Game Teleporter

By Tony Morelli

12/15/2006 morelli@cs.unr.edu

Abstract

Developing software for multiple platforms is a very difficult and tedious task. It is also very difficult for someone who does not know any programming language to create games to demonstrate a talent for game creation. What is proposed here is a solution to both of these issues. A software tool, written from scratch, is used to convert a certain input file format to a different output file format. A plug-in architecture is used in such a way that a programmer can generate either an input plug-in or an output plug-in which is used in the application. This paper will show why this is a good way to create games and demonstrate how it works with two input plug-ins, Adobe Photoshop and Macromedia Flash, and one output plug-in, Playstation Portable.

Introduction

The simple creation and conversion of a game is important as it is a common problem in the development of games. Although not insurmountable, the obstacle of converting games is a very time consuming process. Saving time also saves money and with the budget of video games constantly increasing a tool such as the Game Teleporter will be extremely accepted within the industry.

This project is also important because of the educational aspects. The finished plug-ins can be used in early programming courses as an introduction to programming. Advanced courses can be created that will look at the plug-ins more in depth and allow students to modify existing plug-ins and create new plug-ins. Students will have already been familiar with the tool and looking at it from how it was coded will help them understand the software better as they already know the functionality. Jumping into code is always harder when the complete picture is not known. Knowing exactly how an application works in all situations makes it a lot easier to go through the code and figure out why it works that way. If this tool is used throughout a student's educational career, the student will be much better prepared for real world programming projects.

The concept of this project has had a lot of thought put into it, and is extremely simple on the surface, and extremely complex underneath. Developing this project involved lots of reverse engineering of different file formats. Several different file formats were investigated for input plug-in formats, and even more were looked at for output plug-ins. Outside of reverse engineering some formats, others were learned by looking through file specifications released by open source groups and by the manufacturers themselves. This project demonstrates the ability to create a needed application and to write the code for the application using industry standards when available and inventing new ones when the existing standards are insufficient.

This project looks extremely basic on the surface. Just simply take one file format and convert to the next. The complexity shows up when writing plug-ins for complex file formats. The file format must be known completely and all functions and tags within certain file format must be decoded correctly. Also all the intricacies of the target platform of the output plug-in must also be known. The Nintendo DS has different requirements as far as resolution of the screen(s), supported picture types, and audio formats than the Playstation Portable. These differences need to be taken into consideration when creating an output plug-in.

The inherent problems of this project are visible when something else is needed to be represented within the core of the Game Teleporter in order to support a new feature. For example, a picture is read from the input plug-in and stored in an intermediate format before the output plug-in reads it and does whatever conversion it likes. In this example each of the input plug-in, the core, and the output plug-in are all aware of the temporary image and what formats it should be in. If, for example, the input plug-in has a new feature such as a 3-D surface, the core would need to be modified to support the surface in order to properly pass it on to the output plug-in which would also need to know what to do with this object. This is not unexpected, and must be supported and considered as new file formats and file types will always be on the horizon.

Problem Formation

The object of this project is to simply take one target format of a finished project and automatically output a finished project of a different target format. This will make it easier for people to develop games. Although it does not set a standard for how all games need to be developed it sets up the framework for all varieties of possible development types. It also allows for flexible integration of new features as well as new input and output types of files. The Game Teleporter is organized in such a way that integrating features is a relatively easy task, although the actual coding of the features may be difficult. The difference here is the actual functionality of the new feature and the integration of the new feature. Features of the future cannot be known, but this project should allow for easy integration of whatever they may be.

An in depth look at the functionality of the Adobe Photoshop file format is a large part of this initial demonstration of how the project functions. Also a part of this is how software for the Playstation Portable is built and run. These are taken as an example and not as a complete guide. As will be shown throughout this paper, the Photoshop file format contains a very large set of commands, and for this project, only the necessary functions are implemented in the input plug-in. The same goes for the Playstation Portable output plug-in. The available commands and features of the Playstation Portable are very large. This plug-in simply generates the necessary source code to correctly convert the sample Photoshop (PSD) file. Again, the object of this project is to demonstrate how the Game Teleporter should function, not a complete plug-in for all Photoshop features, or all Playstation Portable features. A list of supported features is documented within this paper.

Background

Current methods of converting games from one platform to another can be a very long and drawn out process. Currently game developers must take the graphics, sounds, and game description and create the game. If the game has been created for the PC, and is to be run on the Playstation Portable, the game is basically started from the ground up again. This is a large waste of time and with the cost of game development rising, any reduction in the development process is a great asset. And being able to get the game out on multiple platforms at the same time allows the final product to reach a far greater audience quicker and more efficiently.

