, which is 0.5!
- `s32 speed2 = 64;` Same here, how much ? 1/4, so 0.25/frame.

By the way, why did we use *s32* variables? *s* for signed, because we could have negative values. And 32, because we need 32 bits: with like 8 bits, we would only have values up to 127, which is not even enough to move by 1 pixel per frame! With 32 bits, we’ll have plenty of values available.

That was easy! Now we need to see how to change the sprite's coordinate in fixed point, and the correct code is:

- `spritex1 += speed1;` Ok, that's just like all the code we've seen before, it has nothing to do with fixed point We just add the speed to the sprite's position.

Now, the real trick resides in how we position the sprite on the screen. To do so, we'll use a simple PA_SetSpriteX, because we don't want to touch at Y for now:

- `PA_SetSpriteX(0, 0, spritex1»8);` Oho !! what's this *» 8* ? It means `divide by 256`. Except that a division is really slow, while the *»8* is really fast. You don't need to know much more, actually, just that we divided the position in fixed point math by 256 ! And it works perfectly, as a position of x = 256 means x = 1, etc...

So in the end:

- To go from normal (0, 0.5, 1, etc...) to fixed point, just multiply by 256 (or do *«8*)
- To go from fixed point to normal (when you need to display on the screen), just divide by 256 (or do *»8*).

And that's it for the first tutorial!

Using Gravity (for jumps, etc...)

Requires that you have read the fixed point math tutorial... It's a very simple code, which can be used for any platform game... You have to understand a little fixed point math before understanding it fully, so re-read through the tutorial if you haven't understood everything...

The gravity code is nice, because it can be used in tons of games, and especially platform games, such as mario, etc... It relies on a simple basis : your sprite/mario/player/whatever has a vertical speed, and the gravity is an acceleration. Following this means 3 things :

- Each turn, the speed varies according to its acceleration
- Each turn, the sprite's position varies according to the speed
- At any time, if your player touches the ground (or is 'beneath' it), it should be put back over it, and it means that your speed should be null....

Because the acceleration and speed can have many values, and we don't want to be limited, we'll used fixed point values... That's why you have the bit shifts («8 and »8 everywhere in the code). This example is just a little shuttle going up, and falling back down. You can change the gravity and starting force (equivalent to jumping) to see how the shuttle's reaction changes...

```
#define FLOOR (160<<8) // Floor y level

int main(void){

        PA_Init(); //PAlib inits
        PA_InitVBL();

        PA_InitText(1, 0);

        PA_OutputText(1, 2, 4, "Press A to take off !");
        PA_OutputText(1, 2, 5, "Gravity change : Left/Right");
        PA_OutputText(1, 2, 7, "Takeoff Speed change : Up/Down");

        PA_LoadSpritePal(0, 0, (void*)sprite0_Pal)

        PA_CreateSprite(0, 0, (void*)vaisseau_Sprite, OBJ_SIZE_32X32, 1, 0, 50,
50);


        s32 gravity = 32; // change the gravity and check the result :)
        s32 velocity_y = 0;
        s32 spritey = FLOOR; // at the bottom
        s32 takeoffspeed = 1000; // Takeoff speed...

        while(1) // Infinite loops
        {

                takeoffspeed += (Pad.Held.Up - Pad.Held.Down)*8; // Change takeoff
speed...
                gravity += (Pad.Held.Right - Pad.Held.Left)*2; // Change gravity
speed...

                PA_OutputText(1, 4, 8, "Takeoff speed : %d   ", takeoffspeed);
                PA_OutputText(1, 4, 6, "Gravity       : %d   ", gravity);

                if((spritey >= FLOOR) && Pad.Newpress.A)   { // You can jump if not
in the air...
                        velocity_y = -takeoffspeed;  // Change the base speed to
see the result...
                }

                // Moves all the time...
                velocity_y += gravity; // Gravity...
                spritey += velocity_y; // Speed...

                if(spritey >= FLOOR) // Gets to the floor !
                {
                        velocity_y = 0;
                        spritey = FLOOR;
                }

                PA_OutputText(1, 0, 0, "Y : %d   \nVY : %d    ", spritey,
velocity_y);

                if (spritey>>8 > -32) PA_SetSpriteY(0, 0, spritey>>8); // show if
on screen
                else PA_SetSpriteY(0, 0, 192);

                PA_WaitForVBL();
        }

return 0;
}
```

Here comes the explaination :

- `#define FLOOR (160«8)` is the floor level. We used *«8* to convert it to fixed point... 160 is towards the bottom of the screen...
- `s32 gravity = 32;` is the default gravity we used. This is 100% random, lol, you could

use anything else. 32 being 256/8, it's equivalent to a 1/8 pixel/frame acceleration

- `s32 velocity_y = 0;` is the ship's default speed... By default, it's 0 ! (not moving up or down).
- `s32 spritey = FLOOR;` is the ship's default position, at the floor level...
- `s32 takeoffspeed = 1000;` is the ship's takeoff speed. This will determine, together with the gravity, up to what height the ship can go...

Next comes the basic stuff to change the gravity and takeoff speed, it's nothing much more than we've already seen :

- `takeoffspeed += (Pad.Held.Up - Pad.Held.Down)*8;` to change takeoff speed...
- `gravity += (Pad.Held.Right - Pad.Held.Left)*2;` to change gravity speed...

Now we'll get into the real important part of the code :

```
if((spritey >= FLOOR) && Pad.Newpress.A)    { // You can jump if not in the air...
        velocity_y = -takeoffspeed;  // Change the base speed to see the result...
}
```

This isn't too hard, it just means that if you press A while you are at the floor level (or below it), you take off ! To take off, just add the takeoff speed to the vertical velocity... Why is there a - there ? Because the takeoff speed was of 1000. But the higher you go, the lower *y* should be, so it has to be a negative value...

Next comes the code to adjust the speed according to the gravity, and the position... All this being in fixed point, of course :

```
velocity_y += gravity; // Gravity...
spritey += velocity_y; // Speed...
```

- First we adjust the speed by adding the gravity (the gravity should be positive, because it brings the ship down... So it's the opposite sign than the take off...)
- Then we adjust the position, according to the new velocity...

If you think of it, what will happen ? First you have a really high velocity, negative, when the takeoff just occured... Each frame, this gets lower and lower, because the gravity adds to it. At one points, it gets null, then changes sign, becoming positive... And then the sprite goes back down again, faster and faster... Just like in real life !

This is enough to have it work, *but* you shouldn't forget to add this :

```
if(spritey >= FLOOR) // Gets to the floor !
{
        velocity_y = 0;
        spritey = FLOOR;
}
```

If you don't, your ship will fall under the ground level ! With this, you tell it that if it goes beyond the floor, its speed should be null, and it should be put back at floor level...

And all you need to finish this is to show the sprite on the screen :

- `PA_SetSpriteY(0, 0, spritey»8);`, with *»8*, because we need to convert the fixed point back to normal integer position in order to display it !

And that's it for now ! Wasn't too hard, right ? Now, compile, and test (even in dualis) with different values of takeoffspeed and gravity, to see how it affects the ship.

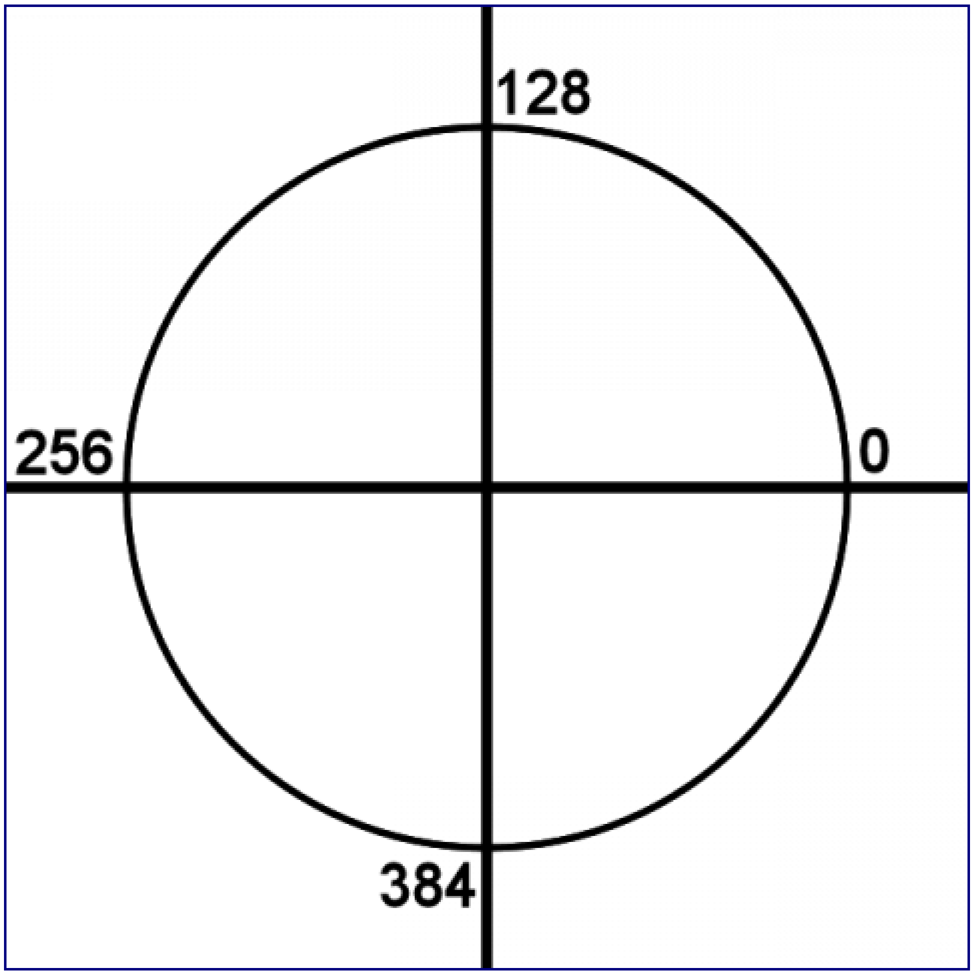
Trajectories and Angles

This section will be treated in 3 parts :

- First we'll see how the PAlib specific angles are, with a small and simple example
- Then a little theory on Cos and Sin functions
- And last, a concrete example of how to apply this in a game...

Angles in PAlib

The angles used in PAlib aren't normal degree angles, but angles adapted to the DS use. They range from 0 (right side) to 511, and are counter-clockwise. So angle 0 would be 3 o'clock, 128 would be 12 o'clock, 256 would be 9, etc... I think I already explained this in the sprite rotation tutorial, it's the same thing...



We'll see a pratical use of angles in the GetAngle code ! (Math/GetAngle/) It's a very simple example, which can be useful later on if you use the touchscreen... I'll just post the important code :

```
PA_OutputText(1, 5, 10, "Angle : %d  ", PA_GetAngle(128, 96, Stylus.X, Stylus.Y));
```

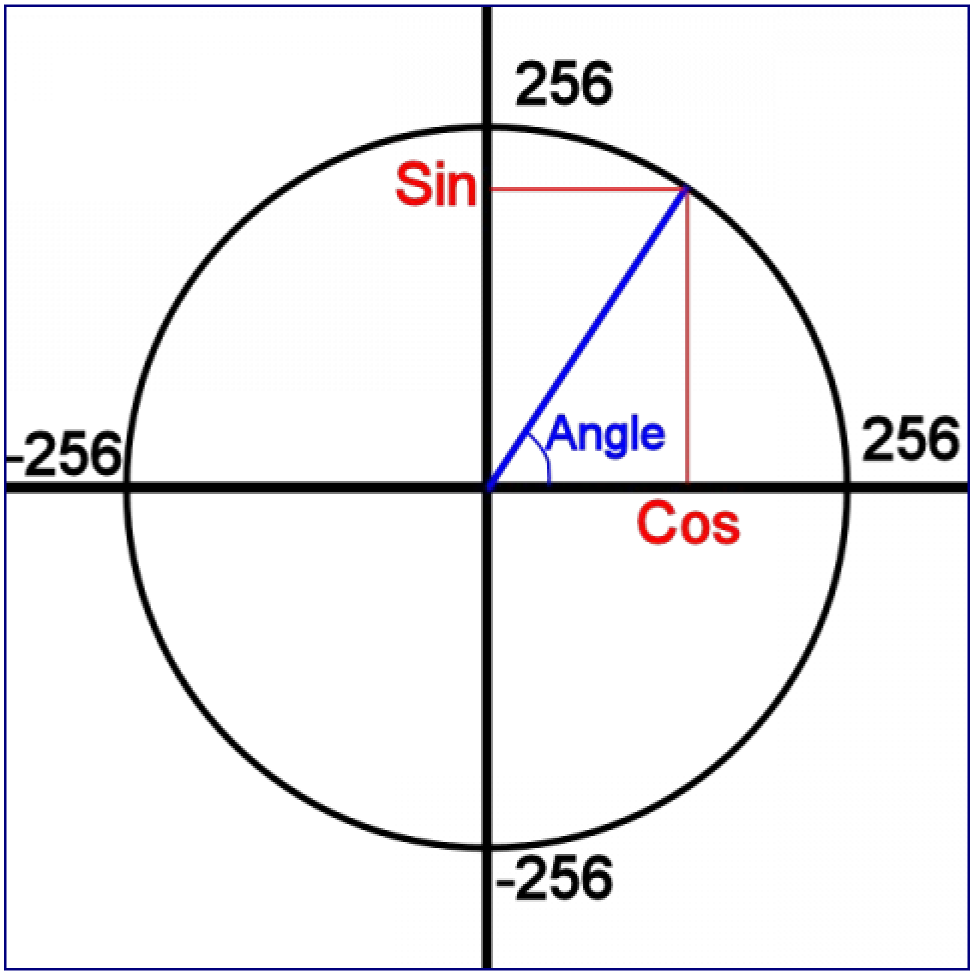
Lol, I swear that's the only important line... *PA_GetAngle(x1, y1, x2, y2)* is a simple function that returns the angle formed by a horizontal line and 2 given points... Most of the time, you'll want to use Stylus.X and Stylus.Y as one of the points... In this example, you can see we're taking the angle between the screen's center and the Stylus's position. This could be used in a game to get an angle and shoot from a given point on the screen, using the stylus to aim !

Now, off we go the the wonderful world of Sin and Cos, your new best friends when it comes to angles and movements...

Sin and Cos Basics

How to use PA_Sin and PA_Cos correctly... you need to have read the fixed point tutorial in order to use it perfectly...

These functions don't return a value between -1 and 1, but between -256 and 256. Why ? Because that's better for fixed point stuff, and that's pretty much what you should be using...



As you can see on this drawing, for a given angle, Cos is the horizontal coordinate of the point on the circle, and Sin the vertical coordinate... So using PA_Cos and PA_Sin, it should be possible to move a sprite according to a given angle, instead of only doing up/down and left/right ! That’s how we’ll be able to move a sprite according to a given trajectory !

As the PA_Cos and PA_Sin functions use fixed point math, you can see that the highest possible value is 256, and the lowest -256... Off you should go to see how to use this in a concrete ship-moving example !

Trajectory Example

This example is taken from PAlibExamples/Math/AngleSinCos, will be in the next PAlib update... What does it do ? It displays a ship in the center of the screen, turns it towards the left when you press left, right when you press right, and makes it move forwards/backwards when pressing up/down ! It shows how simple it can be to use angles in a game... All this using PA_Cos, PA_Sin, and fixed points.

Here’s the code :

```
PA_InitText(1,0); // On the top screen

PA_LoadSpritePal(0, 0, (void*)sprite0_Pal);

PA_CreateSprite(0, 0,(void*)vaisseau_Sprite, OBJ_SIZE_32X32,1, 0, 128-16, 96-
16); // Create the ship in the center...
PA_SetSpriteRotEnable(0,0,0);// Enable rotations and use Rotset 0...

s32 x = (128-16) << 8; // ship x position in 8bit fixed point
s32 y = (96-16) << 8; // Y
u16 angle = 0; // direction in which to move !

while(1)
{
	angle += Pad.Held.Left - Pad.Held.Right;
	PA_SetRotsetNoZoom(0, 0, angle); // Turn the ship in the correct direction

	if (Pad.Held.Up){ // Move forward
		x += PA_Cos(angle);
		y -= PA_Sin(angle);
	}
	if (Pad.Held.Down){ // Move backwards
		x += -PA_Cos(angle);
			y -= -PA_Sin(angle);
	}

	PA_OutputText(1, 5, 10, "Angle : %d  ", angle);

	PA_SetSpriteXY(0, 0, x>>8, y>>8); // Sprite position converted to normal...

	PA_WaitForVBL();
}
```

Here comes the comments... First we create a sprite and enable rotations for it... If this is unclear, please re-read the sprite rotation tutorial.

Then we initialize the variables we’ll need. As this is a basic example, all we need is x and y for the position, and angle for the angle... We could have used a structure, and have like ship.x, etc... but this is just a basic example

```
s32 x = (128-16) << 8; // ship x position in 8bit fixed point
s32 y = (96-16) << 8; // Y
u16 angle = 0; // direction in which to move !
```

As you can see, the angle has nothing special, but x and y are using fixed point (hence the *«8*).

Turning the ship is quite simple...

```
angle += Pad.Held.Left - Pad.Held.Right;
PA_SetRotsetNoZoom(0, 0, angle); // Turn the ship in the correct direction
```

- First you adapt the angle to the key press... That's like we have often seen. you might want the ship to turn faster, in which case just multiply all that by the chosen rotation speed (1 for normal, 2 for twice as fast, etc...)
- Then we just turn the ship according to it's angle, using the normal rotation function... here, I didn't use any zoom...

I guess the most important part of the code is the following :

```
if (Pad.Held.Up){ // Move forward

        x += PA_Cos(angle);
        y -= PA_Sin(angle);
}
```

This is what makes the ship move forward when you press up...

- For the *x* position, just add PA_Cos(angle). As said before, PA_Cos and PA_Sin are already suit for 8bit math, as they return values from -256 to 256 ! Nice, it makes things much easier now, hope you understand why...
- For *y*, there's a minus sign... Why ? because in a normal Sin function, when moving up, Sin should be positive... On the DS screen, the top of the screen has *y = 0* as value... so that's the opposite from the normal Sin !!! To counterbalance this, just add the negative sign...

The rest of the code is really easy. First, it's the same thing, but to move backwards... Second (and last), it's to move the sprite to the desired position.

Even though this code is really easy and doesn't do much, you could use and modify it to do more stuff :

- Variable turning speed
- Variable ship speed when moving forwards/backwards
- Adding bullets ! Now that you have seen how to use angles in a game, it's trivial to shoot and move the bullets !
- One nice variation I like, to use the DS's specs, is to change angle by the PA_GetAngle from the preceding example, and change Pad.Held.Up by Stylus.Held... What will it do ? It'll turn the ship in the stylus's direction and move it forward when you touch the screen ! Isn't that nice ?

I ended up adding a stylus example in the demos section : Following the Stylus

Scroll Grids

Scroll grids are easy to use. They basically alter a sprite's (BTW it doesn't even have to be a sprite!) position in pixels and can make it scroll on or off the screen depending on how far you scrolled in the first place. Lets say you scrolled 5 pixels left, so the scroll grid would automatically scroll your sprite five pixels right to give the impression of movement. In reality, *you* don't scroll, it's the sprite that does the moving.

```
if (Pad.Held.A) {
    scrollX += 5;
}
PA_SetSpriteX(screen, sprite, -(scrollX));
```

In this basic example of the scroll grid (you'd want to consider putting the scroll grid in a separate .c file once you're into object-oriented programming), scrollX is incremented by 5. This should move you 5 pixels to the right. To give the impression of movement, the scroll grid moves the sprite five pixels to the *left* instead. This can be done with backgrounds and such. Of course, we're assuming scrollX starts off a 0, otherwise you'd get the sprite jumping to negative values right away. But what if we start off with a value other than 0? That's OK, because if you get the scroll grid to run right away in the main loop, it should already scroll all of the objects for you once it runs for the first time. So, if we start with a scrollX of -100, the sprite should already be moved 100 pixels right if you put the code above in the main loop, or anywhere in the main function, as long as it runs before any use input. If I put the PA_SetSpriteX() function in an if(Stylus.Held) block, it would run only when I press the stylus to the screen, and then it would jump to its new position. Scroll grids also let you multiply or divide the values to make the sprite go faster or slower. If I modified the code to look like this:

```
if (Pad.Held.A) {
    scrollX += 5;
}
PA_SetSpriteX(screen, sprite, -(scrollX*2));
```

Then the sprite would move 2x faster every time I press A, or 10 pixels instead of 5. If you understood everything I said so far (you have every right not to), try to make a sprite scroll using the scroll grid in the demo FollowStylus. Actually, that's what I've been perfecting for the last week (just goes to show how complex scroll grids can be as well).