Most attempts at doing this have succeeded to some degree. One of the best examples is Java. Java is a platform independent language. The compiled code runs through an interpreter, called the Java Virtual Machine, which is platform dependent. If the Java program is desired to be run on a different platform, the interpreter is simply installed on the target. The same program can be run on Microsoft Windows and on Linux. This works out very well, but it has two side effects. First of all, the code is not as efficient as it could be as the interpreter is taking the code and changing it into code that the OS will understand run time. Instead of just feeding the code into the CPU, the interpreter must convert the Java program into instructions that processor can understand, then running the program through the CPU. Another disadvantage of this approach is that a Java interpreter is required on the target machine. If you wish to run your Java program on a platform that is currently unsupported, you must make a Java interpreter to run on that platform. This would require a lot of work, and could be an acceptable solution, however some platforms, especially embedded platforms, do not allow for the overhead of the Java Virtual Machine. It would be very difficult to port a version of the Java Virtual Machine that would run at an acceptable rate on the Gameboy Advance.

Another attempt at a platform independent language is Flash, originally developed by Macromedia. To run Flash files, a platform specific flash player is required. This has the same drawbacks as the Sun Java Virtual Machine, with one difference. Adobe now owns the rights to Flash, and they require all Flash players to be released by them. So if the target platform does not have a Flash player written for it, a request must be made to Adobe to create a player for that target platform. This would most likely take a lot of time and money to get completed.

The best attempt at creating a cross platform file format is web browsers. All html files are meant to be read on all platforms that have a web browser written for it. This works really well, and many platforms are supported. However, although web pages are interactive, they are not the best suited for games. Games require immediate reaction when buttons are pressed, and the loading of web pages is too slow to be run real time. Also, web pages use hyper links to move from one page to the next and the use of arrow keys to make movements is not directly supported.

My Approach

The approach described here creates platform independent native machine code. The Game Teleporter will convert from one file format to another format without the need for a runtime interpreter. This will allow the

program to run as fast as possible on the target platform.

The idea for this came from a different project I completed in the past. In this project, the first cross platform multiplayer game was created for the most recent generation of handheld gaming consoles. In that project the simple game of Tic Tac Toe was implemented in both the Playstation Portable and in the Nintendo DS. Players were allowed to play a game of Tic Tac Toe against a person playing on either the same handheld, or on the opposite handheld. The code for each was written in C, and a lot of the code was re-used unchanged. A compiler for each target machine was run on the code generating a binary image that could be run natively on the respective machine. The code segments that were specific to each console were specific to the same elements. For example, the method of drawing an X or an O on the screen was different between platforms. The initialization of the screen was different on each platform, and the methods of printing characters to the screen were different. Both platforms needed to print characters to the screen at the same time, both platforms needed to initialize the screen at the same point in the code, however the actual function calls were different. This left an opening for another layer of abstraction. An intermediate higher level code could be used above the C layer to generate the code, and then another layer in the compilation process could be used to generate the specific function call. For example, the function InitializeScreen() is needed in both the Nintendo DS and the Playstation Portable. The above generic function call is made, and is then converted to the known specific set of function calls for each platform. This is where the idea for this project was born, and it has been developed as described in this paper.

An input file is submitted to the application with the desired output format specified. In this case, an input file can be either a Adobe Photoshop file (*.psd), or a Flash file (*.swf). The desired output must be a Playstation Portable as that is the only supported output plug-in. The application is then run and outputs source code that can be compiled in the psp gnu tool chain. The Game Teleporter is designed to run on Windows XP, although the code is written in C++ for easy porting to other operating systems.

Installation of Necessary Components

To test the software, several components must be installed on the development station. First of all, in order to test the different input plug-ins, a computer must be installed with Adobe Photoshop and Macromedia Flash. These utilities are necessary to generate the input files. The current code has many limitations and the capabilities of the plug-ins are described later in this document.

To test the output plug-in, a Playstation Portable development environment must be installed. The process for installing the development environment is somewhat tedious, but if the following directions are followed correctly, there should be no issues getting up and running.

The first step in installing the Playstation Portable development environment is to install cygwin. Cygwin is available at http://www.cygwin.com. Cygwin is a Linux-like environment for windows which will give a user the appearance of working within the linux environment.