Scroll grids keep track of how far you have scrolled, but they do not have to move the sprite itself. The following code illustrates it better:

```
if (Pad.Held.A) {
    scrollX -= 5;
}
if (Pad.Held.X) {
    scrollX += 5;
}
if (scrollX <= -50) {
    PA_SetSpriteX(screen, sprite, -(scrollX));
}
```

If you scroll too much to the left, the sprite will not move anymore, which eliminates the aspect of motion. Since scrollX is still being decremented (we're decrementing it so that the sprite gets moved to the right and not off the screen as it's origin is 0,0), you can scroll as much as you want, but the sprite won't move anymore (although other sprites could). To move the sprite again, you have to increment scrollX until it's over -50 to have the sprite move leftwards.

I used the sprite just as a simple example to explain how scroll grids work. Moving sprites on the scroll grid won't really work in real life, because it will actually make it seem like you're moving the sprite itself. To make it more realistic, consider scrolling the background instead, and leaving the sprite at, say, the middle of the screen. This will give a better feeling of motion, and will keep you moving forever. My challenge for you today is to tweak the Follow Stylus example to scroll the background, and not move the sprite, to make it seem like *something* is moving.

A few points to keep in mind:

- To give the feeling of motion, decrementing scrollX moves the sprite to an <u>incremented</u> coordinate, that is, on a more positive one. Scrolling left (decrementing) moves the sprite right (incrementing) and vice versa. Same thing applies to the Y axis (scrollY, although it was never used).
- Scroll grids don't have to be in pixel-based values. They can be as small or as large as you want in contrast to single pixels (in my app, about 10 values are the equivalent of 1 pixel). This gets tricky when you're using a lot of formulas such as the PA_Distance function, which returns the square of what you actually want.
- You can multiply, divide, add, or subtract the scroll grid values to make it go slower, faster, or shift it in any direction.
- Scroll grids should run every frame.
- Technically, they *can* be all over the place, although you're better off putting the formulas for them in a separate file later on.

-Phaezon (martin.hanzel@live.ca)

Pathfinding Functions

PAlib has some pathfinding functions, it's really easy to use...

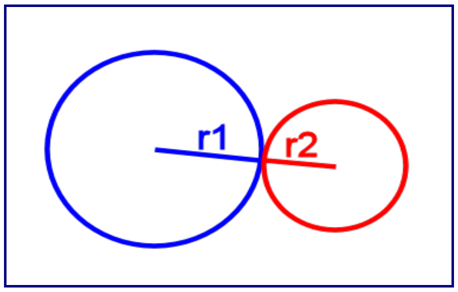
Sprite Collisions

We'll first see 2 kinds of sprite collisions : round sprites and square/rectangle sprites...

Round Sprites

Circular Collision Theory

If you consider sprites as round entities, it becomes really easy to make them collide correctly... Look at this small drawing, with 2 circles on it :



As you can see, these circles collide. At what distance are they ? Their centers are at exactly distance *r1* + *r2*... So if you calculate the distance between the circles' centers, they collide if the distance is less than *r1* + *r2*.

Yup, it's that simple to make them collide, so now we'll see this in a concrete example...

Circular Collision Code

This code can be found in Math/CollisionRound/, and is really simple.

It's based on 2 round sprites, identical, even though they could have different sizes... One of the circles (the centerd one) is fixed, the other can be moved using the stylus. And if they collide, it shows it on the top screen :

```
// This'll be the movable sprite...
PA_CreateSprite(0, 0,(void*)circle_Sprite, OBJ_SIZE_32X32,1, 0, 0, 0);
s32 x = 16; s32 y = 16; // Sprite's center position

// This will be the fixed circle
PA_CreateSprite(0, 1,(void*)circle_Sprite, OBJ_SIZE_32X32,1, 0, 128-16, 96-16);

while(1)
{
        if (PA_MoveSprite(0)){
                x = PA_MovedSprite.X;
                y = PA_MovedSprite.Y;
        }

        // Collision ?
        if (PA_Distance(x, y, 128, 96) < 32*32) PA_OutputText(1, 2, 10,
"Collision !!");
        else PA_OutputText(1, 2, 10, "             ");

        PA_WaitForVBL();
}
```

I won't comment all the code, as always... The first part just creates 2 sprites, numbered 0 and 1...

```
if (PA_MoveSprite(0)){
        x = PA_MovedSprite.X;
        y = PA_MovedSprite.Y;
}
```

As done before, if sprite 0 is moved, we'll memorize the X and Y coordinates of it's center. This will be used to calculate the distance between the 2 sprites...

```
if (PA_Distance(x, y, 128, 96) < 32*32) PA_OutputText(1, 2, 10, "Collision !!");
```

- *PA_Distance(x1, y1, x2, y2)* is a simple function that returns the square of a distance... Why not the real distance ? Because that would require using a square root, while the square is enough to estimate the distance, as we don't need a precise distance, but just enough to compare it...
- Why compare it to 32*32 ? 32 is the collision limit for the 2 circles : they both have a radius of 16 pixels, so 16 + 16 = 32... And since we need a squared distance, we multiply 32 by itself... All this is faster than taking the distance's square root...
- If the distance is indeed smaller than the limit, the 2 sprites are colliding, and we'll display it on the top screen...

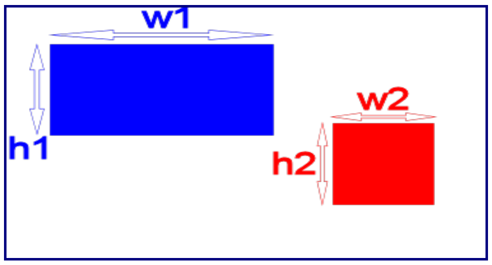
And that's it ! It was really simple.

You could think that circle collision isn't too applicable in a game. Actually, it's something I use really often... First example would be a shoot'em up... Your ship could be assimilated to a circle, as well as the bullets, so checking if you are hit would just be like checking a collision between 2 circles... fast and powerfull ! And I swear that most of the time the game goes so fast that you won't see it's not a pixel-perfect collision

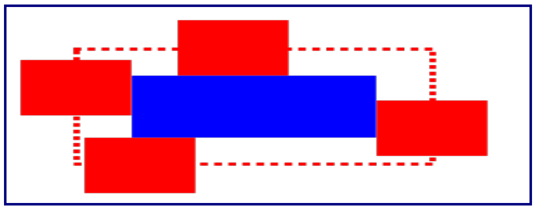
Rectangular Sprites

Reactangle Collision Theory

Rectangular collisions don't work quite the same way as circular ones, so I made 2 little drawings to illustrate how to each these :



As you can see on this first drawing, we have 2 different rectangles, each of them being defined by its width and height (w1 and h1, w2 and h2). We'll see with the next one how this can be used...



On this one, I've shown the 4 sides by which the red rectangle can enter in collision with the blue one... If you consider the red rectangle's center when it collides, you'll notice that it describes a perfect rectangle around the blue rectangle ! So all we need to know is this new rectangle's width and height... Look more closely... Yes, the new rectangle has a width of w1 + w2 and a height of h1 + h2, and is centered on the blue rectangle's center !!!

So if we add x1, y1 and x2, y2 as the respective rectangles' centers, we have the following equation for collisions : `(x2 >= x1 - (w1 + w2)/2) and (x2 ⇐ x1 + (w1 + w2)/2)` for the width and `(y2 >= y1 - (h1 + h2)/2) and (y2 ⇐ y1 + (h1 + h2)/2)` for the height ! And this should be enough to calculate if there's a collision...

Rectangle Collision Code

Now that we've seen the theory, the actual code should be fairly easy to you !

```
// This'll be the movable sprite...
PA_CreateSprite(0, 0,(void*)rect_Sprite, OBJ_SIZE_32X16,1, 0, 0, 0);
s32 x1 = 16; s32 y1 = 8; // Sprite's center position
s8 w1 = 32; s8 h1 = 16; // width and height...

// This will be the fixed circle
PA_CreateSprite(0, 1,(void*)rect_Sprite, OBJ_SIZE_16X32,1, 0, 128-8, 96-16);
s32 x2 = 128; s32 y2 = 96; // Sprite's center position
s8 w2 = 16;  s8 h2 = 32; // width and height...

while(1)
{
        if (PA_MoveSprite(0)){
                x1 = PA_MovedSprite.X;
                y1 = PA_MovedSprite.Y;
        }

        // Collision ?
        if ((x2 >= x1 - (w1 + w2)/2) && (x2 <= x1 + (w1 + w2)/2) && (y2 >= y1 - (h1
+ h2)/2) && (y2 <= y1 + (h1 + h2)/2)) PA_OutputText(1, 2, 10, "Collision !!");
        else PA_OutputText(1, 2, 10, "Collision !!");

        PA_WaitForVBL();
}
```

First, we create a rectangle of 32×16, and put in a variable it's position, width, and height...

```
PA_CreateSprite(0, 0,(void*)rect_Sprite, OBJ_SIZE_32X16,1, 0, 0, 0);
s32 x1 = 16; s32 y1 = 8; // Sprite's center position
s8 w1 = 32; s8 h1 = 16; // width and height...
```

Then, we'll do the same thing, but with a 16×32 sprite... No need to repost that code...

The moving around code is the same one as the previous code, so I won't detail it again...

And last, the actual collision code, which is just a copy/paste of what I wrote above : `if` 1) `PA_OutputText(1, 2, 10, “Collision !!”);`

And that's it !

Rectangular collisions can be used anywhere you didn't want to use circular collisions

note that the equation...

```
(x2 >= x1 - (w1 + w2)/2) && (x2 <= x1 + (w1 + w2)/2) && (y2 >= y1 - (h1 + h2)/2) &&
(y2 <= y1 + (h1 + h2)/2)
```

can be changed to...

```
(x2 >= x1 - ((w1 + w2)>>1)) && (x2 <= x1 + ((w1 + w2)>>1)) && (y2 >= y1 - ((h1 +
h2)>>1)) && (y2 <= y1 + ((h1 + h2)>>1))
```

hopefully this will run a bit faster because of the lack of division.

Use & instead of % when possible

The % operator works with two numbers, **a%b**, and returns the rest of a/b so, if you have 2%4 you get 2, 4%4 returns 0 and 5%4 returns 1. This operation uses a division and so it is pretty slow.

In certain conditions, you can use binary operations (which are infinitely faster to use), the only requeriment it's that b (from **a&b**) **<u>needs to be a power of 2</u>** , so you can replace x%2, x%4, x%8, etc... The only thing you must consider when using &, is that you don't put 'b', but 'b-1', i.e.

```
x%2;   ->   x&1;
x%16;  ->   x&15;
```

Using bit shifts (<< and >>)

Bit shifting is an efficient way of dividing and multiplying by powers of 2.

In order to divide by a power of two you bit shift to the right...

8>>1 = 4 (8/2^1 = 8/2 = 4)

8>>3 = 1 (8/2^3 = 8/8 = 1)

In order to multiply by a power of two you bit shift to the left...

2<<1 = 4 (2*2^1 = 2*2 = 4)

2<<3 = 16 (2*2^3 = 2*8 = 16)

Function Pointers

These are bloody weird to be frank, and bear in mind I'm only writing here what I currently understand on the subject, so it might be good if someone added what I missed.

OK! Well, for those of you who don't know what a pointer is, it is essentially a variable which rather than holding an ordinary value, it holds a memory location. They are defined like so:

```
s16 * intptr;      // pointer to a signed 16 bit variable
char * chrptr;     // pointer to a character variable
bool * bptr;   // pointer to a boolean variable
```

Note that these pointers do not “point” to anything yet. To make a pointer point to a variable, it must be initialized with the address of this variable. To do so you must use the reference operator “&”:

```
s16 intvalue;
char charvalue;
bool boolvalue;
intptr = &intvalue; // pointer is initialized with address of "intvalue" variable
chrptr = &charvalue;
bptr = &boolvalue;
```

Now these pointers can be “dereferenced” to access to the value of the variable they point to. This is done by placing an asterisk before the pointer name, like so:

```
*intptr = 16;  // the "intvalue" variable now contains the value 16
*chrptr = 'g';
*bptr = true;
```

If you were to output these pointers as they are, they would output the memory locations. They must be dereferenced to output the value they are actually pointing to:

```
//This isn't code to be run on the DS, this is just standard c++ code

cout << intptr << endl;  // prints a memory location, that look something like
0x28482873 etc etc
cout << *intptr << endl; // prints the value intptr was pointing to, which was 16
```

The main use for pointers is more for pointing to a variable which has been declared elsewhere in the program, which saves you having to make a copy of it in many cases:

```
char * pMychar;     // Create character pointer
char Mychar = 'a'; // Create character variable
pMychar = &Mychar; // Point character pointer to character variable

*pMychar = 'b'     // This will alter the value of Mychar
```

The first two lines should make sense, however *pMychar = &Mychar;* may make less sense. Placing a & before a variable name returns its memory location, so in this case I assigned the memory location of *Mychar* to *pMychar.*

The last line dereferences the pointer and assigns a new value to the variable it is pointing to. So in this case I have changed the value of *Mychar* from the pointer, rather than doing it directly.

Pointers also come in very useful for functions (This still is not Function pointers yet, that will come later):

```
// Define function to wrap a variables value
void wrapValue(s16 * pVal) {

   // wrap value if it is above 300,
   // so 300 becomes 0, 301 becomes 1...
   if(*pVal >= 300) *pVal -= 300;

}

// Here is how you would use the function
s16 myVal = 0;

myVal = 200;
wrapValue(&myVal); // In this case, variable doesn't wrap

myVal = 340;
wrapValue(&myVal); // This time it would wrap
```

I'll run through this and make sure you understand it. First the function:

```
// Define function to wrap a variables value
void wrapValue(s16 * pVal) {

   // wrap value if it is above 300,
   // so 300 becomes 0, 301 becomes 1...
   if(*pVal >= 300) *pVal -= 300;

}
```

As you can see, this function accepts a memory location of an s16 variable, and assigns it to an s16 pointer. It then checks to see if the value stored in the memory location is over 300, if it is then it reduces the variables value by 300, effectively creating a wrapping variable. Yes this function only wraps properly if the variable is under 600, but I'm just keeping things simple so we can focus on the pointers.

So basically pointers have allowed us to change values of variables which are **outside** our function. This is more efficient than accepting a normal s16 variable, because that requires the program to make a copy of the variable to be used within the function. Though you may have to do so when you want to accept a variable as an arguement and change that variable **without** changing variables outside the function.

Ok next bit:

```
// Here is how you would use the function
s16 myVal = 0;

myVal = 200;
wrapValue(&myVal); // In this case, variable doesn't wrap

myVal = 340;
wrapValue(&myVal); // This time it would wrap
```

So here I've tested the function on a couple of values. The first call would not actually do anything as the variable is lower than 300. The second call would reduce the variable to 40.

The only possible confusing part of this code is the *wrapValue(&myVal)*. If you remember in the function declaration, it accepted the **memory location** of an s16 variable, we therefore cannot give the variable *myVal* as an arguement on it's own, we have to give it's memory location. If you remember from before we do this by placing *&* before the variable name, so *&myVal* will give the memory location of *myVal* as an arguement.

Ok this is just about all you really need to know for now with pointers, if you want to know more try this site for more advanced pointers info.

Right, on to function pointers. They are still just variables, only this time they point to functions rather than other variables. Here is an example:

```
//This is not for the DS, just standard C++ (cout will not work on a DS i dont
think..)

void pointlessFunction(char myChar) {

   // Painfully simple function here...
   cout << "Arguement passed was " << myChar << endl;

}

// Here's the interesting part, defining a pointer to a function
void (*pt2Func)(char) = NULL; // Initialised to null

// Assign a function to the function pointer
pt2Func = &pointlessFunction;

// Now we call the function through a pointer
pt2Func("A");
```

Yeah I know, its really odd.. I'll walk you through it. I won't explain *pointlessFunction()* as that's self explanatory. So, defining a function pointer:

```
// Here's the interesting part, defining a pointer to a function
void (*pt2Func)(char) = NULL; // Initialised to null
```

Function pointers are defined in the following fashion: *returnType (*pointerName)(Arg1Type,Arg2Type,...)*

So say my function definition is:

```
int myFunction(char myChar, int myInt, bool myBool, char anotherChar)
```

Then I would create a pointer in the following way:

```
int (*pt2MyFunc)(char, int, bool, char);
```

Not too hard once you learn the formula. OK onwards:

```
// Assign a function to the function pointer
pt2Func = &pointlessFunction;
```

So here we pass the memory location of the function (using *&* as usual followed by the function name) to the pointer. Not much to it.

```
// Now we call the function through a pointer
pt2Func("A");
```

Hey presto you can use the pointer to call the function! Function pointers help us to avoid unneccessary Ifs, Elses, Switches, etc. So rather than having:

```
Switch(variable)
{
   case somecase: function1(val1, val2);
   case somecase2: function2(val1, val2);
   case somecase3: function3(val1, val2); //etc...
}
```

We can just create a function pointer and let it “specify” what function we are going to use. For more info on function pointers try this link.

Using a Division Table

if/else and for Optimising

if/else statements are expensive, so you want to reduce using them in trivial cases. Fortunately, you can abuse logic statements. For instance, instead of using the following code:

```
(if levelup==true)
 x++;  //add one to x
```

use the following code:

```
x+=(levelup==true);
```

for those that don’t understand the prefix, this stands for x= x+(levelup==true); see, the compiler evaluates levelup==true, if it’s true, it returns 1. which will add one to x. If levelup==false, then the statement will return 0, and add that to x. which does nothing!

Get the FPS

The following code keeps track of how many times the game loop executes each second. The DS’s video screen updates at 60Hz so the fps will be equal to 60 unless the code inside the game loop takes longer than 16.7ms to execute.

At the time of this writting emulators do not implement the DS timer so the code will only work in real hardware.

```
#include <PA9.h> //Include PA Lib

s8 fps = 0; //Our FPS
s8 time = 0;
s8 lastTime = 0;

int main(void)
{
   PA_Init();   //Initialize the main library
   PA_InitVBL(); // Initialize Vertical Blanking

   PA_InitText(1,2); //Init the text on both screens
   PA_InitText(0,2);
   PA_OutputText(1,0,0,"Starting...");

   lastTime = PA_RTC.Seconds;

   //We may start halfway into a second
   //let's wait until the clock starts the next second
   while(PA_RTC.Seconds == lastTime)
   {
      PA_WaitForVBL();
   }
   lastTime = PA_RTC.Seconds;

   while(1)
   {
      PA_WaitForVBL();   //Wait for Video Blank

      fps +=1;  //up the fps every time the code runs through

      time = PA_RTC.Seconds;  //Grab the time

      //If the second has changed we are done counting frames
      if(time != lastTime)
      {
         lastTime = time;  //Start looking for end of next second

         PA_ClearTextBg(0); //Clear the text so we don't get text mishaps
         PA_ClearTextBg(1);
         PA_OutputText(1,0,0,"Fps %d /60",fps);

         fps = 0;        //Start counting back at zero
      }
   }
}
```

<back to top>

<day7>

Day 7 - Sound

Most of the built in PALib functions have been replaced by the very snazzy ASlib by Noda. Check it out in PALibExamples! I'll keep the rest here for reference.

Today, we'll talk a bit about sound playing on the Nintendo DS. PAlib offers you 2 different sound output methods :

- The RAW format: Very much like playing a wav, which is perfect for special effects (pressing a button, gun shot, beep etc.). This format is completely unsuitable for music.
- The mod player (thanks to Deku!), which is perfect for music playback, because each file takes very little space in the ROM. MOD files are like MIDI files and are not like a voice recording. They are instead a list of notes and beats to be played.

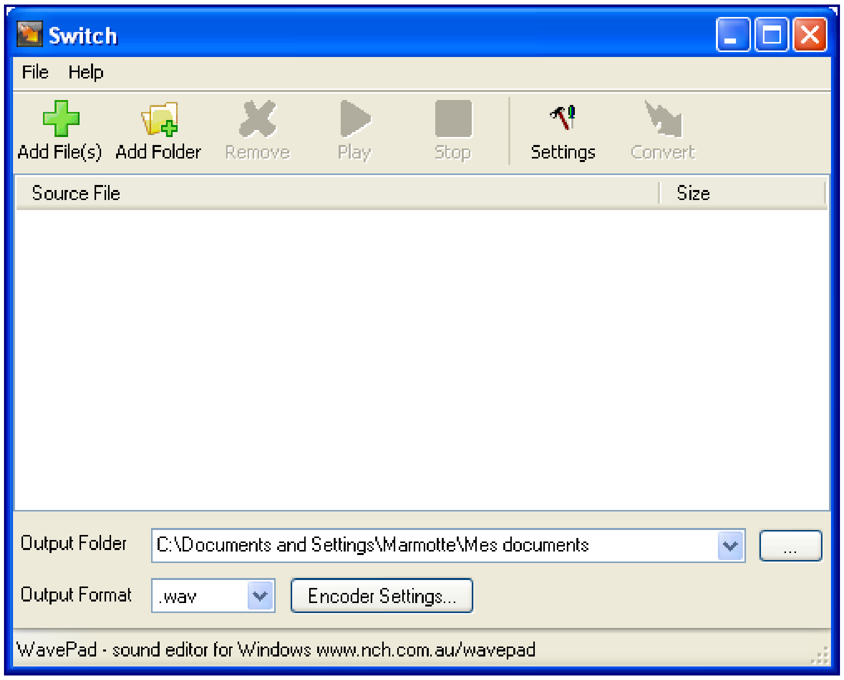
Raw Sound Files

The DS cannot play WAV or OGG files directly, so you have to convert them to RAW format. RAW Sounds are included inside the NDS ROM, so there is a limit of 4mb for resources. Converting a normal music track, for example of around 3 minutes will probably eat up all the space you have for your whole game! This is just the way PAlib works, if you want full length tracks then you will have to use EFS or the PAlib File System to add files after you have run build.bat and the NDS is created.