The Playstation Portable compiler runs in this environment so it is necessary to first install cygwin and make sure it is functioning correctly. To install cygwin, click on the install icon on the web page indicated earlier. This will begin the installation process. The default selections are sufficient until the setup reaches the Select Packages stage. In this stage, select whatever packages are of interest. If this is unfamiliar territory, be sure to select everything under the *devel* category. The devel territory contains the gcc compiler which will be used later. It is also a good idea to install a text editor of choice under *Editors* at this time. VI is a great editor and was the editor of choice for this project. Specific packages that are needed include, autoconf2.1, automake 1.9, gcc, gcc-g++, make, diff, patchutils, subversion, wget. Do not worry about selecting all the correct packages, if something is missing it will alert the user that an update is needed. To update, simply go back to the website and click on the update icon. It will determine what is installed on the development machine, and changes can be made via the Select Packages screen. If all of this cygwin business is unfamiliar, or is seemingly complicated, have no fear. This is all just used to create files for the PSP output plug-in. The plug-in will do most of the work. This is all just up front installation that is required to be done only once. The default selections are fine for the rest of the installation. Just sit back and let it do its thing. This part can take awhile depending on the speed of the internet connection of the development machine.

The second step in installing the Playstation Portable development environment is to install the PSP Tool Chain. The PSP Tool Chain contains libraries necessary to build images suitable for running on the Playstation Portable. The tool chain is avail at www.sourceforge.net. Simply search for psp tool chain and download it to within your home directory in cygwin. For example, if cygwin was installed to C:\cygwin, the home directory resides in c:\cygwin\home\<username>. Once downloaded, run cygwin and type *ls*. A listing should appear and should contain the PSPTool Chain. My directory contains a psptoolchain-20060120.tar.gz. Untar this file using the command, *tar –zxvf psptoolchain-20060120.tar.gz*. This will create a directory called psptoolchain-20060120. Change directories into this newly created one by the following: *cd psptoolchain-20060120*. Then install the toolchain by typing, *./toolchain.sh*. This will complete the installation of the required tools for the Playstation Portable output plug-in.

The ability to run unsigned code on the Psp is slightly complicated and also slightly understood. Sony designed the Psp to run only software signed by official developers. Since we will be writing software to be run on a Psp, we either need to become official developers, or find some kind of hole that Sony has left for us. My preferred method is known as the *kxploit* method. This method only works on version 1.5 firware Psps. For other versions of Psp Firmware there are other methods of booting unsigned homebrew code including the buffer overflow or the Grand Theft Auto exploit. But a little more in depth about how the chosen method of the kxploit works. First off there are 2 directories created with the target name. If the target name is defined as TARGET, two directories are created, one named TARGET and

another named TARGET%. The real unsigned code lives inside of TARGET, and the contents of TARGET% contain an actual signed piece of code released by Sony for other purposes. The accidental feature of firmware v1.5 is that the section of code which checks the validity of the file about to be ran does not support the percent sign, so it is dropped. The code wants to check the validity of TARGET%, but ends up checking the validity of the contents of TARGET (which are valid). The function returns the piece of software has been determined to be a valid Sony software, and the execution piece executes the binary file inside of TARGET%.

Another required installed component is ImageMagick (http://www.imagemagick.org). ImageMagick is an open source freely distributed graphics software package which is accessed through a command prompt. Being accessible through the command prompt is important as easy conversions can be done within the application, instead of invoking a GUI and requiring user interaction. In this specific project, ImageMagick is used to export individual layers of a PSD file into separate PNG files.

Description

A normal development process usually starts off with an artist creating graphic images, a sound engineer creating the sounds, and finally a programmer putting everything together to create a game that the public likes to play. If the game is going to be released on multiple platforms, for example the Playstation Portable and the Nintendo DS, the programmer will start the project over from scratch using slightly modified graphics and sound files. Although this process is straight forward, it can be time consuming and certain aspects can take time. Using this application, all of the standard conversions are handled internally by the conversion utility, and is transparent to the developer and the audio and graphics teams. If the output plug-in is properly written, all of this is taken care of automatically. The use of quick code conversion has been demonstrated in a project where a simple Tic Tac Toe game was created which allowed cross platform multiplayer games to be played over the internet between a player on a PC, Playstation Portable, or Nintendo DS. C/C++ compilers do a lot of the work as the code remains portable, however the code can be broken down into even more basic building blocks to allow for the porting of the code. Also, there currently is no common sound file formats across all platforms. This tool will allow for that.

Another reason for this tool is to allow non programmers to enter the industry by showing off their skills. People who can draw might have an idea for a game to create, however it is very hard to put that idea into existence as they do not know how to program. Using this tool with correctly implemented plug-ins will allow artists to create a game and show the game running on real hardware to perspective employers. Also, this tool can be used for a introductory course in game development as the students can create games without knowing the in depth knowledge of how the coding process

works. Once the students are very interested in the programming aspect, they can have the opportunity to modify plug-ins or create new plug-ins as new standards and technologies come with time.

In this project, the input plug-ins demonstrated are Adobe Photoshop and Macromedia Flash. These are two very well known programs used by artists in the gaming industry. Using these tools as a basis for demonstrating the functionality of the Teleporter is important because professional artists are very comfortable with these software packages. Instead of having to relearn a completely new software package, this utility will let the artists use the tools he wishes to use as long as the appropriate input plug-in exists. The conversion process will notify the artists that the picture drawn either conforms or does not conform to the specifications of the output plug-in. For example, the proper resolution of the Playstation Portable is 480x272 pixels. A correctly coded Playstation Portable output plug-in will have a check for this value. If the original Flash or Photoshop file is not the correct size, the output plug-in should either abort the conversion process, or automatically scale all files appropriately.

The project was originally started using Adobe Photoshop as the only input plug-in. As this project focuses mainly on games, the Photoshop format quickly became too basic. Games require interaction and dynamic environments that are difficult to create within a Photoshop document. Simple programs were able to be created using this as the input plug-in, however to create a fun game a different input plug-in was needed. The second plug-in created was the Macromedia Flash plug-in. This contained the necessary elements to actually be used as an input plug-in for a complex game. This also created an issue as the code for the input plug-in itself became extremely complex. This is acceptable as the complex code is isolated to the Flash Input Plug-in code, and when properly functioning, this code is rarely modified or seen, simply executed.

Input Plug-ins

As described earlier, input plug-ins must support converting any files, or set of files into an intermediate set of files. These files are grouped into two areas, source code and resources. The source code contains commands in a simple language similar to the C programming language. Currently, the only resources supported are images stored in the Portable Network Graphics (Png) format. Other resources that can be added at a later date include sounds, 3 Dimensional Graphics, and other data files.

The Photoshop input plug-in is designed to create a slide show of images that are contained within a Photoshop file. Adobe Photoshop has support for multiple layers. A layer is similar to a page in a notebook made up of transparent paper. A drawing exists on each page of the note book, and when laid on top of each other it is possible to see all images at once. That is what is seen when the Photoshop document is first opened. The Photoshop Plug-in's function is to take the different layers (pages in the notebook), break them out into separate images, and then generate code to display them one at a time keeping the location of the images the

same as where they were in the original Photoshop file.

To achieve this goal, an open source tool was utilized. *Imagemagick* is an open sourced and freely distributed graphics package which contains several utilities to manipulate graphics. Although it is very possible to write complete software solutions not using open source products, they are very useful and it avoids re-inventing the wheel for known problems. In the first stage of the Psd conversion process, the Imagemagick utility *convert* is ran on the Psd file. When convert is run on a psd file, the output of the command is one Png file for every layer contained within the file. This is a very important step to have automatically taken care of by an outside utility, however the convert utility is missing two important pieces of information. It is missing the location of the image within the layer, and it is also missing the name of the layer. The position within the file is needed to properly display the image, and the layer name is needed for potential key word usage later if the psd plug-in is desired to do more than a simple slide show.

To obtain the two missing pieces of data, the position of each layer, and the layer name, the Game Teleporter was written to get these pieces directly out of the Psd file itself. The Psd plug-in is a stripped down version of the Adobe Photoshop 6.0 File Formats Specification. It is only concerned with the name of each layer and the location of each layer. Nothing more. The plug-in will display all sorts of information about the file and about each layer, however that information is simply displayed and not saved for later use. The plug-in can be modified in such a way that available and not currently stored data can be stored, but that is not necessary for the purpose of this plug-in. The code was written to follow the spec and upon completion of processing all data within the Psd file, the plug-in is aware of layer names, and the corresponding bounds which include the coordinates of the Top Left corner, and the Bottom Right corner of the graphic contained on each layer.