Converting Wav to Raw in Switch

I personally use and recommend Switch, because it works (and it's easy to use), but Kleevah would rather recommend Audacity, and some others use Sox (command line). For the purpose of this tutorial, I will use Switch, it is the easiest way to convert sounds to RAW format and requires to extra knowledge (unlike Audacity or Sox, if you can use Windows, you can use this).

After downloading and installing Switch (don't install all the other programs it suggests, they're good, but not needed for DS Development or PAlib), open it and you should get this :

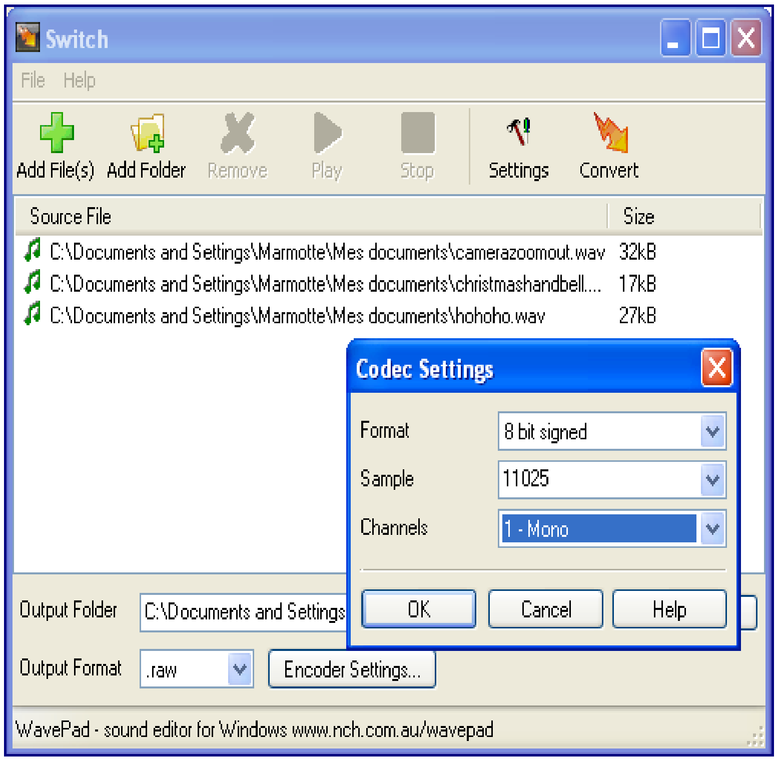

Switch
File Help
Add File(s) Add Folder Remove Play Stop Settings Convert
Source File Size

Output Folder: C:\Documents and Settings\Marmotte\Mes documents

Output Format: .wav

Encoder Settings...
WavePad - sound editor for Windows www.nch.com.au/wavepad

Now, click the Add File or Add Folder buttons to add files to you convert list. Once you have added all the sound files you want to convert, change the output format (bottom left) to .raw format, and change the encoder settings to have this:


Switch
File Help
Add File(s) Add Folder Remove Play Stop Settings Convert

Source File	Size
C:\Documents and Settings\Marmotte\Mes documents\camerazoomout.wav	32kB
C:\Documents and Settings\Marmotte\Mes documents\christmashandbell....	17kB
C:\Documents and Settings\Marmotte\Mes documents\hohoho.wav	27kB

Codec Settings

Format: 8 bit signed

Sample: 11025

Channels: 1 - Mono

Output Folder: C:\Documents and Settings

Output Format: .raw

Encoder Settings...
WavePad - sound editor for Windows www.nch.com.au/wavepad

If you are new to PAlib and just want working sound effects, there is no need to change those settings from what is shown above. You could leave the stereo output if you prefer, and also set something else than 11025 as sample rate (changes the speed of the RAW file, but it sounds bad) You **must** set the format to 8 bit signed if you want the converted file to work. These options give smaller files with decent quality, and I don't care to much about stereo sound (but it can used, like in Mario DS, there's the game where you open the boxes and the sound are different, from left to right, then right to left, etc.).

You can also set the output directory - You could either leave this as it is and copy the RAW files to your /data folder, or set the output directory to /data directly. Once you're ready, click the Convert button, and it's just a matter of seconds before you get your .raw files!!

Converting Most Formats to Raw in Audacity

In Audacity, converting is pretty much the same.

First, open up Audacity. Press Ctrl+P (or go to the Edit menu and click Preferences). Click the File Formats tab. There is a drop-down box called "Uncompressed Export Format." Click it and select other, then put in what we did for using Switch: "RAW (header-less)" and Signed 8 bit PCM. Click OK on that window and the next window.

Now, open the file you want to convert, then go to the File menu and click "Export As RAW" and you're all set!

Playing Raw Files

Now that you have your raw files, place then in your project's *data* directory. If you don't have one, just create it. The example we'll use is Sound/Sound...

You'll see the code is easy to understand :

```
#include <PA9.h>       // Include for PA_Lib
#include "saberoff.h"  // Include the sound (found in the data folder in .raw
format - for saberoff.raw use saberoff.h)

// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL
        PA_InitText(0, 0);

        PA_InitSound();  // Init the sound system

        PA_OutputSimpleText(0, 6, 10, "Press A to play the sound");

        // Infinite loop to keep the program running
        while (1)
        {               // Play the sound if A is pressed...
                if (Pad.Newpress.A) PA_PlaySimpleSound(saberoff);//here, too. Just
the name of the soundfile

                PA_WaitForVBL();
        }

        return 1;
}
```

There are just a few steps to follow:

1. *#include "saberoff.h"* includes your raw file, tells the compiler where it is. It's just *nameoftherawfile.h*, placed after the PAlib include.
2. *PA_InitSound();* initialises PAlib's sound system (both for raw and mod files), and sets the default raw file type to 11025 sample rate and 8 bit signed format.
3. *PA_PlaySimpleSound(soundfile);* This is easy to use, a soundfile...nothing more, nothing less.

Just compile, flash, and enjoy the light saber sound by pressing A!

Troubleshooting Tip - "PA_InitSound was not declared in this scope" or your 8 bit sound sounds mangled:

- For the build of PA_lib I'm using, PA_InitSound wasn't available, so I replaced the line:

```
  PA_InitSound();
```

with

```
  AS_Init(AS_MODE_16CH);
```

or as stated on the forum

```
  // Init the sound system
  AS_Init(AS_MODE_SURROUND | AS_MODE_16CH );
  AS_SetDefaultSettings(AS_PCM_8BIT, 11025, AS_SURROUND);
```

and everything works great! (Without the PA_InitSound call, the sound system wasn't being initialized properly.)

— Daft *09/10/2008 10:25 PM GMT*

Mod Files

Mod Files are really cool to use, because it's easy, takes very little space in the rom, and can sound pretty good. As they aren't too easy to create, it's better to search the web for some nice and free ones... I personnaly recommend checking out The Mod Archive, Exotica , and ModLand . Only take .mod files, though. Also be aware that while the .mods are free to download from these sources, the creators still hold the copyright so you cannot spread any software containing these .mods without permission from the author.

Now, on we go to the example: Sound/ModPlayer!

First, place your mod file in the data directory (I called the one in the example modfile.mod). If your template doesn't have a data directory, just create it.

Now, I'll post the whole code, for once:

```
// Includes
#include <PA9.h>       // Include for PA_Lib
#include "modfile.h"  // Include the mod file (the .mod file is in the data
directory)


// Function: main()
```

```
int main(int argc, char ** argv){

        PA_Init();    // PA Init...
        PA_InitVBL();   // VBL Init...

        PA_InitSound();         // Sound Init, for the mod player...

        PA_PlayMod(modfile);            // Play a given mod

        while(1){       // Infinite loop
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

There are just 3 things to know !

1. *#include "modfile.h"* must be placed in the top of your file, right after the PAlib include... This file has your modfile's name + *.h*. If you don't add it, the program won't know your modfile is there...
2. *PA_InitSound();* is the sound init used by PAlib... if you don't use it, the modplayer will not work
3. *PA_PlayMod(modfile);* plays a given modfile ! Yup, that easy !

Now, last thing : restrictions. Your mod file can have from 1 to 16 channels, but I recommend 8 channels, which leaves you 8 channels for sound effects.

Update: In the last versions of PAlib, the "PA_InitSound();" procedure is not used, so it may cause errors in compilation time, but a line containing MOD i.e: "ARM7_SELECTED = ARM7_MOD_DSWIFI" should be uncommented in the Makefile instead of the default one.

<back to top>

<day8>

Day 8 - DS Hardware

This page will reference all the PAlib functions which are specific to the DS, such as the lid state, user info, clock time, etc...

Date and Time

Getting the date and time is so simple it would be stupid not to learn it ! The example is the *Date_Time* one in PAlibExamples/Other, and only contains 2 lines...

```
PA_OutputText(1, 2, 10, "%02d/%02d/%02d", PA_RTC.Day, PA_RTC.Month,
PA_RTC.Year); // Date
PA_OutputText(1, 2, 12, "%02d:%02d  %02d seconds", PA_RTC.Hour, PA_RTC.Minutes,
PA_RTC.Seconds); // Time
```

You don't really care about the text output functions, but rather the variables in them... *PA_RTC* is a structure updated every frame which contains all the info about the current date and time the DS has. It has the following variables :

- *PA_RTC.Day*, from 1 to 31...
- *PA_RTC.Month*, from 1 to 12...
- *PA_RTC.Year*, from 00 (for 2000) to 99 (for 2099 I guess...)
- *PA_RTC.Hour*, from 0 to 23, for the current hour... 0-11 is AM, 12-23 is PM (just remove 12 to get the US PM hours. If the number is negative, it's AM, so add 12 hours. If positive, leave as is.)
- *PA_RTC.Minutes*, from 0 to 59... the minutes...
- *PA_RTC.Seconds*, from 0 to 59... the seconds...

I guess it was useless to detail everything, but I like having everything nice and clear...

User Info

The User Info isn't any harder to get ! Check the UserInfo example in PAlibExamples/Other... I won't post it here this time, sorry !

If you want to use the PA_UserInfo structure in your project you have to call PA_UpdateUserInfo(); before reading the struct, or else you would get empty values.

The User's Information is contained in PA_UserInfo, a structure with the following variables :

- *PA_UserInfo.Name*, the user's name, mine is Mollusk...
- *PA_UserInfo.BdayDay*, the user's birthday date, the day...
- *PA_UserInfo.BdayMonth*, the user's birthday date, the month...
- *PA_UserInfo.Language*, gives a number for each language:

```
|------> 0 Japanese
```

```
        1 English
        2 Français
        3 Deutsch
        4 Italian
        5 Spanish
```

- *PA_UserInfo.Message*, the message the user put on his DS...
- *PA_UserInfo.AlarmHour*, the hours of the DS Alarm Clock ! (from 0 to 23)
- *PA_UserInfo.AlarmMinute*, the minutes of the DS Alarm Clock...
- *PA_UserInfo.Color*, the user's selected color...

And... that's it !

Pause on Lid Close

I bet you already saw that the DS auto-pauses when you close it, right ? At least in commercial games... Would you like to add that feature to your game ?? It's easy, just a single function to add !!!

PA_CheckLid(); Checks the lid, and pauses if closed... Returns 1 when it unpauses to know that it has been used...

Where should you put that ? It is include in *PA_WaitForVBL();* so you dont have to worry! You can check the Other/CheckLid example in PAlibExamples to see how it works, test it on DS though or a emulator such as no$gba, which have Pause on Lid support, just minimize and maximize the emulator to test.

Screen Lights

Another hardware feature ? Screen lights ! You can activate/deactivate them at will... What for ? You can use it as a power saving feature and also to get a true black screen when you black out a screen for a prolonged period.

Check the Other/ScreenLight example...

```
if (Pad.Newpress.A) PA_SetScreenLight(0, 1); // Turn on bottom light
if (Pad.Newpress.B) PA_SetScreenLight(0, 0);
if (Pad.Newpress.X) PA_SetScreenLight(1, 1);// Turn on top light
if (Pad.Newpress.Y) PA_SetScreenLight(1, 0);
```

PA_SetScreenLight(screen, 1/0 for on/off);, it's as simple as that...

Names and Subnames

These are the names that are displayed before the game is started, such as in the menu screen of the R4 Revolution or in the icon for the DS game in the normal DS boot menu.

To change the names, open the makefile for your project with notepad/other editor of choice.

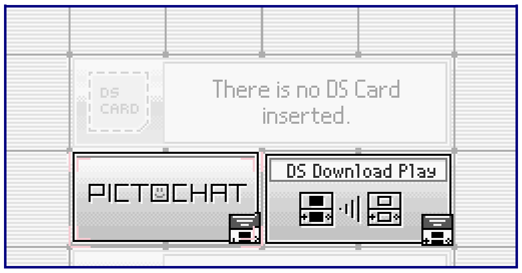
Find the following code:

```
TEXT1            := PAlib Project
TEXT2            := using PAlib
TEXT3            := www.palib.info
```

You can change TEXT1, TEXT2 and TEXT3 to whatever you need to, and it'll be applied when you remake the .nds/.ds.gba file.

Change the Game Icon

The game icon is what you see at the DS boot screen.



An example would be:

The game icon is a bit quirky, it needs to follow a specific format in order to work properly. It needs to be a 256 color 32×32 BMP file with up to 16 colors in its palette. The first color (The color of the upper left pixel of the icon) of the palette will always be transparent, so remember this when making the icon. The file is stored as logo.bmp in the /data/ directory of your project, on default. You can always change the filename you want it as, by simply specifying in the Makefile. The annoying thing about this is that if you have a complex icon, it will probably turn out bad and not look right on the DS.

For Mac OS X, a good choice of program to use to do this is Pixen, whilst The GIMP is a good choice for Linux and Paint.NET is a good choice for Windows.

<back to top>

<day9>
====== ANYTHING AFTER 'SIMPLE FILE LOADING' IS NO LONGER SUPPORTED OR INCLUDED IN PALIB ====== Use EFS_LIB instead.

How about a link to what EFS_LIB is and how to use it? –Lukas

Day 9 - FAT File System

Overview

First things first... what is a file system? Why would you need one?

Well, a file system isn't much, it's just a different way to add files to your rom. You currently know the basic, simple way : adding them at the compilation, either using a .c file, or using a file in the data directory (check out the gif/jpeg examples). But in fact, there's another way, by adding the files *after* the compilation. Yes, it's possible, and has quite a few advantages too :

- You can surpass the 4MB limit, because these files will not be stored in ram, but in a folder on your flashcard!... This also means that such files will work only if your flashcard supports DLDI.
- Because you can add files after the compilation, it means the end user can add files... This can be used for tons of applications, like a Mod Player, an image viewer, etc... nDoS uses such a system to store Outlook/Thunderbird contacts and read them... Many people have used the FAT system to add customisable games and such, a good example is BassAceGold's Dropfall and Mario Paint Composer, which both use FAT to load themes, and save/load music.

There are a few disadvantages, though...

- Usually the folder needs to be in root, which can tend to make your flashcard a bit messy (also see Folder Location)
- You may not want the end-user to edit your files, in which case you should use EFS instead.
- A bit of a hassle for the end-user, as they may forget to put the folder on the card, etc..

Simple File loading

Here's a small example to show the main FAT loading functions.

```
//Include the files we need
#include <fat.h> //Including libfat, which includes our general FAT functions..

#include <PA9.h> //PAlib inclusion

//Start our main function
int main(int argc, char ** argv)
{

PA_Init(); //Standard PAlib initialisation
PA_InitVBL();
PA_InitText(0, 0);  // Initialise the text system on the bottom screen
PA_InitText(1, 0);  // Initialise the text system on the top screen

if (!fatInitDefault) //Initialising FAT with default settings
{
   PA_OutputText(1, 1, 0, "FAT Init failed!/n Is your .nds patched with DLDI?"); //
We check if FAT init fails, if it has, output a message
}

PA_FatInitAllBuffers(); Initialise all the buffers

PA_FatSetBasePath("demo") //We set our FAT base folder as "demo". Note that it
starts from "/" (root), so in reality the base path is '/demo/'

// -------- FAT Loading Rules --------
// Sprites must go in the 'sprites' folder
// Backgrounds must go in the 'bg' folder
// Sounds are placed in the 'sfx' folder

PA_FatEasyBgLoad(0, 3, "stage"); //Just like PA_EasyBgLoad!

PA_FatEasyLoadSpritePal(0, 0, "characters/mario"); //We load it like usual, but
without the '(void*)' or '_Pal'. Don't forget the " "s!

PA_FatEasyCreateSprite(0,                          //Screen
                       0,                          //Sprite number
                       "characters/mario",         //Sprite name, without the
'_Sprite'
                       OBJ_SIZE_32X32,             //Size
                       1,                          //Colour type (256 colours)
                       0,                          //Palette number
                       32,                         //Sprite X
                       90                          //Sprite Y
                       );

// Infinite loop to keep the program running
while (1)
{
    if(Pad.Newpress.A) PA_FatEasyPlaySfx("mario/yahoo"); //Just like normal sound
playback
    if(Pad.Newpress.B) PA_FatEasyPlaySfx("mario/exclamation");
    if(Pad.Newpress.X) PA_FatEasyPlaySfx("mario/yipee");

    if(Pad.Newpress.Start)
    {
        PA_DeleteSprite(0, 0); //Like the non-FAT functions..
        PA_DeleteBg(0, 0);

        PA_PowerOff(); //Function to shut down the DS
    }

    PA_WaitForVBL(); Sync at 60 frames per second
}


} //End of main()
```

We start off with 2 text inits, one for each screen, and a FAT init with an error message if it fails. We also init all of the FAT buffers. Then, we set our base path, starting from to root of the flashcard. Afterwards, we load the sprites and backgrounds. Luckily for you, the functions are pretty much the same as the default ones you normally use. The only major difference that you should notice is the fact that the '(void*)' and '_Sprite' is missing, and instead there are two "s at each end of the path.

Note: When loading sprites and such, you should only include and folders/files that are *inside* the 'sprite' folder. So, when we load "characters/mario", the actual path of the file should be '/demo/sprites/characters/mario'. It is the same with backgrounds and sounds, but they have their own respective folder ('bg' for backgrounds and 'sfx' for sound files).

Using Directories

I guess it's not *that* usefull to have directories in a File System, but it can help you read the files later on... For example, you could have the user put his music in Files/Music, and his images files in Files/Images, etc...

As it's not so intuitive (sorry, did my best !), I'll explain in detail how to use them...

The PAlib Example is PAFS Folders... I won't detail all the PAFS functions we saw right before, so I hope you memorized all that already ! If not, just skim through it again and check some stuff if you're not sure...

This example is a bit more complicated than all the example we've seen before, but I'll do my best to make it easy. It contains 2 functions, the normal plain main function, and *WriteFolders*. Now, that's not really a function you'll use with PAFS, but it's used here to display the folders' informations...

The main code is exactly like the one we just saw, it just has *WriteFolders(0);* more in it. This functions reads the content (subfolders and files) of folder number 0... If you don't know how many folders there are, just check PA_FSSys→Nfolders, it'll tell you (this one counts the folders AND subfolders together...). Folder 0 is actually the *Files/* folder.

```
void WriteFolders(u16 N){ // Recursive function to write the number of folders

u16 Nfolder = N;
        // Write the folder's name...
PA_OutputText(1, indent, foldery, "%s : %d folders, %d files",
PA_FSFolder[Nfolder].Name, PA_FSFolder[Nfolder].NFolders,
PA_FSFolder[Nfolder].NFiles);
        foldery++;

        if (PA_FSFolder[Nfolder].NFolders > 0){ // For every subfolder, check their
subfolders
                indent+=2; // Indents for the subfolders
                s32 k;
                for (k = 0; k <  PA_FSFolder[Nfolder].NFolders; k++){
                        // Note : PA_FSFolder[i].FirstFile stores the number of the
first file,
                        //so the files go from
                        //PA_FSFolder[i].FirstFile to PA_FSFolder[i].FirstFile +
PA_FSFolder[i].NFiles
                        WriteFolders(PA_FSFolder[Nfolder].FirstFolder + k); //
Check the subfolders
                }
                indent -= 2;
        }
}
```

To stay as clear and simple as possible, I won't describe all the lines of code in this function. Just know that it outputs the folders and subfolders and different levels on the screen, to show which folder contains which others... All you need to understand is what is contained in the *PA_FSFolder* array of structure... It stores the different folder informations, and here are the variables :

- *PA_FSFolder[Nfolder].Name* is the folder's name, like `Files`
- *PA_FSFolder[Nfolder].NFolders* is the number of folders it contains...
- *PA_FSFolder[Nfolder].FirstFolder* is the number of the first folder it contains. So if you have 3 subfolders, and the first one is number 2, your 3 subfolders have numbers 2, 3, and 4...
- *PA_FSFolder[Nfolder].NFiles* is the number of files it contains, kind of like for the folders...
- *PA_FSFolder[Nfolder].FirstFile* is like for the folders, the number of the first file. Then you just add 1 to get to the next file, etc...

And there you go !

Just do like before, compile, use PAFS.bat, and check on your DS the result...

I won't detail more than that, and we'll move on to using PAFS on emulators (such as dualis) and WMB...

PAFS in Ram

I said at the beginning that PAFS stored the files in the rom instead of the ram, but you can force the files into ram... This will make it work on emulators, WMB, and any type of carts... Though it has a few limitations, the first one being the 4MB maximum size for the rom, and the second being that you have to specify in your code how much memory you want to give to the user... If you don't specify enough, it won't work...

The code is fairly simple, though there are 2 more things than in the normal plain PAFS code... Open PAFS_Ram and check it out !

```
// Includes
#include <PA9.h>        // Include for PA_Lib


PA_FSRam(100000);  /* This defines the size of the memory you allocate for files at
the maximum ... I set it to 100kb, here.
You can use more (like 2 megs, but not much more) or as little as a few KB if you
don't need to put much in it...*/


// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL

        PA_InitText(0, 0);  // Initialise the text system on the bottom screen
        PA_InitText(1, 0);  // Initialise the text system on the top screen
        PA_OutputText(1, 0, 0, "Loading PAFS...");

        PAFSStart = (char*)PA_FileSystem; // Tells to use the ram...
        u32 FileNumber = PA_FSRamInit(); // Inits PA Ram File System, and returns
the number of files
```

I posted only the beginning of the file and the PAFS Init, all the rest is identical to the other PAFS examples...

- First thing you'll notice is that we added something right after the PAlib include ! *PA_FSRam(100000);* is important, because it tells PAFS how much memory you give him. In his example, I allow only 100kb, which isn't much, because a mod file could be greater than that...
- By default, PAFS looks for your files in the rom... in order for it to look at the right place, you need to specify that it's in the ram... That's where *PAFSStart = (char*)PA_FileSystem;* comes in. It might not seem intuitive, but it means "PAFS starts in the ram...",i,,ğ

* And to finish, you'll see that the init is slightly different : *FSRamInit*. It's a special init to look for PAFS directly in the ram...

Yup, that it ! We've already seen all the rest, so you can compile, add the files, and test on dualis (that's what I do) to see if it really works

<back to top>

<day10>
The current version of PAlib does not render true 3D GFX. See the 3D Sprite examples.
<Back to top>

<day11>

Day 11 - Video Functions

This video stuff is really outdated. The best working and efficient way I have come accross is EMC: http://forum.palib.info/index.php?topic=3971

{the highlighted contains:http://forum.palib.info/index.php?topic=3971 }

{Nicoco73 and I have both been working on new movie formats for the DS, since no one seems to be able to understand Moonshell's pasta-code enough to put together a working DPG playback library (seriously, that source tree is a mess!). My offering is EMC, the Eponasoft Multimedia Codec. It is still a working prototype, but it is mostly functional.

http://www.eponasoft.com/EMC.zip

This zip file contains a demo application that uses the format, and a conversion tool to make EMC files. Both contain complete source code, the demo application is coded in straight C and requires PAlib, the conversion utility is coded in Visual Basic 6 with Service Pack 6 applied. It is released under a BSD-like license, the license details are included. Note that the conversion tool is basically a complex frontend for several other tools, which can be obtained from the URLs listed in the documentation.

There is a minor timing problem with the format, and I'm not yet sure if it's in the sampling size or somewhere in the demo code. Any assistance in hunting down this glitch is greatly appreciated.

PLEASE read the documentation before using, as it covers a great deal of information. I spent quite a long time writing it and tried to cover everything. If there's something not covered by the documentation though, feel free to bug me.

Also, there is one minor error in the playback function...on line 46 of the sourcecode, you will find the line

Code:

```
PA_LoadJpeg(0,pixBuf) ;
```

which needs to be changed to
Code:

```
PA_LoadJpeg(screen,pixBuf) ;
```

to function as documented. I discovered this bug while implementing the playback library in one of my many projects.

Finally, if you want to try it out right away, I made a conversion of one of my favorite music videos, "Nymphetamine" by Cradle Of Filth. Download here:

http://www.eponasoft.com/nymphetamine.rar

Rename the file to outfile.emc and put it in /dev/video so the demo program can find it.

Enjoy! 😌

}

Here’s a quick tutorial on how to make videos work in PAlib !

1)download viDeoconverterS v3 : viDeoconverterS3

2)modify the file convert.bat : modify Data/chevalier by the name of your .avi file. the parameter : -f is the fps(16 is good), -x and -y are the sizes of the video(256*192 is the size of your screen), -c is the compressage : GOOD, AVERAGE or BAD

3)you obtain one .h file and 1 or more .vid files

4)download the template : vidtemplate

5)Put your .vid files and your .h file *put them where?*

6)do build.bat and GBFSinclude.bat

7)you obtain your file.

how to use the videos in a homebrew?

use PA_InitGBFS(); and use PA_LoadMultiVidGBFS(s8 idfile); with the id of the .h file

you must to be in 16 bit mode to use the video : PA_Init16bitBg(1, 3); or PA_Init16bitBg(0, 3);

<back to top>

<day12>

Day 12 - Quick Demos

This part of the tutorial will contain a few small codes, either new or based on some of the other examples, which can help you a little more in DS developpement...

Following the Stylus

This is an example with a ship following the stylus. You could use this for a GTA 2D like movement system. You can look at the final version of this code in the PAlib Examples, under Demos/FollowStylus/. It simply shows a sprite following the stylus, turning towards it.

I'll only post and comment the difference with the other code Trajectory Example :

```
s32 x = (128) << 8; // ship x position in 8bit fixed point
s32 y = (96) << 8; // Y
u16 angle = 0; // direction in which to move !

while(1)
{
        angle = PA_GetAngle(x>>8, y>>8, Stylus.X, Stylus.Y);
        PA_SetRotsetNoZoom(0, 0, angle); // Turn the ship in the correct direction

        if (Stylus.Held){ // Move forward
                x += PA_Cos(angle);
                y -= PA_Sin(angle);
        }

        PA_OutputText(1, 5, 10, "Angle : %d  ", angle);

        PA_SetSpriteXY(0, 0, (x>>8)-16, (y>>8)-16); // Sprite position converted to
normal...

        PA_WaitForVBL();
}
```

- First off, *x* and *y* are now the sprite's central point, and not the sprite's upper left corner... why ? Because it gives a better result, you'll see...

- `angle = PA_GetAngle(x»8, y»8, Stylus.X, Stylus.Y);` This is completely different from the other angle code... This time, we get the angle using the sprite's center and the stylus (hence the *»8*, and using *x* and *y* as the center of the sprite...).
- `if (Stylus.Held){` was changed in order to move when the stylus is pressed...

And that's it ! It was a pretty simple demo, I must admit, but that's because it's the first one in here

Throwing and Bouncing Frisbees

This will be a tutorial in several steps... The end goal will be to have a nice demo with 10 frisbees flying around, thrown by the stylus, and colliding... We'll do this in 3 easy steps :

- First, displaying one frisbee flying around on both screens, thrown by the stylus
- Then, modifying that code to have 10 frisbees, using a structure array
- In the end, adding collisions to that code !

Part 1 - 1 Frisbee

Here comes a nice little example : it shows how you can use the stylus codes to catch a frisbee, throw it, and have it bounce on the walls (on both screens...). And have it turn on itself, just because I wanted it too...

You'll find this example in Demos/Frisbee in the next PAlib version (there was an old outdated one without stylus support)

I'll post a large chunk of code :

```
#define FRISBEE 10 // Sprite number...
#define SCREENHOLE 48 // Size of the space between the screens... This is what
looked the best


typedef struct{
        s16 x, y; // This is for the frisbee's position
        s16 vx, vy; // Frisbee speed
        s16 angle; // To make the frisbee turn on itself
}frisinfos;

frisinfos frisbee;  // Frisbee structure variable



int main(void)
{

// Initialise the lib...
PA_Init();
PA_InitVBL();
```

```
PA_InitText(1, 0);

// Load the palettes for the sprites on both screens
PA_DualLoadSpritePal(0, (void*)sprite0_Pal);

// Create the sprite on both screens...
PA_DualCreateSprite(FRISBEE, (void*)frisbee_Sprite, OBJ_SIZE_32X32, 1, 0, 96, 300);
// Bottom screen
PA_DualSetSpriteRotEnable(FRISBEE, 0); // Enable rotation/zoom, rotset 0

// Sprite initial position...
frisbee.x = 96+16;
frisbee.y = 300+16; // on the bottom screen

// Speed of frisbee in both ways
frisbee.vx = 0;
frisbee.vy = 0;

while(1)
{
        // Move with the stylus, or move on...
        if (PA_MoveSprite(FRISBEE)){
                frisbee.x = PA_MovedSprite.X;
                frisbee.y = PA_MovedSprite.Y + 192 + SCREENHOLE;
                frisbee.vx = PA_MovedSprite.Vx;         frisbee.vy =
PA_MovedSprite.Vy;
        }
        else{
                // Now, the frisbee's fixed point position will be updated
according to the speed...
                frisbee.x += frisbee.vx;
                frisbee.y += frisbee.vy;

                // If the sprite touches the left or right border, flip the
horizontal speed
                if ((frisbee.x -16 <= 0) && (frisbee.vx < 0)) frisbee.vx =
-frisbee.vx;
                else if ((frisbee.x + 16 >= 256)&&(frisbee.vx > 0)) frisbee.vx = -
frisbee.vx;

                // Same thing, for top and bottom limits...
                if ((frisbee.y -16 <= 0) && (frisbee.vy < 0)) frisbee.vy =
-frisbee.vy;
                else if ((frisbee.y + 16 >= 192 + 192 + SCREENHOLE)&& (frisbee.vy >
0)) frisbee.vy = - frisbee.vy;
                // The bottom limit is at the bottom of the bottom screen, so that
would be 2 screen heights, plus the space in between...
                PA_DualSetSpriteXY(FRISBEE, frisbee.x-16, frisbee.y-16);
        }

        PA_OutputText(1, 2, 10, "SpeedX : %d    ", frisbee.vx);
        PA_OutputText(1, 2, 11, "SpeedY : %d    ", frisbee.vy);

        frisbee.angle+=4; // Make the frisbee turn...
        PA_DualSetRotsetNoZoom(0, frisbee.angle);

        PA_WaitForVBL();  // Synch to the framerate...
        }
```

And now, the comments...

```
#define FRISBEE 10 // Sprite number...
#define SCREENHOLE 48 // Size of the space between the screens... This is what
looked the best
```

Are just 2 definitions that will make our life easier... *FRISBEE* is the frisbee's sprite number, and *SCREENHOLE* the space in between the screens, in pixels.

```
// Load the palettes for the sprites on both screens
PA_DualLoadSpritePal(0, (void*)sprite0_Pal);

// Create the sprite on both screens...
PA_DualCreateSprite(FRISBEE, (void*)frisbee_Sprite, OBJ_SIZE_32X32, 1, 0, 96, 300);
// Bottom screen
PA_DualSetSpriteRotEnable(FRISBEE, 0); // Enable rotation/zoom, rotset 0
```

Here we load the palette, create the sprite on both screens (hence the *Dual* prefix), and enable rotations... Have you noticed the sprite's Y coordinate ? *300*... That's on the bottom screen. In Dual Mode, PAlib considers the DS like having a single screen of 384+SCREENHOLE pixels, in our case 48 pixels... (which looks the best on DS, but is horrible on emulators...)

Then we initialise the frisbee structure values :

```
frisbee.x = 96+16;
frisbee.y = 300+16; // on the bottom screen

frisbee.vx = 0;
frisbee.vy = 0;
```

The position is set to 96, 300... *+16* because we'll use the sprite's center... and it's a 32×32 sprite. And the speed to 0... This does not use fixed point, but we could have...

Next comes one of the important parts of code...

```
if (PA_MoveSprite(FRISBEE)){
        frisbee.x = PA_MovedSprite.X;
        frisbee.y = PA_MovedSprite.Y + 192 + SCREENHOLE;
        frisbee.vx = PA_MovedSprite.Vx;         frisbee.vy = PA_MovedSprite.Vy;
}
```

This part of the code has multiple functions :

- *PA_MoveSprite(sprite number)* is an important PAlib function we've already seen. It checks if the stylus's touches the sprites, and moves it around according to the stylus position... If it does move the stylus, it will return 1... If that's the case, we have several things to do :
- Memorise the new X and Y coordinate of the frisbee... That's done by doing `frisbee.x = PA_MovedSprite.X;`.
    - Why *PA_MovedSprite.X* ? Because PA_MovedSprite.X returns the moved sprite's central point...
    - Why *PA_MovedSprite.Y + 192 + SCREENHOLE;* ?? Because the MoveSprite function returns the position for a sprite on the bottom screen, so that would be y being between 0 and 191... However, we are considering the 2 screens as only 1, so we have to convert that to the correct positiono... So we add *192* for the top screen, *SCREENHOLE* for the space inbetween the screens...
- Then we memorise the stylus's speed, horizontally and vertically (respectively *vx* and *vy*).

In the end, what you'll want to remember, is the PA_MovedSprite structure, which stores information on the sprite's position, as well as the current moving speed... So as you see, there actually is more the PA_MoveSprite than just moving the sprite around

Now, if the sprite wasn't touched, we go into a different loop, which starts with

```
else{
        // Now, the frisbee's fixed point position will be updated according to the
speed...
        frisbee.x += frisbee.vx;
        frisbee.y += frisbee.vy;
```

This bit of code moves the sprite according to the current speed... Nothing special about that

```
// If the sprite touches the left or right border, flip the horizontal speed
if ((frisbee.x -16 <= 0) && (frisbee.vx < 0)) frisbee.vx = -frisbee.vx;
else if ((frisbee.x + 16 >= 256)&&(frisbee.vx > 0)) frisbee.vx = - frisbee.vx;

// Same thing, for top and bottom limits...
if ((frisbee.y -16 <= 0) && (frisbee.vy < 0)) frisbee.vy = -frisbee.vy;
else if ((frisbee.y + 16 >= 192 + 192 + SCREENHOLE)&& (frisbee.vy > 0)) frisbee.vy
= - frisbee.vy
```

This is more important... It checks if the frisbee is moving out of the screen ! If that's the case, it'll change the speed to make the frisbee move in the correct direction... Concerning the Y position, you'll notice that we have to check with 192+192+SCREENHOLE, because that's the total height of both screens, taking into account the space...

```
        PA_DualSetSpriteXY(FRISBEE, frisbee.x-16, frisbee.y-16);
}
```

This ends the *else* loop... It just positions the sprite at the correct place. Notice the *Dual* prefix, to use both screens as 1... *-16* is there because x and y are the central position, and we need to give the top left corner position...

The end of the code is pretty trivial... First we show the current speed on the top screen, just to check it out. And then we make the frisbee turn, with

```
frisbee.angle+=4; // Make the frisbee turn...
PA_DualSetRotsetNoZoom(0, frisbee.angle);
```

It turns at a speed of 4 PAlib degrees per frame, which looks pretty good... I used DualSetRotset to make it turn on both screens...

And that it ! I recommend compiling and testing the code (even works on Dualis r11, except that the colors are wrong), and you'll see it was a pretty nice example...

Guess what ? For the next example, I'll do the exact same thing, but with 10 frisbees You'll see there are like 2 lines of code to add, using an array of structures...

Part 2 - 10 Frisbees

As said before, this should be fairly easy... I'll post most of the code, but won't comment the things we have already seen right before...

```
typedef struct{
        s16 x, y; // This is for the frisbee's position
        s16 vx, vy; // Frisbee speed
        s16 angle; // To make the frisbee turn on itself
}frisinfos;

frisinfos frisbee[10];  // 10 Frisbees !!



int main(void)
{

// Initialise the lib...
PA_Init();
PA_InitVBL();

PA_InitText(1, 0);

// Load the palettes for the sprites on both screens
PA_DualLoadSpritePal(0, (void*)sprite0_Pal);

s32 i; // will be used in for loops to cycle through the frisbees...

PA_InitRand(); // Init the random stuff...

for (i = 0; i < 10; i++){
        // Sprite initial position...
        frisbee[i].x = (PA_Rand()%256)-16; // random position on the screen
        frisbee[i].y = 192+SCREENHOLE + (PA_Rand()%192)-16; // random position on
the bottom screen;

        // Speed of frisbee in both ways
        frisbee[i].vx = 0;
        frisbee[i].vy = 0;

        frisbee[i].angle = 0;

        // Create the sprite on both screens...
        PA_DualCreateSprite(FRISBEE+i, (void*)frisbee_Sprite, OBJ_SIZE_32X32, 1, 0,
frisbee[i].x-16, frisbee[i].y-16);
        PA_DualSetSpriteRotEnable(FRISBEE+i, i); // Enable rotation/zoom, rotset 0
}

        while(1)
        {
                for (i = 0; i < 10; i++){
                        // Move with the stylus, or move on...
                        if (PA_MoveSprite(FRISBEE+i)){
                                frisbee[i].x = PA_MovedSprite.X;
                                frisbee[i].y = PA_MovedSprite.Y + 192 + SCREENHOLE;
                                frisbee[i].vx = PA_MovedSprite.Vx;
frisbee[i].vy = PA_MovedSprite.Vy;
                        }
                        else{
                                // Now, the frisbee's fixed point position will be
updated according to the speed...
                                frisbee[i].x += frisbee[i].vx;
                                frisbee[i].y += frisbee[i].vy;

                                // If the sprite touches the left or right border,
flip the horizontal speed
                                if ((frisbee[i].x - 16 <= 0) && (frisbee[i].vx <
0)) frisbee[i].vx = -frisbee[i].vx;
                                else if ((frisbee[i].x + 16 >= 256)&&(frisbee[i].vx
> 0)) frisbee[i].vx = - frisbee[i].vx;

                                // Same thing, for top and bottom limits...
                                if ((frisbee[i].y - 16 <= 0) && (frisbee[i].vy <
0)) frisbee[i].vy = -frisbee[i].vy;
                                else if ((frisbee[i].y + 16 >= 192 + 192 +
SCREENHOLE)&& (frisbee[i].vy > 0)) frisbee[i].vy = - frisbee[i].vy;
                                // The bottom limit is at the bottom of the bottom
screen, so that would be 2 screen heights, plus the space in between...
                                PA_DualSetSpriteXY(FRISBEE+i, frisbee[i].x-16,
frisbee[i].y-16);

                        }
                        frisbee[i].angle+=4; // Make the frisbee turn...
                        PA_DualSetRotsetNoZoom(i, frisbee[i].angle);
                }
```

Ok, as you may see, there are a few differences...

- First one : `frisinfos frisbee[10]; /`, this is now an array !! Yahoo ! Of a total size of 10, for 10 frisbees...
- Then comes the *s32 i;*, which will be used for the for loops...
- There's also *PA_InitRand();*, which isn't much, but means that on DS you should have the frisbees at different places each time you try it...

The frisbee init code has changed a bit, too :

```
for (i = 0; i < 10; i++){
        // Sprite initial position...
        frisbee[i].x = (PA_Rand()%256)-16; // random position on the screen
        frisbee[i].y = 192+SCREENHOLE + (PA_Rand()%192)-16; // random position on
the bottom screen;

        // Speed of frisbee in both ways
        frisbee[i].vx = 0;
        frisbee[i].vy = 0;

        frisbee[i].angle = 0;

        // Create the sprite on both screens...
        PA_DualCreateSprite(FRISBEE+i, (void*)frisbee_Sprite, OBJ_SIZE_32X32, 1, 0,
frisbee[i].x-16, frisbee[i].y-16);
        PA_DualSetSpriteRotEnable(FRISBEE+i, i); // Enable rotation/zoom, rotset 0
}
```

- This gives a random position on the screen, and creates the sprite at that position...
- Plus, notice the *FRISBEE+i* instead of the sprite number ? This means the first frisbee will have number *FRISBEE* (10), then FRISBEE+1 (11), etc... pretty cool, isn't it ?
- Same thing for the rotset, they each have one number (0-9)

- Instead of frisbee.x, you now have frisbee[i].x, as it’s an array... this accesses frisbee number i... You’ll see the same thing in the rest of the demo...

The last bit of code that changes is in the infinite loop...

- *for (i = 0; i < 10; i++){* was added to run the sequence for each sprite ! Yes, it’s THAT simple.
- The rest is strictly identical, just replaces frisbee by frisbee[i], to move all the frisbees 1 by 1...

And that’s all there is to know for this time ! Next tutorial to come up ? 10 frisbees with collisions !

Part 3 - 10 Frisbees Colliding

This example is just another quick modification of the Firsbee2 example, but adding a basic collision system so that the frisbees don’t run into each other... As it is the exact same code with only a few lines added at 1 part of it, I decided to only post the new code... I left in the DualSetSpriteXY(...) so that you see where to add it...