Knowing the layer names, the plug-in then makes modifications to the Png files created by the convert program earlier. The convert program knows nothing about layer names, so it names the files according to the layer number. So what is output is a number followed by .png (i.e. 1.png, 2.png, 3.png, etc…). This is somewhat not useful, but fortunately the layers are processed in the same order by the Game Teleporter. So the psd plug-in can simply rename each file based on an index into an array of filenames generate by the psd plug-in when it read the file earlier. At this point, the input plug-in has created the resources necessary for the intermediate stage and copies them to an intermediate directory. The final step of the input plug-in stage is to generate the source code needed for an output plug-in to create the desired output.

The Psd input plug-in creates two source code files. The first file created is named Pictures.h and contains a known structure for pictures. The information stored within that file contains the path name of the image resources, and their associated bounds. A sample Pictures.h file is shown below:

```
/*****************************/
static struct Picture picts[] = {
  {"Green_Box.png",1,1,34,45},
  {"Red_Box.png",0,432,35,479},
  {"Blue_Box.png",224,0,265,51},
  {"Yellow_Box.png",226,427,270,478}
```

```
        };
#define TOTAL_PICTURES 4
/*****************************/
```

Figure 1: Pictures.h

The above code is generated runtime by the Psd Plug-in and includes all relevant info including the total number of pictures.

The second piece of code generated by the Psd Input Plug-in is the code that describes the necessity for displaying the slide show. This code is formatted in a C like syntax in order for ease of conversion for most out put plug-ins. Most platforms support some kind of C compiler, so that is why that format was chosen. The main code piece not only contains commands, but it also contains XML markers designating different sections of the code. The xml tags generate by the Psd Plug-in include, Includes, Declarations, and Mainloop. The includes section is designed to describe any external files created by the plug-in, which in this case is Pictures.h. The Declarations section is much like a declaration section in a C programming where it outlines what variables are about to be used. The mainloop is a section which contains the actual commands to generate the desired output. A sample output of the Psd Plug-in code generation is shown below:

```
/*****************************/
<INCLUDES>
#include "Pictures.h"
</INCLUDES>
<DECLARATIONS>
int picIndex = 0;
InputButton button;
</DECLARATIONS>
<MAINLOOP>
  GetInput();
  if (button & RIGHT)
  {
    ClearScreen();
    picIndex++;
    if (picIndex >= TOTAL_PICTURES)
    {
      picIndex = 0;
    }
    DisplayImage(picts[picIndex]);
  }
  if (button & LEFT)
  {
    ClearScreen();
    picIndex--;
    if (picIndex < 0 )
    {
      picIndex = TOTAL_PICTURES-1;
    }
    DisplayImage(picts[picIndex]);
  }
</MAINLOOP>
/*****************************/
```

Figure 2: Intermediate Code Sample

A quick run through of the code shows that our main loop calls a function GetInput which is then assumed to place whatever buttons are pressed in a variable labeled button. Then based on what button is pressed (either right or left), the screen is cleared and the next image is displayed. This keeps going on forever. This is a pretty simple state machine, and it will be shown later how this simple code is turned into Playstation Portable Code.

Output Plug-ins

Output plug-ins is designed to utilize the code and resources generated by the input plug-in to create a desired result. The plug-in discussed here is a Playstation Portable plug-in, but the concepts discussed here can be used to create plug-ins for different platforms.

The Psp Output Plug-in begins with a skeleton set of files which act as a

starting point for a Psp Project. These are grouped into two categories, Necessary Files, and Modifiable files. The Necessary Files are files necessary to build a Psp Executable, and are not able to be modified by the Psp plug-in. These files are simply copied into every project directory. The other set of files, the Modifiable Files, are files necessary to build a Psp Binary, however these need to be modified based on the output of the input plug-in.

The two files being modified are *Makefile*, and *main.c*. The changes to Makefile are extremely basic. The Makefile contains two identifiers for each Psp Project, Target and Title. The Target defines the executable name and shall not contain any spaces. The Title is what appears as the title on the Playstation Portable Game Selection Screen and may contain spaces. Both of these configuration options are entered on the command line when the application is started and inserted into the file when the Psp Output Plug-in is ran.

The second and most complicated modifiable file in the Psp output plug-in is main.c. The skeleton main.c file contains all the include files, functions, and definitions required for all Psp applications. It also includes keywords such that the Psp Output Plug-in can insert the necessary code at the correct places within the file. The file contains XML tags for Includes, Declarations, and the MainLoop. Since this plug-in is creating an application to be built by a C compiler, the plug-in can simply copy in all includes defined in the intermediate code file. They will be a direct drop in. The same goes for the declarations with one exception. The variables are declared in the same fashion as in the intermediate code, however the Psp Output Plug-in must maintain an internal list of variables for use which will be described later in this document. The variable types supported by this plug-in are an Integer and a Button type.