```
        PA_DualSetSpriteXY(FRISBEE+i, frisbee[i].x-16, frisbee[i].y-16);
}

u8 j;
for (j = 0; j < i; j++){ // Test collisions for all frisbees with a smaller
number...
        if (PA_Distance(frisbee[i].x, frisbee[i].y, frisbee[j].x, frisbee[j].y) <
32*32) {
                frisbee[i].vx = (frisbee[i].x - frisbee[j].x)/6;
                frisbee[i].vy = (frisbee[i].y - frisbee[j].y)/6;
                frisbee[j].vx = -frisbee[i].vx;
                frisbee[j].vy = -frisbee[i].vy;
        }
}
```

Yes, that’s all you need to have ALL the frisbees hitting each other pretty well !

The first thing that was needed was to check for collisions between all the frisbees... This is done with a for loop : `for (j = 0; j < i; j++)`. Have you noticed anything strange about this loop ? If it had been like the first for loop, it would be from 0 to 10... but here, it’s from 0 to... i ??? Why ?

- We are already in a for loop, which moves all frisbees, from 0 to 9 included. by adding another similar loop in it, we could test, for every frisbee, if it hits with frisbees 0 to 9... This would however lead to 2 problems :
    - It would test every frisbee collision twice : testing frisbees 0 vs 1, then 1 vs 0, for example... This is a huge waste of time.
    - It would test the collision with twice the same number : between 0 and 0, for example ! And as that’s the same frisbee, it would return a collision, and bug...
- So to avoid this, it tests the collisions only with the frisbees already moved at that frame. Even if it might not look intuitive, let’s write down the different tests :
    - Frisbee 0 : doesn’t do any test (exists if j < i, and i is 0, so exits before doing anything)
    - Frisbee 1 : tests only with frisbee 0 (only frisbee to be lower than 1 in number)
    - Frisbee 2 : tests with frisbees 0 and 1, which weren’t tested again frisbee 2, so that’s ok
    - Frisbee 3 : tests versus 0, 1, and 2, that’s perfect...
    - etc...

- So it works perfectly !

Next comes the collision check, taken from the Circular Collision tutorial (in the Math Dev stuff) : `if (PA_Distance(frisbee[i].x, frisbee[i].y, frisbee[j].x, frisbee[j].y) < 32*32)`, which checks if the distance between the 2 frisbees is compatible with a collision... if that's the case, it'll just modify the speed of both of the frisbees in collision For the first one :

```
frisbee[i].vx = (frisbee[i].x - frisbee[j].x)/6;
frisbee[i].vy = (frisbee[i].y - frisbee[j].y)/6;
```

The new speed is defined by the substraction of the frisbee's position. This guarantees that they will move appart from one another. I divided by 6 to have a nice speed, if not it would be way too fast... You can change the 6 to get a different speed when it collides...

```
frisbee[j].vx = -frisbee[i].vx;
frisbee[j].vy = -frisbee[i].vy;
```

This code gives the opposite direction to the second frisbee, nothing special about it...

And that's it ! It was really simple, wasn't it ? There is one flaw to this system, though... It doesn't come from the collision detection code, but from the new speed code... It does not take into account the speed at which the frisbees were moving. So even if they hit really hard or really slowly, they'll move apart at the same speed...

If you need more precise speed correction, I recommend that you check the next example, as it takes speed into account in a pretty effective way...

Puck Hitting

Here comes another demo ! This one is derived from the circular collision code, as well as the frisbee example... What does it do ? You have a Puck in the middle of the screen (just a blue circle, nothing special, but if anyone wants to change the graphics...), and you have your 'raquette' (another blue circle). When you hit the puck with the raquette, it sends the puck bouncing all over the place. The harder you hit, the faster it'll go.

Here's the code, you'll see it's almost nothing new... Most of it is just recycled code...

```
typedef struct{
        s16 x, y; // position
        s16 vx, vy; // speed
}puckinfos;

puckinfos puck;

#define SCREENHOLE 48

int main(void){

        PA_Init();
        PA_InitVBL();

        PA_InitText(1,0); // On the top screen

        PA_DualLoadSpritePal(0, (void*)sprite0_Pal);
        // This'll be the movable sprite...
        PA_CreateSprite(0, 0,(void*)circle_Sprite, OBJ_SIZE_32X32,1, 0, 16, 16);
        s32 x = 16; s32 y = 16; // Sprite's center position

        // This will be the hit circle
        PA_DualCreateSprite(1,(void*)circle_Sprite, OBJ_SIZE_32X32,1, 0, 128-16, 
96-16);
        puck.x = 128; puck.y = 96+192+SCREENHOLE; // central position on bottom 
screen
        puck.vx = 0; puck.vy = 0; // No speed


        while(1)
        {
                if (PA_MoveSprite(0)){
                        x = PA_MovedSprite.X;
                        y = PA_MovedSprite.Y;
                }

                // Collision ?
                if (PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE) < 32*32) {
                        // Collision, so we'l change the pucks speed to move it out 
of our 'raquette'
                        u16 angle = PA_GetAngle(x, y, puck.x, puck.y-192-
SCREENHOLE); // New direction angle
                        u16 speed = (32*32-PA_Distance(x, y, puck.x, puck.y-192-
SCREENHOLE))/32; // The closer they are, the harder the hit was...
                        puck.vx = (PA_Cos(angle)*speed)>>8;
                        puck.vy = -(PA_Sin(angle)*speed)>>8;
                }

                puck.x += puck.vx;
                puck.y += puck.vy;

                // If the sprite touches the left or right border, flip the 
horizontal speed
                if ((puck.x -16 <= 0) && (puck.vx < 0)) puck.vx = -puck.vx;
                else if ((puck.x + 16 >= 256)&&(puck.vx > 0)) puck.vx = - puck.vx;

                // Same thing, for top and bottom limits...
                if ((puck.y -16 <= 0) && (puck.vy < 0)) puck.vy = -puck.vy;
                else if ((puck.y + 16 >= 192 + 192 + SCREENHOLE)&& (puck.vy > 0)) 
puck.vy = - puck.vy;
                // The bottom limit is at the bottom of the bottom screen, so that 
would be 2 screen heights, plus the space in between...
                PA_DualSetSpriteXY(1, puck.x-16, puck.y-16);



                PA_WaitForVBL();
        }
        return 0;
}
```

And the comments...

```
typedef struct{
        s16 x, y; // position
        s16 vx, vy; // speed
}puckinfos;

puckinfos puck;
```

Classic definition we used over and over again, to store the puck's position and speed... In this example, we haven't used fixed point, though we could have.

Then the palette is loaded, the first sprite too, and then

```
PA_DualCreateSprite(1,(void*)circle_Sprite, OBJ_SIZE_32X32,1, 0, 128-16, 96-16);
puck.x = 128; puck.y = 96+192+SCREENHOLE; // central position on bottom screen
puck.vx = 0; puck.vy = 0; // No speed
```

This loads the puck for both screens, and sets its current position and speed...

Then comes the classic

```
if (PA_MoveSprite(0)){
        x = PA_MovedSprite.X;
        y = PA_MovedSprite.Y;
}
```

Just to store the raquette's position (central point)...

The next part is the only new code... It's what is used to hit the puck with a different speed depending on the way you hit it, and send it in the correct direction...

```
if (PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE) < 32*32) {
        // Collision, so we'l change the pucks speed to move it out of our
'raquette'
        u16 angle = PA_GetAngle(x, y, puck.x, puck.y-192-SCREENHOLE); // New
direction angle
        u16 speed = (32*32-PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE))/32; //
The closer they are, the harder the hit was...
        puck.vx = (PA_Cos(angle)*speed)>>8;
        puck.vy = -(PA_Sin(angle)*speed)>>8;
}
```

- `if (PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE) < 32*32)` Check if there's a collision, like said before... 32 is the distance between the 2 circles (must be change if you change the size of one of the circles), and 32×32 because we use the squared distance, for speed issues...
- We used *puck.y-192-SCREENHOLE*, because the coordinate is for 2 screens in height for the puck, while the raquette is only on the bottom screen.... So ywe have to substract the size of the top screen (192) and the screen space (SCREENHOLE)...
- `u16 angle = PA_GetAngle(x, y, puck.x, puck.y-192-SCREENHOLE);` is to get the angle formed with the puck's and the raquette's centers... What for ? You can draw this on a paper, and you'll see that this angle is the direction in which the puck should go !! Yup ! I don't have the time to do another drawing for this, but if someone ever does one, I'll add it here...
- `u16 speed = (32*32-PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE))/32;` is the second very important line, as it will determine the puck's speed ! How does it work ? The closer you are from the puck when it hits, the faster you were moving... why ? Because if you touch the puck by only 1 pixel, it means you were moving very slowly...

But if you touch the puck by like 16 pixels ? It means that before the frame was updated, you had the time to move by at least 16 pixels, which is 16 times faster than the 1 pixel hit...

- `32×32` is the square of the distance needed to hit. If you had a collision by just 0 pixels, you didn’t hit the puck, since your speed wasn’t high enough...
- Then the distance (`PA_Distance(x, y, puck.x, puck.y-192-SCREENHOLE)`) is recalculated, and substracted to the squared distance... This will give a higher speed when you move fast...
- And at last, this speed is divided by 32... why ? because it was just trial and error, lol... YOu can change 32 to have higher or lower speeds, and thus adjust it to your need...

- `puck.vx = (PA_Cos(angle)*speed)»8;` determines the horizontal (*vx*) speed of the puck, with the new hit. This speed is found by using the Cos of the angle and the global speed it will have... It’s just like the Trajectory tutorial, but with the speed added in...
  - But in that case, why is there a `»8` ??? Because, as said at the beginning, I haven’t used fixed point math in this example. And PA_Cos returns a number in fixed point math, so I had to convert it to a normal number again...
  - Then why is the `»8` after the speed, and not right after the Cos ?? Because if done after the PA_Cos, as it’s an integer, the result would always be 0 ! When done after the multiplication with the speed, the total value before division is superior to 256, so dividing by 256 (which is the same as `»8`) will not result in given you 0, but rather the correct speed in pixels, rounded down (a true speed of 2.5 will give you 2, etc...).
- The same then applies to the vertical speed, with a negative sign because it’s Sin... (and Sin is bad, negative, lol... you must not Sin !)

And that’s about it ! I won’t comment the rest of the code, as it’s just a copy/paste of the frisbee code (I just replaced ‘frisbee’ by ‘puck’).

Hope this was clear enough ! Enjoy

<back to top>

<day 13>

Day 13 - Platform Game

This is going to be a pretty dense tutorial, and it will most likely need for you to have read most, if not all, of the other tutorials/demos... You must have knowledge on sprites, backgrounds, collisions, animation, and ALL the Dev-related Math stuff before starting it out...

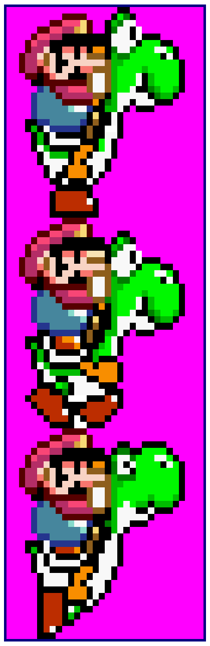
It will show the basics of starting your platform game, using mario ! Hehe, he’s just everywhere. As always, this project will be built in several steps...

1. Animating mario, moving him on the screen, having him jump around...
2. Adding a simple background, without scrolling, and having mario move on it.
3. Adding background scrolling, with parallax scrolling

Step 1 - Moving, Jumping...

This is going to be easy, as it just re-uses stuff we have already seen... I'll post the complete code in here. For once, I decided to cut the code into several functions. So I'll start out with the declarations and the main function, then move on to the second function :

First, we'll look at mario's simple graphics :



As you can see, there are only 3 frames. The first 2 are for the walking animation, and the third one is for jumping... Pretty simple...

Now, here comes the first chunk of code, taken from Demos/Platfrom :

```
// Includes, only one sprite
#include <PA9.h>


// PAGfxConverter Include
#include "gfx/all_gfx.c"
#include "gfx/all_gfx.h"


typedef struct{
        s32 x, y;
        s32 vy; // used for jumping...
} mariotype;

mariotype mario;

#define GRAVITY 48


void MoveMario(void);
```

```
// Main function
int main(void)  {
        // PAlib init
        PA_Init();
        PA_InitVBL();

        PA_InitText(1, 0);

        PA_LoadSpritePal(0, 0, (void*)sprite0_Pal);     // Palette....

        mario.x = 0<<8; mario.y = (192-32)<<8; // bottom of the screen... fixed
point
        mario.vy = 0; // not jumping
        PA_CreateSprite(0, 0,(void*)mario_Sprite, OBJ_SIZE_32X32,1, 0, mario.x>>8,
mario.y>>8); // Sprite

        while(1)
        {
                MoveMario();

                PA_SetSpriteXY(0, 0, mario.x>>8, mario.y>>8);

                PA_WaitForVBL();
        }

        return 0;
}
```

I won't detail all the beginning, as it's just all the normal includes, followed by mario's movement structure (will use fixed point), and then the macro for the gravity...

Next, we have a little `void MoveMario(void);`, which is just the declaration of the function we will use to move mario around... If the code's not there, why do we need it ?? Because the function is written AFTER the main function, so in order for *main* to know about it, we just have to copy/paste its declaration at the beginning...

Then come the normal inits, with text on the top screen...

After that, we just load the sprite (on the bottom screen) and initialize the position... Note that we are using fixed point math, that's what the «`8` is there for... And the »`8` to convert back to normal position, when placing the sprite...

```
while(1)
{
        MoveMario();

        PA_SetSpriteXY(0, 0, mario.x>>8, mario.y>>8);

        PA_WaitForVBL();
}
```

For once, the main loop doesn't have much in it... It has a function call (to move mario), then places the sprite on the screen, and waits for the screen sync... Nothing much.

Now we'll check the MoveMario function, as that's where most of the code resides :

```
void MoveMario(void){
        if(Pad.Newpress.Right) {
                PA_StartSpriteAnim(0, 0, 0, 1, 6);
                PA_SetSpriteHflip(0, 0, 0);
        }
        else if(Pad.Newpress.Left) {
                PA_StartSpriteAnim(0, 0, 0, 1, 6);
                PA_SetSpriteHflip(0, 0, 1);
        }

        if ((Pad.Newpress.A) && (mario.vy == 0)){  // If pressed A and not in the
air
                mario.vy = -1000; // Start jumping
        }

        // Moving Code
        mario.x += (Pad.Held.Right - Pad.Held.Left)<<8;  // in fixed point...

        // Add gravity
        mario.vy += GRAVITY;
        mario.y += mario.vy;
        if (mario.y >= (192-32)<<8) {
                mario.y = (192-32)<<8;
                mario.vy = 0;
        }

        if (mario.vy != 0) PA_SetSpriteAnim(0, 0, 2); // If going up or down, means
the sprite is jumping !
        else if(!((Pad.Held.Left)||(Pad.Held.Right))) PA_SetSpriteAnim(0, 0, 0);//
Image if not in the air and not walking
}
```

This should look famliliar, as it has some code copied from the Animation examples in PAlib...

```
if(Pad.Newpress.Right) {
        PA_StartSpriteAnim(0, 0, 0, 1, 6);
        PA_SetSpriteHflip(0, 0, 0);
}
```

This simply means that if you press Right for the first time, the sprite should not be flipped, and the animation should start, from frame 0 to 1 (because the walking animation only has 2 frames...), at a speed of 6 frames per second... 6 fps is really slow, but we only have 2 frames... so that's good.

If you haven't pressed right, it checks if you pressed left, in which case it's pretty much the same thing, but this time the sprite must be flipped, in order to look in the correct direction...

Then comes the first part of the jumping/gravity code, with

```
        if ((Pad.Newpress.A) && (mario.vy == 0)){
        mario.vy = -1000; // Start jumping
}
```

When you press A, the vertical speeds is set to it's maximum (well, you can choose to put more if you want), so mario will go up... Why did we have to add the `mario.vy == 0` condition ? If not, you could press A several times in a row, and just move up and up !! With this, you cannot start a jump unless you are already on the floor level...

In order to move mario left and right, there's a little `mario.x += (Pad.Held.Right - Pad.Held.Left)«8;` in there... What's the `«8` here for ? Remember we are working with fixed point numbers, so you'll want to have more than just 1/256 pixel movement per turn ! Note that with this code, you can change direction and move while in the air...

Then comes the second part of the gravity code :

```
mario.vy += GRAVITY;
mario.y += mario.vy;
if (mario.y >= (192-32)<<8) {
        mario.y = (192-32)<<8;
        mario.vy = 0;
}
```

This is just like we have already seen... Each frame, you change the speed according to the gravity, then move the sprite according to the speed... If the sprite moves down the floor level, you have to correct its position and set its speed to 0...

The last part of the code (wow, was pretty short !) is what changes the animation if you are jumping, and stops it if you aren't walking :

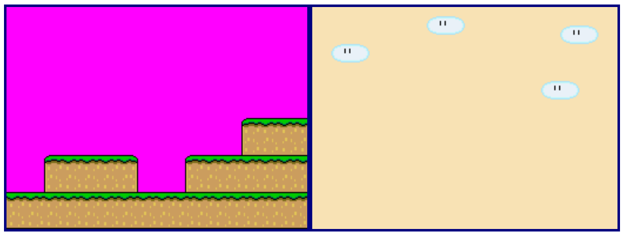
```
if (mario.vy != 0) PA_SetSpriteAnim(0, 0, 2);
else if(!((Pad.Held.Left)||(Pad.Held.Right))) PA_SetSpriteAnim(0, 0, 0);
```

First, if your speed is not null (either going up or down... so when you are in the air...), it sets the animation to the last frame (frame 2), which is the frame with mario and yoshi in the air... If you aren't jumping, and neither going left or right, then why just set the animation to 0 (the normal standing frame).

And that's already it for step 1 !! Come back soon for the next, more interesting but harder part, with simple tile collisions...

Step 2 - Basic Tile Collisions

Now that our little mario can run around, we'll add a background and make him collide with it... Here are the 2 backgrounds I added (one for collisions, and one for the back...



As you can see, they are just the size of the DS screen, because we won't have them scrolling yet...

I'll post the whole code again, but in several parts, explaining it part by part... :

```
// Includes, only one sprite
#include <PA9.h>


// PAGfxConverter Include
#include "gfx/all_gfx.c"
#include "gfx/all_gfx.h"


typedef struct{
        s32 x, y;
        s32 vy; // used for jumping...
        s32 flip;
} mariotype;

mariotype mario;

#define GRAVITY 48
#define MARIO_SPEED 512


void MoveMario(void);
void CheckCollisions(void);
u8 GetTile(s16 x, s16 y);
u8 LeftCollision(void);
u8 RightCollision(void);
u8 DownCollision(void);
u8 TouchingGround(void);
```

This is the code preceding the main function. As you can see, I added a *flip* variable in the structure, which will be used later on. There are also quite a few new functions declared, which have pretty easy names, I bet you could be able to guess what each of them do... Some of these functions aren't declared as *void*, but as *u8*, which means it'll return a value...

On we go to the main's code :

```
// Main function
int main(void)  {
        // PAlib init
        PA_Init();
        PA_InitVBL();

        PA_InitText(1, 0);

        PA_LoadSpritePal(0, 0, (void*)sprite0_Pal);     // Palette....

        PA_LoadPAGfxLargeBg(0, 1, mario_world); // platfroms...
        PA_LoadPAGfxLargeBg(0, 3, back); // back

        mario.x = 0<<8; mario.y = (128-32)<<8; // bottom of the screen... fixed
point
        mario.vy = 0; // not jumping
        mario.flip = 0;
        PA_CreateSprite(0, 0,(void*)mario_Sprite, OBJ_SIZE_32X32,1, 0, mario.x>>8,
mario.y>>8); // Sprite

        while(1)
        {
                MoveMario();

                PA_SetSpriteXY(0, 0, mario.x>>8, mario.y>>8);

                PA_OutputText(1, 2, 9, "X : %d   ", mario.x >> 8);
                PA_OutputText(1, 2, 10, "Y : %d   ", mario.y >> 8);

                PA_WaitForVBL();
        }
        return 0;
}
```

Haven't changed much in this code. It only has 2 extra background loadings :

- `PA_LoadPAGfxLargeBg(0, 1, mario_world);` loads the background we will collide with... I used LargeBg because we'll need to have more than 512 pixels large when we'll want to scroll it...
- `PA_LoadPAGfxLargeBg(0, 3, back);` is the back, with some clouds on it. It's at position 3, because nothing can be behind it.

Why haven't I used background number 0 for mario_world ? Because we can keep it to display the score, the coins, etc...

Last thing changed : I added 2 small texts displaying mario's position...

And now, the MoveMario function :

```
void MoveMario(void){
        if(Pad.Newpress.Right) {
                PA_StartSpriteAnim(0, 0, 0, 1, 6);
                PA_SetSpriteHflip(0, 0, 0);
                mario.flip = 0;
        }
        else if(Pad.Newpress.Left) {
                PA_StartSpriteAnim(0, 0, 0, 1, 6);
                PA_SetSpriteHflip(0, 0, 1);
                mario.flip = 1;
        }

        if ((Pad.Newpress.A) && (TouchingGround())){  // If pressed A and not in
the air
                mario.vy = -1200; // Start jumping
        }

        // Moving Code
        mario.x += (Pad.Held.Right - Pad.Held.Left)*MARIO_SPEED;         // in
fixed point...

        // Add gravity
        mario.vy += GRAVITY;
        mario.y += mario.vy;

        CheckCollisions();

        if (!TouchingGround()) PA_SetSpriteAnim(0, 0, 2); // Not on the ground
        else if(!((Pad.Held.Left)||(Pad.Held.Right))) PA_SetSpriteAnim(0, 0, 0);//
Image if not in the air and not walking
}
```

Nothing much has changed here, except :

- `if (Pad.Newpress.A) && (TouchingGround){` Now, mario jumps when we have checked that he is touching the ground... This uses one of the new functions we'll see later on.
- `mario.x += (Pad.Held.Right - Pad.Held.Left)*MARIO_SPEED;` I decided he didn't move fast enough, so I added a MARIO_SPEED macro, and made him go like 2 pixels per frame, which looks better.

- `CheckCollisions();` is the new collision checking function... we'll see that right after
- `if (!TouchingGround()) PA_SetSpriteAnim(0, 0, 2);` is the command to show mario as jumping... Instead of using the speed like before, it now checks if mario is touching the ground or not...

That's about it for this function, nothing much, as I had said. But now, all the remaining functions are new, I'll detail them almost 1 by 1...

```
u8 GetTile(s16 x, s16 y){
	if (x < 0 || x > 256) return 1; //Say it was a collision if the sprite 
tries
	                                //to move out of the map horizontal 
boundaries
	return mario_world_Map[((y>>3)*32) + (x>>3)];
}
```

This is a very simple function which returns the tile number when given the position in pixels... Why did I put that in a function all by itself ? Because when we'll add scrolling, we'll just have to change this function, and everything will be updated... Isn't that easier ? How does it find the correct tile ?

- First, it checks that x is a correct value (>0 or <256), and if not will return 0 (no tile).
- Then, it reads directly in the backgrounds map the tile...
  - `(x»3)` is the tile position. X is divided by 8 (equivalent to »3) because a tile is 8 pixels wide...
  - `(y»3)*32` is composed of :
  - `(y»3)` to get the tile vertical number, just like x»3, because tiles are 8 pixels high
  - `*32`, because the background is 32 tiles wide, so to go down by 1 tile, you have to add 32 tiles... We'll have to change that when we add scrolling, to make it work on bigger maps...

So this function returns the number of the tile... If there is no tile (transparent tile), the number returned will be 0... In this example, we'll do a collision for any number but 0. This means that you will collide with everything. If you need different collisions, like passing through certain walls, you have to adapt the code to your need (this is just a basic code for starters...)

```
u8 LeftCollision(void){
	return GetTile((mario.x>>8)+2, (mario.y>>8)+8+(mario.flip*13));
}

u8 RightCollision(void){
	return (GetTile((mario.x>>8)+29, (mario.y>>8)+8 + ((!mario.flip)*13)));
}
```

These 2 functions will be used to check if there is a left or right collision. First thing, it uses the GetTile function we have just seen, and gives it mario's position in normal, non fixed point, coordinates.

- For the left collision, it checks either position (on mario) (2, 8) or (2, 21), depending on if the sprite is flipped or not... Why ? Because if flipped, the collision will be done either with the nose or the back (there's a free space under the nose). So if flipped, the vertical position will change to test the collision at a different spot...
- For the right collision, it's the same, but with x = 29 instead of 2... same thing with the nose...

Why does it test x = 2 or 29, and not x = 0 and x = 31 ? Because if you look at mario's sprite, on the top of the page, you'll see that it has 2 free pixels on each side... So we want to test the collision only where the actual drawing starts...

```
u8 DownCollision(void){
        return (mario.vy >= 0 && GetTile((mario.x>>8)+10 + (mario.flip*11),
(mario.y>>8)+31));
}
```

Theis one tests if mario is touching a floor or not. If yes, we'll need to move him back up a little... Here, there's the mario.flip again !! Why this time ? Because, look at the sprite's feet, which will determine the actual collision... They aren't centered ! This means that depending on the side he's looking at, his feet will be a little more to the left or to the right. This is taken into account by changing the x position checked from 10 to 21 if flipped...

```
u8 TouchingGround(void){
        return GetTile((mario.x>>8)+10 + (mario.flip*11), (mario.y>>8)+32);
}
```

This is the last collision check. It is the exact same code as the previous one, but 1 pixel lower (+32 instead of +31). It check if there's floor underneath yoshi's feet...

Now comes the actual CheckCollisions function, which will use all the functions we have just talked about and adjust the position/speed...

```
void CheckCollisions(void){

        while(LeftCollision()){ // Collision on the left of the sprite...
                mario.x+= 256; // Move by 1 pixel...
        }
        while(RightCollision()){ // Collision on the right of the sprite...
                mario.x-= 256; // Move by 1 pixel...
        }

        while(DownCollision()){ // Collision on the bottom of the sprite...
                mario.y -= 128; // Move by 1/2 pixel...
                mario.vy = 0; // TOuched the floor...
        }
        if(TouchingGround()) mario.vy = 0;

}
```

Nothing much to say, in fact ! While the sprite touches left, move right, and while it touches right, move left... Feels logical...

For the floor collisions, I made it move by half a pixel, just for fun, lol... If you touch the floor level, then the vertical speed must be set to 0...

There are 2 different parts to the ground level check :

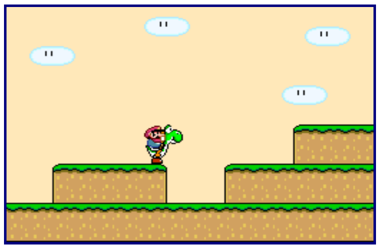
- One that actually checks if you are IN the ground, and moves mario and yoshi accordingly (while...)
- One that just looks if there's ground right underneath yoshi, and just puts the speed to 0 if that's the case...

And that's it !!

Now that this is working, we just need to add scrolling (and parallax scrolling, for a nice effect) and see the result Once again, this is a BASIC tile collision system, as it treats all the tiles the same way.

You could have a different check, and have the player react differently to each type of tile...

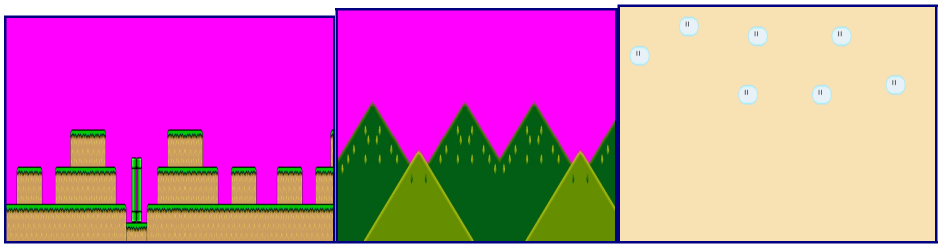
Here's what I get in dualis r12 (r11 doesn't work with LargeMaps) :



Edit

Step 3 - Background Scrolling

Now that this works, we'll add background scrolling (parallax scrolling) to give it a nice depth when scrolling. The 3 backgrounds we'll use are :



As you probably noticed, they don't all have the same length !! Why ? Because they won't scroll at the same speed. The first background will scroll at full speed, so I made it 1024 pixels wide to test it out. The second will be a bit slower, and the clouds will scroll at like 1/4th of the speed, so don't need to be very long...

I'll post the text in PAGfx.ini for once :

```
#TranspColor Magenta

#Sprites :
mario.png 256colors sprite0

#Backgrounds :
mario_world.png LargeMap
hills.png LargeMap
back.png LargeMap
```

As you can see, the transparent color is magenta... In the platform and hills backgrounds, you have tons of magenta, which will all become transparent... The last one (clouds), however, has no magenta... As it's behind, you don't want it to be transparent... All the backgrounds are declared as LargeMap, because they'll need to be more than 512 pixels wide...

For this third part, I'll only post blocks of code, as there have been very little changes...

Concerning mario's structure, I've just added a scroll parameter :

```
typedef struct{
        s32 x, y;
        s32 vy; // used for jumping...
        s32 flip;
        s32 scrollx; // Scroll value...
} mariotype;
```

Nothing much to say about it I guess. It starts at 0...

Then we need to load the background, each at a different level :

```
PA_EasyBgLoad(0, 1, mario_world); // platfroms...
PA_EasyBgLoad(0, 2, hills); // hills
PA_EasyBgLoad(0, 3, back); // back
```

Now that the backgrounds are loaded, we have to initialize the parallax scrolling... If you are unsure of what that is, go back to the background tutorial and reread the parallax scrolling part... The code is simple :

```
PA_InitParallaxX(0, 0, 256, 128, 64);
```

- This means we'll start parallax scrolling on screen 0, and not activate it (0) on the first background (which could/should be used for the score and all...).
- For background 1, it has a speed of 256, which means normal speed...
- For background 2, it has a speed of 128, so that would be only half the speed...
- For background 3, the speed is 64, so 1/4 of the speed...

Here comes the only code added, for scrolling when mario gets to the edge :

```
MoveMario();

if ((((mario.x-mario.scrollx)>>8) > 160) && ((mario.x>>8) < 1024-128)){ // Scroll 
more...
        mario.scrollx = mario.x - (160<<8);
}
else if ((((mario.x-mario.scrollx)>>8) < 64) && ((mario.x>>8) > 64)){
        mario.scrollx = mario.x - (64<<8);
}

PA_ParallaxScrollX(0, mario.scrollx>>8);

PA_SetSpriteXY(0, 0, (mario.x-mario.scrollx)>>8, mario.y>>8);
```

I added it right after MoveMario, so that it has the latest sprite position...

As you can see, when will it scroll to the right ? When mario's position on the screen is further than pixel 160, but only if it's not the end of the level (1024-128, because it stops scrolling before the end...)

And how does it get the correct scrolling value (in fixed point) ? `mario.scrollx = mario.x - (160«8);` This means that we scroll enough to position the sprite at a maximum x = 160 value... (except, again, if you're at the end of the level)

The same thing applies the other way around, scrolling only if the sprite is at more than 64 pixels from the start, and never putting the sprite closer than 64 pixels if possible...

```
PA_ParallaxScrollX(0, mario.scrollx>>8);
```

This scrolls all the background by the given scroll value, with »8 because we were using fixed point values, and the parallax scrolling takes normal values...

Another change is the sprite’s position :

```
PA_SetSpriteXY(0, 0, (mario.x-mario.scrollx)>>8, mario.y>>8);
```

Here, we remove the scroll value before positionning the sprite... This means that if the mario.x is at 512, it will be placed back on the screen according to the scroll value... isn’t that neat ?

The last change is in the GetTile function :

```
u8 GetTile(s16 x, s16 y){
        if (x < 0) return 1; // Say it was a collision...
        return mario_world_Map[((y>>3)*128) + (x>>3)];
}
```

I changed the number of tiles horizontally from 32 (256 pixels) to 128 (1024 pixels wide), to make it work again...

And that’s all !!

Note that to have more effective collisions, you’ll want to use a collision map, which is a map you do like any normal map in an editor, but you don’t actually show it on the screen...

<back to top>

<day14>

Day 14 - Carré Rouge

Carré Rouge is a javascript game which I find pretty cool :p Check it out over here :

French Version English Version

What this tutorial will do is show you how to make a similar game on Nintendo DS, from start to finish... We’ll do this in a few steps :

- Long step : Move all blocks arounds and add a collision check
- Add a Time counter and make the game restart when you lose
- Highscore save (with name, but only on gba flash carts...) and make the blocks go faster and faster...

Step 1 - Moving and Colliding

The functions we’ll use in this part are new at all... We’ll already used the MoveSprite before, as well as seen how to check collisions between 2 rectangles...

I decided to cut down this first demo into lots of functions, and so we’ll see the program one bit at a time :

```
// PAGfxConverter Include
```

```
#include "gfx/all_gfx.c"
#include "gfx/all_gfx.h"

#define MAXBLUE 4 // number of blue ones...

typedef struct{
        s16 x, y; // position in pixels
        s32 fx, fy; // Position in fixed point
        s32 vx, vy; // speed...
        s16 w, h; // HALF width and HALF height, because that's what we'll use...
} rectangle;

rectangle blue[MAXBLUE]; // MAXBLUE possible rectangles, we'll probably use less

rectangle red; // Our rectangle...
```

Nothing special here... We add the graphics (basic graphics...), and have a macro defined for the number of blue rectangles we want... Why ? Because that way we can change the number of rectangles, and everything that goes with it, by editing just 1 line, and not replacing the number all over the place...

Then comes the structure we'll use for the rectangles.

- x and y for the normal position
- fx and fy for the fixed point position
- vx and vy for the speeds (fixed point)
- w and h for width and height

Now, one might ask why we have both fixed point and normal positions... That to avoid having to do `»8` all the time, nothing more, nothing less... We'll just need to do it once per frame to update x and y, and then we'll use x and y to check if it bounces on a wall or collides...

`rectangle blue[MAXBLUE];` is the blue rectangles array... How many are there ? MAXBLUE ! If you change the 4 at the top by 5, it'll change the array's size automatically... nice !

We use that structure for our main sprite too...

On we go to the function declarations :

```
// Blue rectangle inits
void Blue16x32(u8 number);
void Blue32x16(u8 number);
void Blue32x32(u8 number);

void MoveBlue(u8 number); // Blue rect move
void CheckCollision(u8 number); // Checks if there is a collision...
```

- The first 3 declarations will be functions to initialise a blue rectangle of a given size...
- *MoveBlue* is used to move the blue rectangle number *number*. It moves it and checks for bounces off the screen limits
- *CheckCollision* checks for a collision between a given blue rectangle (*number*) and the red one... also contains the code to display "Collision" on the top screen...

The next functions to come are a little different from what we are used to :

```
extern inline void ToFixedPoint(u8 number){
        blue[number].fx = blue[number].x <<8;
        blue[number].fy = blue[number].y <<8;
}

extern inline void ToNormal(u8 number){
        blue[number].x = blue[number].fx >>8;
        blue[number].y = blue[number].fy >>8;
}
```

This, time, they are declared as *extern inline* functions, and their code is directly written... What's the main difference between a normal function and an inline one ? When the code is compiled, each time a function is declared inline, it is just copy/pasted into the code... This means that it can be faster, because you don't have the waiting time while calling a given function... However, if you put all your big functions inline, it'll copy/paste all the huge chunk of code everywhere you use it, and thus produce much bigger code. I used it here to show you how it works, and because these functions just have 2 really short lines of code...

What do these functions do ? The first one, *ToFixedPoint* converts the .x and .y normal position to the fixed point position, storing it in .fx and .fy... The second one, *ToNormal* does the opposite, copying the fixed point position into normal position in .x

Then *WHY* do we need to have a fixed point position, if we'll use the normal one anyway ? Because the default rectangle speed will be given like a trajectory, which allows more directions and will allow later on to make them accelerate very slowly...

Now, here comes the *main* function :

```
int main(void){

        PA_Init();
        PA_InitVBL();

        PA_InitText(1,0); // On the top screen

        PA_LoadSpritePal(0, 0, (void*)sprite0_Pal);

        red.x = 128; red.y = 96;
        red.w = 16;  red.h = 16; // Half width and height...
        PA_CreateSprite(0, 0,(void*)red_Sprite, OBJ_SIZE_32X32,1, 0, red.x - red.w,
red.y - red.h);

        PA_InitRand(); // We'll put some random stuff in there...

        s32 i;

        for(i = 0; i < MAXBLUE; i++){
                u8 random = PA_Rand()%3;
                if (random == 0) Blue16x32(i);
                else if (random == 1) Blue32x32(i);
                else if (random == 2) Blue32x16(i);

                // Position and random angle
                blue[i].x = (i&1)*256; blue[i].y = (i>>1)*192;
                u16 angle = PA_Rand()&511;
                blue[i].vx = PA_Cos(angle);
                blue[i].vy = -PA_Sin(angle);

                ToFixedPoint(i); // Add the fixed point values based on the
position...
        }
        while(1)
        {
                if (PA_MoveSprite(0)){
                        red.x = PA_MovedSprite.X;
                        red.y = PA_MovedSprite.Y;
                }

                for (i = 0; i < MAXBLUE; i++) {
                        MoveBlue(i); // Move all the blue rectangles...
                        CheckCollision(i); // Check collision
                }

                PA_WaitForVBL();
        }
        return 0;
}
```

As you can see, the main code is pretty short... The important features are stored in different functions we'll see...

Step 2 - Score

Coming someday to a computer near you!

<back to top>

<day15>

Basic Changes

OK, so we are going to be continuing on from where the first part of the platform tutorial, day 13, left off. Bassically I started with the finished product of day 13 and added the features that I thought were most useful. I hope you think so too. But before adding this extra functionality I changed a few things to make the code more readable. First I added #define's for the fixed point stuff...

Fixed Point Defines

```
#define norm_fix(x)             ((x)<<8)
#define fix_norm(x)             ((x)>>8) //xxx truncates instead of rounds
#define fix_mult(x,y)           (((x)*(y))>>8)
#define ratio_fix(x,y)  ((256*(x))/(y))
#define fixed_fraction(x)       ((x)&&0xff)
```

norm_fix takes a normal number and changes it to fixed point. fix_norm takes a fixed number and makes it normal, fix_mult takes two fixed points and multiplies them, ratio_fix takes a ratio such as 1,3 (1/3) and turns it into fixed point. Finnaly fixed_fraction gets the fraction portion of a fixed point number. I use these instead of bitshifting in the code as I think it’s easier to read. —

OK so that is the most basic change we have, but now we get into the first major change and that’s to make a collision map.

Code for Collision map

The second thing that I decided to change was the code for collision detection. As it is now the collision detection code contains lots of “magic numbers” and will only work with the players sprite. I decided to change this as we are going to need the enemies to run into stuff as well as the player. In order to do this I store the players four collision points in a struct called hitboxinfo which contains four pointinfo structs.

```
typedef struct{
   s8 x, y, flipx;
}pointinfo;

typedef struct{
   pointinfo left, right, up, down;
   u8 flipped;
}hitboxinfo;
typedef struct{

        s32 x, y;
        s32 vy; // used for jumping...
        s32 scrollx;
        u8 sprite; //the sprite number
        hitboxinfo hitbox;
        u32 speed;
}mariotype;
```

Pointinfo has all the information for one point x is the x compared to 0,0 on a sprite, y is the y compared to 0,0 on a sprite, and flipx is how much to add or subtract to the x value if the sprite is flipped. hitboxinfo contains four of these pointinfo’s one for left, right, up, and down. left is the point used for a collision on the left side, right for right side, up for the top, and down for the bottom. I also added hitbox to the mariotype struct for the mario hitbox info. I set mario’s hitbox info here...

```
mario.hitbox.left.x=2;
mario.hitbox.left.y=8;
mario.hitbox.right.x=29;
mario.hitbox.right.y=8;
mario.hitbox.down.x=10;
mario.hitbox.down.y=31;
mario.hitbox.up.x=5;
mario.hitbox.up.y=0;
mario.hitbox.left.flipx=0;
mario.hitbox.right.flipx=0;
mario.hitbox.up.flipx=10;
mario.hitbox.down.flipx=11;
```

As you can see I get the postions for x and y from the old methods in platform demo 3...

```
must add in
```

Now we need the methods used for handling this new type of data GetTile is still the same but instead of having four different collision methods we now only have one...

```
//check for a collision with a tile at pointx, pointy, and adding in flipx if the
sprite is flipped
u8 TileCollision(pointinfo point, u8 flipped){
   return (GetTile(fix_norm(mario.x)+point.x + (flipped*point.flipx),
fix_norm(mario.y)+point.y));
}
```

As you can see this method takes a pointinfo. The pointinfo tells what point to check on mario. To suplement this method you can also check all sides at once using our hitboxinfo type object.

```
u8 anycollision(u8 tile, hitboxinfo hitbox){
   if(TileCollision(hitbox.left, hitbox.flipped)==tile)return 1;
   if(TileCollision(hitbox.right, hitbox.flipped)==tile)return 2;
   if(TileCollision(hitbox.up, hitbox.flipped)==tile)return 3;
   if(TileCollision(hitbox.down, hitbox.flipped)==tile)return 4;
   else return 0;
}
```

The tile you pass it is the tile your checking for a hit, and hitbox is the hitboxinfo variable that you are passing it. The method returns 1 for left, 2 for right, 3 for up, and 4 for down. Using this code we can now change the collision method.

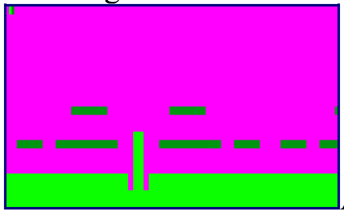
```
void CheckCollisions(void){

        while(anycollision(1, mario.info.hitbox)==1){ // Collision on the left of
the sprite...
                mario.x+= norm_fix(1); // Move by 1 pixel...
        }
        while(anycollision(1, mario.info.hitbox)==2){ // Collision on the right of
the sprite...
                mario.x-= norm_fix(1); // Move by 1 pixel...
        }
        while(anycollision(1,mario.info.hitbox)==3){ // Collision with the
something on top
           mario.y+=norm_fix(1);
           mario.vy=0;
        }
        while(anycollision(1,mario.info.hitbox)==4){ // Collision on the bottom of
the sprite...
                mario.y -= norm_fix(1); // Move by 1/2 pixel...
        }


        if( (TouchingGround()==1 || TouchingGround()==2) && mario.vy>0){ //if he's
touching the ground and moving down he can't hit ground while moving up
                mario.vy = 0;
                mario.jumping=0;
        }
```

1 is the number of the tile that we want mario to be able to stand on (remember how we put that green box in the upper left corner this is why, to make sure that green is tile number 1), and mario.info.hitbox is mario’s hitbox that we want to check for collisions with. What happens when there is a collision is exactly the same. — *dustin rhodes 28/06/2006 21:37*

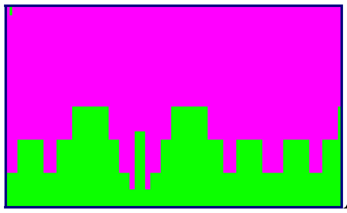
Adding Collision Map Functionality

Changing The Map

So by now we've managed to add a collision map but we haven't actually added anything to the game. However because we switched to using a collision map it is very easy to now add other types of tiles. We'll start easy and add that type of block that you can jump up through and then land on top of. To do this we go back to our collision map and add another type of tile. Here is the new Collision map...



And this is the old collision map



As you can see there are a couple key differences. For one thing we have added in a new type of tile, tile number 2 the dark green. We know it's tile number two because we placed it right after the light green tile, tile 1, in theupper left corner. We placed this tile on areas we want the player to be able to jump up through but then land on top of. The second thing you will notice is that we took out areas of light green. This means that mario will not collide with those areas, if you play the demo at this point you will see that it plays like a mario game should.

Changing The Code

So now we have the map changed but our code doesn't reflect these changes yet. Luckily it's incredibly easy to add this new funtionality to our code. All we have to do is change our collision method just a bit. This is the bit we need to change.

```
while(anycollision(1,mario.info.hitbox)==4 || anycollision(2,mario.hitbox)==4){ //
Collision on the bottom of the sprite...
                mario.y -= norm_fix(1); // Move by 1/2 pixel...
        }


        if( (TouchingGround()==1 || TouchingGround()==2) && mario.vy>0){ //if he's
touching the ground and moving down he can't hit ground while moving up
                mario.vy = 0;
                mario.jumping=0;
        }
```

As you can see all we have done is add || anycollision(2,mario.hitbox)==4 and || TouchingGround()==2. What this does is says that the player is colliding with something underneath him if the tile returned is number 2, our dark green tile, he is also touching the ground if he is standing on tile number 2. We do not add || anycollision(2,mario.hitbox) to any of the other sides because the only way mario can collide with these tiles is on the bottom of his sprite. This allows him to jump up through the tiles and then come down on top. At we have completed Platform Demo part 4. — *dustin rhodes 28/06/2006 23:33*

Download

Here is platform game demo 4

(MAY NOT WORK)

Sprite Allocation

Looking at the code from platform game demo 4 you may notice two methods I have not yet talked about...

```
u8 getsprite(){
    int i;
    for(i=0;i<128;i++){
        if(sprite[i]==0){
            sprite[i]=1;
            return i;
        }
    }
    return -1;
}

void deletesprite(u8 spritenumber){
    sprite[spritenumber]=0;
    PA_DeleteSprite(0,spritenumber);
}
```

As I'm sure many of you have realised coming up with and remembering sprite numbers for every sprite soon becomes impossibly difficult. These two methods along with an array of u8's called sprite solve this problem. They are incredibly simple to use but have saved me loads of hassle. when creating an object that has a sprite set it's sprite variable using getsprite(); like this...

```
mario.sprite=getsprite();
```

Then create the sprite using the mario.sprite variable like this

```
PA_CreateSprite(0, mario.sprite,(void*)mario_Sprite, OBJ_SIZE_32X32,1, 0,
fix_norm(mario.x), fix_norm(mario.y));
```

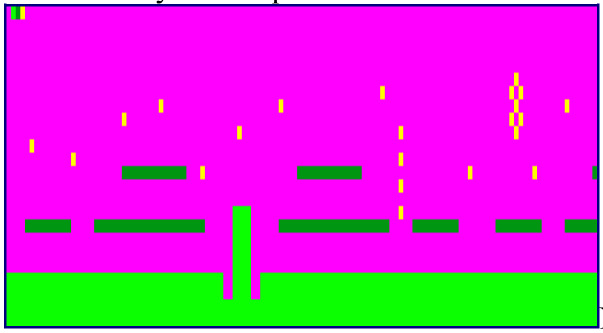
This way you never have to remember sprite numbers and you are guaranteed to not try to use the same sprite number twice. Get sprite works by going through the sprite array until it comes to an empty sprite, a 0. After finding one it changes it to a 1 showing that it has been filled and returns the number. deletesprite takes the sprite you give it and sets it back to empty, a 0. It then deletes the sprite from the screen. These two functions come in very useful later on.

Coins

Our next task is to make coins that act in the same way as coins do in mario games. We will also handle this using our collision map. For every spot you wish to place a coin all that you will have to do is place a 8×8 (1 tile) square of yellow. We will then make methods that scan through the level and get the cordinates for all the coins, a method that scrolls coins as the background scrolls, a method for collision with coins, and finaly a method that deletes the sprites for coins when they move off the screen and remakes the sprites when they move on the screen.

Changing the Map

We add the yellow squares where we want coins to be like this...



Now in game where ever we placed one of those yellow squares our program will know to place a coin. Again we put the one in the upper left corner so that we know this is tile 3.

Changing the Code

Unfortunatly there is quite a bit more work to do with the code then with the map but that's ok it's not to bad. First we create a struct for our coins...

```
typedef struct {
   s32 x,y;
   u8 alive;
   u8 sprite;
} coininfo;

coininfo coin[maxcoins];
```

x and y are the x and y cordinates of the coin, alive is whether or not the player has gotten it or not. Sprite is the sprite number, a sprite of 0 means it's not currently on the screen because the player always has sprite 0. coin[maxcoins] is an array that will hold all of our coins in it. maxcoins is a #define that you set to the maximum number of coins any one level will have. Next we need our code for placing the coins...

```
//places coins whenever it finds yellow
void placecoins(){
   int i;
   int j;
   int coinnumber=0;

   for(i=0;i<levellength;i++){
      for(j=0;j<levelheight;j++){
         if(GetTile(i*8,j*8)==3){
            coin[coinnumber].x=i*8;
            coin[coinnumber].y=j*8;
            coin[coinnumber].alive=1;
            PA_OutputText(1,1,coinnumber,"x: %d, y: %d",coin[coinnumber].x,
coin[coinnumber].y);
```

```
            if(coin[coinnumber].x<=fix_norm(mario.scrollx)+256 &&
coin[coinnumber].x>=fix_norm(mario.scrollx)){
               coin[coinnumber].sprite=getsprite();
                PA_CreateSprite(0, coin[coinnumber].sprite,(void*)coin_Sprite,
OBJ_SIZE_8X8,1, 0, coin[coinnumber].x, coin[coinnumber].y);
                }
            coinnumber++;
         }
                }
        }
}
```

What this code does is goes through every tile in our level and checks if it's tile number three. If it is then it places a coin at that x and y. The variable coinnumber is just the current coin we are placing. Because there can not be two coins on one tile we only check each tile once. Since a tile is 8×8 we multiply both our x and y (i and j) by 8. When we find a tile that's equal to three (GetTile(i*8,j*8)==3) We place set coin[coinnumber].x equal to that position coin[coinnumber].y equal to that position and set the coin to be alive. I then print out the coins location just to make sure our code is working. The next line checks if the coin is currently on screen. If it is then we get it a sprite using getsprite() and create it's sprite using our sprite,x, and y variables. If it's not on the screen we don't create a sprite for it or give it a sprite number, this way we don't have to use our limited amount of sprites for coins not on screen. At the end of the loop we add one to coinnumber so that next time the next coin in the coin[] array will be used. This method should only be called once right before your main loop starts. And now we find how to scroll coins properly...

```
void scroll(){


        if (((fix_norm(mario.x-mario.scrollx)) > 160) && (fix_norm(mario.x) < 1024-
128)){ // Scroll more...
                mario.scrollx = mario.x - norm_fix(160);
        }
        else if ((((mario.x-mario.scrollx)>>8) < 64) && ((mario.x>>8) > 64)){
                mario.scrollx = mario.x - norm_fix(64);
        }


        //automatic scrolling
//      mario.scrollx+=ratio_fix(1,2);
//      mario.scrolly+=ratio_fix(1,2);

        PA_ParallaxScrollXY(0, fix_norm(mario.scrollx),fix_norm(mario.scrolly));

        //move player
        PA_SetSpriteXY(0, mario.sprite, fix_norm(mario.x-mario.scrollx),
fix_norm(mario.y-mario.scrolly));

        //scroll the coins
        int i;
        for(i=0;i<maxcoins;i++){
           if(coin[i].alive){
                        if(coin[i].x<=fix_norm(mario.scrollx)+256 &&
coin[i].x>=fix_norm(mario.scrollx)-8){
                              //if it is then move it to the correct position
                              if(coin[i].sprite!=0){
                                PA_SetSpriteXY(0,coin[i].sprite,coin[i].x-
fix_norm(mario.scrollx),coin[i].y-fix_norm(mario.scrolly));
```

```
                                }
                                //if it's not then create a sprite for it
                                        else{
                                           coin[i].sprite=getsprite();
                                                PA_CreateSprite(0, coin[i].sprite,
(void*)coin_Sprite, OBJ_SIZE_8X8,1, 0, coin[i].x-fix_norm(mario.scrollx),
coin[i].y-fix_norm(mario.scrolly));
                                        }
                                }
                           //if the coin is offscren delete it
                           else{
                              //don't delete already gone stuff
                              if(coin[i].sprite!=0){
                                      deletesprite(coin[i].sprite);
                                      coin[i].sprite=0;
                                   }
                           }
                        }
        }

        //PA_OutputText(1, 2, 11, "X : %d   ", fix_norm(mario.x));
        //PA_OutputText(1, 2, 10, "Y : %d   ", mario.y);
        //PA_OutputText(1, 2, 12, "Scroll : %d   ", fix_norm(mario.scrollx));
}
```

A lot of this method should be familiar to you, but we added the part that scrolls coins. First we make a for loop that will loop through our entire array of coins. This could probably be made to be more effecient but I'm trying to keep it simple. What we do first is check if the coin is alive, if it is then we scroll if it is not then we don't do anything with it. Next we check if it is on the screen (if(coin[i].x<=fix_norm(mario.scrollx)+256 && coin[i].x>=fix_norm(mario.scrollx)-8)) If it is on the screen then we check if it has a sprite or not, if it does not have a sprite then it's sprite variable will be set to 0. If it has a sprite all we need to do is move it to it's correct position using PA_SetSpriteXY(0,coin[i].sprite,coin[i].x-fix_norm(mario.scrollx),coin[i].y-fix_norm(mario.scrolly)); However, if it is not onscreen already then we need to create a sprite for it. We set the coins sprite variable using getsprite and create the sprite using the coins x,y, and sprite variables. Now we have to deal with what happens if a coin is not onscreen. If the coin is offscreen and it's sprite is 0 we don't do anything, we don't want offscreen coins to have a sprite. If it does have a sprite though we need to get rid of it's sprite. We do this by using our deletesprite() method and passing in the coins sprite number. We then set the coins sprite number equal to 0 so that we know that it currently doesn't have a sprite. And that's it, now we should have coins that scroll, so our next and final task is to get the collision to work.

```
u8 collision(s32 x2, s32 y2, u8 h2, u8 w2){
   if ( (fix_norm(mario.x)>x2+w2) || (fix_norm(mario.x)+30<x2) ||
                     (fix_norm(mario.y)>y2+h2) || (fix_norm(mario.y)+30<y2)){
      return 0;
   }
        else{
           return 1;
        }
}
```

To begin collision detection I decided to write a generic method to decide if something had collided with mario. You pass in an x, y, height, and width. With this information we can return 1 if there is a collision with mario and 0 if there isn't. Now lets see how we use this to check for coin collisions...

```
void CheckCollisions(void){

        while(anycollision(1, mario.hitbox)==1){ // Collision on the left of the
sprite...
                mario.x+= norm_fix(1); // Move by 1 pixel...
        }
        while(anycollision(1, mario.hitbox)==2){ // Collision on the right of the
sprite...
                mario.x-= norm_fix(1); // Move by 1 pixel...
        }
        while(anycollision(1,mario.hitbox)==3){ // Collision with the something on
top
           mario.y+=norm_fix(1);
           mario.vy=0;
        }
        while(anycollision(1,mario.hitbox)==4 || anycollision(2,mario.hitbox)==4)
{ // Collision on the bottom of the sprite...
                mario.y -= norm_fix(1); // Move by 1/2 pixel...
        }


        if( (TouchingGround()==1 || TouchingGround()==2) && mario.vy>0){ //if he's
touching the ground and moving down he can't hit ground while moving up
                mario.vy = 0;
                mario.jumping=0;
        }

        //coin collisions
        int i;
        for(i=0;i<maxcoins;i++){
           if(coin[i].sprite!=0){
                   if(collision(coin[i].x,coin[i].y,8,8)){
                      coin[i].alive=false;
                      deletesprite(coin[i].sprite);
                      coin[i].sprite=0;
                        }
                }
        }

}
```

Most of this code is the same, only the small part under the coin collision comment is new. In this part we again loop through all the coins. We first check that they have a sprite, because if they don't then why check for collisions against them? If they have a sprite we pass your collision method the coins x and y and 8 for the coins height and width. If this returns true we set the coin's alive variable to false delete the sprite for the coin and set the coins sprite equal to 0. If the collision method returns false then we don't have to do anything. And that's it, you've now added coins to our program. — *dustin rhodes 29/06/2006 18:24*

Download

You can get the rom/source code with coins emplemented here... Here is platform game demo 5

Things get Tricky

We can’t go much further without our code getting incredibly messy and nasty or using some more advanced techniques. Bassicaly we are going to have to use pointers which I’ll try to explain as best I can. new code.

Pointers

A pointer is exactly what it says it is. Everything in your program is stored some place in memory correct? Well a pointer is a pointer to that place in memory. These pointers can point to mostly anything such as structs, arrays, or routines. What we are going to use them for is to let each object know what it’s sprite is (the sprite is really just an array) and also let each object have it’s own scroll method, and ai method. This way our code will be much nicer and easier to read. So enough about why we are going to use pointers, I’ll explain the syntax of pointers.

To declare a pointer to a certain type of data you say the data type you wish to point to then a * then a space and the name of your pointer. For instance if you wanted a pointer called pointer1 that pointed to a u8 you would say...

```
u8* pointer1;
```

So now that you have it declared you have to tell it what to point to. To do this we must use the & symbol. The & symbol means, give me the address of this particular variable so you could say something like this...

```
pointer1 = &mario.x;
```

This would set pointer1 to the address of mario.x, in other words pointer1 now points to mario.x. While this is interesting it is not particularly useful, we are going to use pointers that point to arrays, methods, and structs.

Pointer to Struct

The main use of a pointer to a struct is for use in methods. Usually when a struct is passed to a method, the entire struct is coppied in memory. This means that excess memory is used and also if you modify the struct in the method the changes will not be saved. In order to modify a struct in a method and have the changes stick you must pass in a pointer to that struct instead like this...

```
void coinscroll(objectinfo* mover)
```

As you can see again the * follows the variable type. The only difference you need to know is that instead of . you use → so it might look something like this... mover→x instead of mover.x However if you have more then one layer then you say this mover→info.x not mover→info→x or mover.info.x Hope that’s clear now onto our final use, pointers to methods.

Pointer to Array

To make a pointer point to an array is simple all you do is say...

We make the pointer and array like normal but then we set the pointer equal to the array. We don’t have to use the & symbol because the name of an array is already a pointer that points to the first variable of the array. If this confuses you, you can also set arraypointer equal to &sprite[0].

Pointer to Methods

Pointers to routines/methods are very usefull as they allow methods/routines to be stored as variables. To initialize a pointer to a method you simply say this...

```
void (*ai)(u8);
```

void is the return type * represents that it is a pointer and ai is the name of the pointer. The (u8) is what type of variables you must pass the method. To set this variable equal to something you say...

```
ai=&enemyai;
```

We use & to get the address and enemyai is the name of the method we want to point to. Finnaly to call the method using the pointer we say this..

```
(ai)(3);
```

(ai) is the pointer of the method we are calling and 3 is the u8 we are passing in. That’s it on pointers so now we’ll get into the structure of our code. — *dustin rhodes 01/07/2006 01:17*

Basic Structure of New Code

Before we start coding using our pointers I’m going to explain how our new code will work and hopefully why it should be easier. Every object (the player, enemies, and items) will be a struct which contains all the information necesary to draw it (x,y,pallet number, size, a pointer to the sprite) and also an ai method which will tell the object what to do every turn, a scroll method for how to scroll the object, and a collision method which will detect for collision with the player and react accordingly. We will have a large array with all these objects in it and every turn the ai, scroll, and collision will be called. This will make it very easy to make a new type of object. For instance say we wanted to introduce an enemy that moved up and down into our game. All we would do is set the enemies x and y using something similiar to how we set the coins, then set it’s sprite equal to the enemies image, it’s ai equal to y++ or something and it’s collision to kill the player if it touches him. Very easy. — *dustin rhodes 01/07/2006 01:23*

Starting code for our more complicated engine

Alright if it seems like you know what we will be attempting lets get to it and start looking at the code.

Object Structs

OK the first thing we have to worry about is the structs for our objects.

```
struct objectinfo{
   void (*ai)(objectinfo*);
   void (*scroll)(objectinfo*);
   void (*collision)(objectinfo*);
   hitboxinfo hitbox;
   s32 x,y;
   s8 sprite;
   u8 lastframe;
   u8 alive;
   const u8* spriteimage;
   u8 pallete;
   u8 size;
   s8 variables;
};
```

Now some of this stuff is familiar it is what used to be our coin struct but now it's much more generic so it can hold any type of object or enemy. We'll get the most complicated part over with first. Those first three lines are probably pretty confusing but they are easy enough after they are explained. The little * should let you know that it has something to do with pointers. All three of those are pointers to methods. void is the return type of the method scroll is the name of our pointer and the * says that it is a pointer. The objectinfo* is the parameter of the method. Because we use objectinfo in our object info struct we have to pre declare it by putting

```
typedef struct objectinfo objectinfo;
```

Even before our method declerations. This means that when we create our struct we don't say typedef struct anymore we simply use struct and then objectinfo. This also means we don't have to put the name at the end of the struct. OK so we now know what the first three lines mean hopefully. The hitboxinfo will store the collision points of our object. X and y should be pretty self explanatory they are the location of the object. Sprite is where we will store the objects sprite number. Last frame is the last frame to be used in the normal animation of the object. Alive tells us whether the object is alive or not 0 no 1 yes. Now we get to one more complicated bit..

```
const u8* spriteimage;
```

this is simply a pointer to an array of const u8's the type that is used to store sprite images, we'll use this whenever we want to draw our object. Pallete is used to store which pallete this object uses and size is the size of the object. 0 is 8×8 1 is 16×16 and so on. Finnaly variables is a generic variable to be used in the objects ai.

Some new methods

Now I'm going to walk you through all of the new/changed methods we have to make our new system easy to use. You don't technically need to know how they work it would be possible just for you to use them but I think it's generally a better idea for you to understand.

- newobject
- deletesprite

- createsprite
- anycollision

newobject

This method is used whenever we want to place a new object somewhere on the map...

```
void newobject(s32 x, s32 y, objectinfo* object, objectdata* data){
   object->spriteimage=data->spriteimage;
   object->lastframe=data->lastframe;
        object->pallete=data->pallete;
        object->size=data->size;
        object->ai=data->ai;
   object->x=norm_fix(x);
   object->y=norm_fix(y);
   object->alive=1;
   object->sprite=-1;
   object->scroll=data->scroll;  //set the scroll method of the coin
   object->hitbox=data->hitbox;
   object->collision=data->collision;
   object->variables=data->variables;
}
```

As you can see it takes an x and y so it knows where to put the object, and a pointer to an objectinfo and an objectdata. We use pointers for two reasons... 1. Passing pointers is much faster then passing the entire struct. 2. If we pass the object c makes a copy of the object to be used in that method and so none of the changes we make inside that method will have any effect outside of that method. Our method uses an objectdata struct to set all the variables in and objectinfo struct. This is much nicer and neater then having to set all this data manually everytime we want to create an object. After going through the objectinfo struct objectdata will be very familiar. The object data struct looks like this...

```
struct objectdata{
   void (*ai)(objectinfo*);
        void (*scroll)(objectinfo*);
        void (*collision)(objectinfo*);
        const u8* spriteimage;
        u8 h,w;
        u8 lastframe;
   u8 pallete;
   u8 size;
   hitboxinfo hitbox;
   u8 variables;
   s16 tile;
};
```

The one odd thing you will notice about this is the variable tile. We use this to know when to place the object. For instence if we wanted to place the object on all the number 4 tiles we would set tile equal to 4. This is put into effect in the placeobjects method.

```
if(GetTile(i*8,j*8)==data[k].tile){
               newobject(i*8,j*8, &coin[getobject()], &data[k]);
            }
```

as you can see now when we add a new object we don’t even have to adjust this method.

deletesprite

Our deletesprite will be used for deleting sprites and making sure all our variables get set correctly when we do so...

```
void deletesprite(objectinfo* todelete){
        sprite[todelete->sprite]=0;
        PA_DeleteSprite(0,todelete->sprite);
        PA_StopSpriteAnim(0,todelete->sprite);
        todelete->sprite=-1;
}
```

this method also takes a pointer to objectinfo. First it sets our sprite array to 0 at this object's position so our program knows we can reuse this sprite. We then physically delete the sprite using PA_DeleteSprite. we then stop the sprites animation just to be safe. Finnaly we change the deleted objects sprite to -1 so we know that it's not displayed. Perhaps you have noticed that when you have a pointer to an array instead of an array itself you use → instead of.

createsprite

This will be used whenever we want to create a sprite. It is especially neat because you simply hand it an objectinfo struct and the sprite will be created. This method is why we put all that information in our struct.

```
void createsprite(objectinfo* todraw){
   todraw->sprite=getsprite();
        PA_CreateSprite(0, todraw->sprite,todraw->spriteimage, 0,todraw->size,1,
todraw->pallete, fix_norm(todraw->x), fix_norm(todraw->y));
        if(todraw->lastframe)PA_StartSpriteAnim(0,todraw->sprite,0,todraw-
>lastframe,6);
}
```

Just like deletesprite this method takes an objectinfo pointer. First we get a sprite for it and set it's sprite variable to be that. Then we actually draw the sprite which is more complicated then usual. We want screen 0 as always. Then we give it our objects sprite number with todraw→sprite. todraw→spriteimage is the pointer to the image for this particular object and we pass that next. 0 shows that it is type 256 color, todraw→size lets CreateSprite know what size the sprite should be and 1 says that the sprite is square. Finnaly todraw→pallete gives the correct pallete and x and y give cordinates. After we have the sprite we check if this object has a last frame (meaning it has a animation) and if it does we start this animation.

any collision

This method won't be changing to much but now we need it to work for things besides mario. To do this we need to pass tile collision both the x and y values of the point we need to be checking. Like this...

```
u8 TileCollision(pointinfo* point, u8 flipped, s32 x, s32 y){
   return (GetTile(fix_norm(x)+point->x + (flipped*point->flipx), fix_norm(y)
+point->y));
}
```

Now we just change we just need to change our any collision method to work with this. All you will have to do is pass anycollision a tile to colide with and an object and it will see if the object collides with the tile.

```
//checks if any of the collisions are true for that tile and returns what side
u8 anycollision(u8 tile, objectinfo* object){
   if(TileCollision(&object->hitbox.left, object->hitbox.flipped, object->x,
object->y)==tile)return 1;
   if(TileCollision(&object->hitbox.right, object->hitbox.flipped, object->x,
object->y)==tile)return 2;
   if(TileCollision(&object->hitbox.up, object->hitbox.flipped, object->x, object-
>y)==tile)return 3;
   if(TileCollision(&object->hitbox.down, object->hitbox.flipped, object->x,
object->y)==tile)return 4;
   else return 0;
}
```

Using these new methods

Delete sprite was already in use and we just added to it's functionality so we don't use that in any new places. createsprite is bassicly just a replacement for PA_CreateSprite so I searched through the program and replaced all the PA_CreateSprites with createsprite. createobject is used where we created coins before. So now instead of hard writing all that info in whenever we find a yellow block we simply use createobject to do it for us. It can do it for us because we have put in all the coins data into and objectdata struct.

```
//set up the coin data
    data[0].spriteimage=coin_Sprite;
    data[0].lastframe=0;
    data[0].pallete=0;
    data[0].size=0;
    data[0].ai=&noai;
    data[0].scroll=&objectscroll;  //set the scroll method of the coin
    data[0].collision=&coincollision;
    data[0].hitbox.right.x=8;   data[0].hitbox.right.y=4;
    data[0].hitbox.left.x=0;    data[0].hitbox.left.y=4;
    data[0].hitbox.up.x=4;      data[0].hitbox.up.y=0;
    data[0].hitbox.down.x=4;    data[0].hitbox.down.y=8;
    data[0].tile=4;
```

As you can see the sprite image points to the coin_Sprite, the last frame is 0 because coins have no animation, the pallete is 0, the size is 0 which means 8×8. The coin has no ai so we give it the address of our noai method, a coin scrolls like normal so we give it the object scroll method, finnaly for collision the coin has the coincollision method which says to remove the coin if mario is touching it. The next four lines are simply the hitbox information of the coin.

Adding a bad guy and explaining ai,scroll, and collision

By now your probably wondering what the advantage of this entire new system is. We went through all that trouble implementing it but now we are still were we started with only 1 type of object. Now I'll go through the process of making a bad guy and while doing that explain just what ai, collision, and scroll do...

When ai, collision, and scroll are called

Before we learn how to write these three types of methods I'll show you where they are called.

```
u16 i;
        for(i=0;i<maxcoins;i++){
           //scroll all the coins
                if(coin[i].alive){
                        (coin[i].scroll)(&coin[i]);
                        (coin[i].ai)(&coin[i]);
                        PA_OutputText(1,1,i,"x:%d, y:%d",fix_norm(coin[i].x),
fix_norm(coin[i].y));
                }
        }
```

As you can see our objects are still in an array called coins but don't let this confuse you. This chunk of code is in our scroll method so after mario and background scrolling has been done this chunk will be reached. what it does is loop through all the objects in the coin array and for every one of them calls whatever method is pointed to by the objects scroll pointer and passes that method the address of the current object...

```
(coin[i].scroll)(&coin[i]);
```

After it does that it does the same with ai...

```
(coin[i].ai)(&coin[i]);
```

The collision methods are called in the check collisions method. Again after mario's collision is checked we check all the objects in the coin array...

```
int i;
        for(i=0;i<maxcoins;i++){
           if(coin[i].sprite!=-1 && coin[i].alive){
              //coincollision(&coin[i]);
                        (coin[i].collision)(&coin[i]);
                }
        }
```

To save time we only check collisions if the object is onscreen (sprite not equal to -1) and the object is alive. Now that we know where these three methods are called lets take a look at writing them...

How to write ai, collision, and scrolling code

A coins ai, collision detection, and scrolling are all very easy so I'll show you those first.

coin ai

luckily for you a coin has no ai it does not move on it's own at all so it's ai method is this...

```
void noai(objectinfo* bady){
}
```

note that even though the method is empty it still has to exist. It must exist because we are going to go through all the objects and call there particular ai method. If an object doesn't have an ai method then our program will freeze. Also the methods for ai, collision, and scroll must all accept an objectinfo pointer type or again our program will freeze.

coin collision

The collision detection for our coins is a little trickier but not much...

```
void coincollision(objectinfo* object){
   if(boxcollision(object->x,object->y,8+object->size*8,8+object->size*8)){
                object->alive=false;
                deletesprite(object);
        }
}
```

As you can see this method has the same header as our ai method it's return type is void and it takes a pointer to an objectinfo. All this method does is checks if there is a collision at our coins x and y with an object of size*8+8 width and size*8+8 height. This means that if size is 0 it will check an 8×8 box, if size is 1 it will check a 16×16 box and so on. If there is a collision it sets the objects alive variable to false and deletes the sprite using deletesprite.

coin scroll

The scroll method for most things will be the same. One example of something that would have a different scroll method then normal would be the lakutas that ride in clouds they do not scroll with the background but merely bounce back and fourth on top of the screen. However if in doubt just point the scroll pointer to this method...

```
void objectscroll(objectinfo* mover){
                   //check if coin is onscreen
                   if(fix_norm(mover->x)<=fix_norm(mario.scrollx)+256 &&
fix_norm(mover->x)>=fix_norm(mario.scrollx)-8){
                      //if it is then move it to the correct position
                      if(mover->sprite!=-1){
                        PA_SetSpriteXY(0,mover->sprite,fix_norm(mover->x)-
fix_norm(mario.scrollx),fix_norm(mover->y)-fix_norm(mario.scrolly));
                        }
                        //if it's not then create a sprite for it
                                else{
                                        createsprite(mover);
                                }
                   }
                   //if the coin is offscren delete it
                   else{
                      //don't delete already gone stuff
                      if(mover->sprite!=-1){
                              deletesprite(mover);
                           }
                   }
}
```

This code should look familiar as it has been used previously. All that has been done is copy and pasting this chunk from the playerscroll method into a method of it’s own. — dustin rhodes *24/07/2006 19:32*

Writing a new object type

OK, now we are going to write in a new type of object. This time we will make a goomba so you can see just how easy it is to add new types of objects. The first thing we are going to do is change our objectinfo array (the one that is currently called coins) to a more appropriate name. From now on what has been the coin[] array wiill be the object[] array. This is simply to avoid confusion now that it will hold both coins and our new type of object.

Putting it in the collision map

The first task as always is to put a new color in the collision map for our new object. We will do this exactly as we always have...

The data structure for our new object

As you can see our placeobject method gets its information about our new type of object from data[1] so we must set that to be the appropriate variables...

```
//set up the goomba data
        data[1].spriteimage=goomba_Sprite;
        data[1].lastframe=1;
        data[1].pallete=0;
        data[1].size=1;
        data[1].ai=&simpleai;
        data[1].scroll=&objectscroll;
        data[1].collision=&badycollision;
        data[1].variables=1;
        data[1].hitbox.right.x=16; data[1].hitbox.right.y=8;
        data[1].hitbox.left.x=0; data[1].hitbox.left.y=8;
        data[1].hitbox.up.x=8; data[1].hitbox.up.y=0;
        data[1].hitbox.down.x=8; data[1].hitbox.down.y=16;
```

This piece of code goes at the begining of our main method right after we set our data[0] variables. You should remember what all these variables mean from our coin object so i’ll just go over the new ones. Last frame is set to 1 instead of 0 because our goomba has 2 frames of animation 0 and 1 that it will loop back and forth through. Size is 1 instead of 0 because the goomba is 16×16 instead of 8×8. The ai method and collision method are different also but the scroll method is exactly the same as it was for the coin. This is another advantage of this system we can easily reuse any of our ai, scroll, or collision methods.

ai and collision

Since the scroll method is exactly the same I won't be going over that. However, the ai and collision methods are different so lets take a look at those...

ai

OK here is our goomba's ai. All it does is fall to the ground and walk in one direction until it hits something then turns around and walks the other direction.

```
void simpleai(objectinfo* bady){
   bady->x+=norm_fix(1)*bady->variables;
   bady->vy += GRAVITY;
        bady->y += bady->vy;

   while(anycollision(1, bady)==1){ // Collision on the left of the sprite...
                bady->x+= norm_fix(1); // Move by 1 pixel...
                bady->variables=bady->variables*-1;
        }
        while(anycollision(1, bady)==2){ // Collision on the right of the sprite...
                bady->x-= norm_fix(1); // Move by 1 pixel...
                bady->variables=bady->variables*-1;
        }
        while(anycollision(1,bady)==3){ // Collision with the something on top
           bady->y+= norm_fix(1);
        }
        while(anycollision(1,bady)==4 || anycollision(2,bady)==4){ // Collision on
the bottom of the sprite...
                bady->y -= norm_fix(1); // Move by 1/2 pixel...
                bady->vy=0;
        }


}
```

The first line of code just moves the goomba by 1 pixel either left or right. if variable is equal to 1 then it is to the right if it is -1 then it is to the left. Next we add the bady's y velocity to his y position and add gravity to his y velocity exactly like we do for the player. The entire rest of the ai should look pretty familiar. It is exactly the same as how the player acts except for a few things. For one instead of passing anycollision the player we pass it the bady. Also if the bady collides with something on the left or right variable gets multiplied by -1. This means that if the bad guy collides with something he will turn around. And that's all there is to it, and now for the collision method.

collision

The collision method isn't to complicated. All it does is check for a collision and then makes sure the player is above the goomba. If the player is not above the goomba then the player dies.

```
void badycollision(objectinfo* object){
   if(boxcollision(object->x,object->y,8+object->size*8,8+object->size*8)){
      if(fix_norm(mario.info.y)+25<fix_norm(object->y)){
        object->alive=0;
        deletesprite(object);
      }
      else{
        //player dies
      }
    }
}
```

The begining if statement checks if a 16×16 box (because our goomba is of size 1 1*8+8 is 16) at the goomba’s location has collided with the player. If it has then we chek if the players y plus his height (so the y position of his feet) is above the goomba’s y (the top of the goomba’s head). If it is the goomba dies (his alive is set to 0) and we delete him. If it is not then well actually nothing happens but if you really wanted the player to die here like a real mario game then we would set the player’s alive variable to 0 and delete his sprite.

Download 3

Here is the source for our latest version of code. There’s some things we didn’t go over in there but you should be able to figure it out. Download

Closing

I hope this guide was able to show you how to get close to making a real platform game. I know some of it, especially the bit about pointers is confusing but I would suggest playing around with the code if you don’t understand it. But now it’s time to go work on finishing my own platform adventure. Good luck to all of you in your coding persutes, see you later.

Dustin

<back to top>

<day20>

Day 20 - Data Transfer and Multiplayer

These functions deal with data transfer, including sockets and HTTP. There are no particular Multiplayer examples here, but if you can communicate with a web server then you can create your own online system for handling data between consoles.

init

There is a template in the zip PAlibwifi : PAlibwifi_template

Use it to enable the use of the wifi in PAlib(or add -ldswifi9 in the makefile of the arm9 of your project : LIBSPA := -lpa9 -ldswifi9)

Before using the sockets, always use PA_InitWifi(); and PA_ConnectWifiWFC();

```
// Includes
#include <PA9.h>       // Include for PA_Lib
// Function: main()
int main(int argc, char ** argv)
{
        PA_Init();    // Initializes PA_Lib
        PA_InitVBL(); // Initializes a standard VBL

        PA_InitWifi(); //Initializes the wifi
        PA_ConnectWifiWFC();

        // Initialise the text system on the top screen
        PA_InitText(0,0);
        PA_InitText(1, 0);

        while (1)
        {
                PA_WaitForVBL();
        }

        return 0;
} // End of main()
```

How to create a socket?

PAlib has a function to create a socket easily : int PA_InitSocket(int *socket,char *host,int port,int mode); host can be either an ip or a dns, port is the port of the socket, mode is the mode of the socket : PA_NORMAL_TCP to a tcp socket PA_NONBLOCKING_TCP to a non-blocking tcp socket

```
int sock;
PA_InitSocket(&sock,"www.google.be",80,PA_NORMAL_TCP);
```

How to create a socket listening for incoming connections ?

int PA_InitServer(int *sock,int port,int mode, int num_connect); sock is the socket the listening socket will be made port is the port of the socket, mode is the mode of the socket : PA_NORMAL_TCP to a tcp socket PA_NONBLOCKING_TCP to a non-blocking tcp socket num_connect is the maximal connection que. Hardware limit is ~20.

```
int sock;
PA_InitServer(&sock,9999,PA_NORMAL_TCP,10);
```

After that, to wait for an incomming connection you have to use accept(). Accept returns the socket-stream, so:

```
int connectedclient;
connectedclient = accept(sock,NULL,NULL);
```

(You see those two “NULL”‘s , those aren’t required. They save info about the connectening client. Google for “accept socket c++” and you’ll find many infos about that function.)

Be warned, if you use a blocking socket the whole process hangs up (=blocks) until a connection comes in.

How to receive a message from the server?

To read a socket, you can use recv() There are 4 parameters : the socket, the buffer, the size of the buffer and 0

```
char buffer[256];
recv(sock,buffer,256,0);
```

How to send a message to the server?

To send a message, you can use send()

```
char buffer[256];
strcpy(buffer,"hello world");
send(sock,buffer,256,0);
```

How to get a file with http?

You can easily get a file in a buffer with http : int PA_GetHTTP(char *buffer, char *adress);

```
char *buffer = new char[256*256];
PA_GetHTTP(buffer,"http://www.google.be/index.html");
```

How to get the IP address of your DS?

```
char *buffer = malloc(256*256);
PA_GetHTTP(buffer,"http://www.invisionsoft.co.uk/ip.php");
```

How to disconnect from the access point?

There is no known PALib function for Wifi disconnect as of now, but this function duo from the Wifi library itself will work:

```
Wifi_DisconnectAP();
Wifi_DisableWifi();
```

<back to top>

<day21>

Table of Contents

Saving and Loading from FAT

This portion of the wiki will deal with basic saving a loading functions. It will provide some quick examples of basic saving and loading functions that can be implemented into almost any game. If you have any questions, comments, or suggestions, feel free to post email them to me at GEMISIS@live.com

Initializing FAT

Initializing FAT for saving and loading is fairly simple. Call all of you basic initializations for palib, then call **fatInitDefault();** to initialize the FAT system. Note: **fatInitDefault();** is an libfat command, and is not a part of palib.

Here is a sample code with **fatInitDefault();** that should work (but does nothing):

```
//Includes
#include <PA9.h>
#include <fat.h>
```

```
//main
int main()
{
        PA_Init();    // PA Init...
        PA_InitVBL();   // VBL Init...

        fatInitDefault();

        while(1)
        {
                PA_WaitForVBL();
        }
        return 0;
}
```

Writing the Save Data

First we will go over how to write save data to a file.

Create the save structure

First you must create a structure for the save data. In it, you will include everything that you will be saving to the file. The structure for saving data in this example will use bool and int, but others should work too. Here is the structure we will be using for this example:

```
typedef struct
{
  bool character;
  int level;
}save_struct;

save_struct save_data;
```

Writing the Data to a file

Now we will actually write the structure to a file use the structure above. To do this, we will need to have defined our structure, and initialized FAT. Here is a sample code for writing save_data to a file:

```
//Includes
#include <PA9.h>
#include <fat.h>

//Save structure
typedef struct
{
        bool character;
        int level;
}save_struct;
```

```
//Save data
save_struct save_data;

//main
int main()
{
        PA_Init();    // PA Init...
        PA_InitVBL();   // VBL Init...

        //FAT initialization
        fatInitDefault();

        //Set the save_data values
        save_data.character = true;
        save_data.level = 2;

        //WB trunacates write and create.  This will be used to create and write to
the save file
        FILE* save_file = fopen("save_file.save", "wb");

        //Write save_data to save_file don't forget the & for save_data
        fwrite(&save_data, 1, sizeof(save_data), save_file);

        //Close the file
        fclose(save_file);

        while(1)
        {
                PA_WaitForVBL();
        }
        return 0;
}
```

First we set the values here:

```
        save_data.character = true;
        save_data.level = 2;
```

Then we open the file that we want to save the data to.

```
        //WB trunacates write and create.  This will be used to create and write to
the save file
        FILE* save_file = fopen("save_file.save", "wb");
```

After that, we write the data to the file with fwrite:

```
        //Write save_data to save_file don't forget the & for save_data
        fwrite(&save_data, 1, sizeof(save_data), save_file);
```

Then finally, we close the file.

```
        //Close the file
        fclose(save_file);
```

And that should work.

Reading the save data

Reading save data is pretty much the same as writing it, but instead we will load values into the structure. We will use the same info from Writing the save data in this too. Please note, that the values for the structure must be identical to the ones that you are loading.

The only thing that you need to change in order to read save data from a file is fwrite. You will change it to fread to read the data instead. Pretty simple, right? That will replace the structures previous values with the ones that are loaded.

Conclusion

In conclusion there are a few things that you should know about this tutorial:

1. It was written by GEMISIS :D
2. This will write it in binary, so don't think you can read it without a hex editor.
3. This code was written fairly fast, so some of it may or may not work.

As stated at the top, if you have any questions, comments, or concerns, you can contact me at GEMISIS@live.com

<back to top>

End of Tutorial

This is take from http://www.palib.info/wiki/doku.php
Last updated: 2/23/09<build2>
Have a nice day!
Happy coding,
And may the force be with you!