After the declarations, the MainLoop Tag is encountered and the guts of the code are added from the intermediate code generated by the input plug-in. The Psp plug-in reads through the intermediate code and makes modifications as necessary. For example, any time the intermediate code makes a call to GetInput(), the Psp plug-in makes a call to ProcessInputs, and stores the results into its variable that has been declared as a button. In this case, the variable name is *button.* Another substitution deals with the use of the variable. Whenever the intermediate code checks the value of *button*, the Psp output plug-in generates code looking at the value of *button.buttons* as that is the naming convention for buttons in the Psp world. The intermediate code generated by the input plug-in as shown above is translated into the following code which is understood by the Playstation Portable compiler.

```
/**************************/
 button = ProcessInputs();
 if (button.Buttons & RIGHT)
 {
  ClearCurrentScreen();
  picIndex++;
  if (picIndex >= TOTAL_PICTURES)
  {
   picIndex = 0;
  }
  DisplayImage(picts[picIndex]);
 }
 if (button.Buttons & LEFT)
 {
```

```
   ClearCurrentScreen();
   picIndex--;
   if (picIndex < 0 )
   {
    picIndex = TOTAL_PICTURES-1;
   }
   DisplayImage(picts[picIndex]);
 }
/*************************/
```

Figure 3: Psp Output Plugin Generated Code

As shown above the code translates pretty easily into Psp native code, but also illustrates why a C compiler cannot be relied upon to do all conversions.

With all the code in place, the plug-in then copies all resources into the destination directory. This sets the game up to be built. One last modification needs to be made prior to building the game. A build script is generated which contains the name and directory of where the source files will be placed. A batch file is then run which will enter the build environment and build the Playstation Portable Binary image.

When the newly created application is run on the actual hardware, a slideshow is presented to the user. The slideshow cycles through all the layers originally drawn within the Photoshop file and displays them in their original location. This simple example illustrates the power of this tool. From a Psd file, to running on real hardware in less than 1 minute is something not shy of amazing.

This software was designed to be run completely from the command line as it will enable people to use the utility in scripts that will automatically generate a desired output. This, however, is not the best approach for people who are using this tool as a development kit for non programmers. These people may be intimidated by the requirement of using a command prompt. A front end for the command prompt, *GT Front End*, was created. This utility allows the user to input all required command line options, such as the input and output files, and the output file names into a Graphical user Interface (GUI) that is in a windowed environment that the user is very comfortable using. This will promote the use of this tool by people who are not completely computer savvy.

Code Walkthrough

The Game Teleporter is made up of a VC++.Net application, and a VB.Net graphical front end. In addition to those two, there are several files associated with the plug-ins. The demonstration files associated with the Photoshop input plug-in and the Playstation Portable output plug-in are functional. The following section gives a more in depth look at the code and sample files used in the above mentioned components.

Game Teleporter VC++.Net Code Walkthrough

Flash.cpp
Flash.h

These files contain the base Flash class. This class inherits from InputPlugin, and is used to decode Adobe Flash Version 8 and lower files. This class is a work in progress, and currently only decodes results and displays them to the screen via printfs. The difficulty in creating this plug-in lies in the decoding of the script language that Flash uses. This is has been partially completed, and is not an impossible goal, however more time is needed for complete implementation of this scripting language. As shown below, a lot of the ground work has been

accomplished, however the details of emulating this still remain to be completed.

FlashDefines.h
This file contains definitions for known values of commands and data types encountered throughout the Flash SWF file.

FlashStack.cpp
FlashStack.h
These files are a class that defines the stack that Flash uses to store commands. Commands are pushed and popped to the stack, and this class adds, removes, and accesses these commands. The plug-in will eventually decode this stack as generic source code which will be used by an output plug-in.

FlashStackEntry.cpp
FlashStackEntry.h
These files define a class for one entry on the stack. This includes the data type of the entry on the stack, as well as what value that entry currently holds.

GameTeleporter.cpp
GameTeleporter.h
These files contain the Game Teleporter code outside of the plug-ins. This code is very basic, and simply calls the correct input plug-in for first stage decoding, and then calls the correct output plug-in for second stage decoding.

Image.cpp
Image.h
These files define the class for an image. The information stored within this class includes the path to the image, and image attributes such as image bounds, whether or not it is a necessary image, and a string identifier.

InputPlugin.cpp
InputPlugin.h
These files define the base class for all input plug-ins. All input plug-ins must inherit from this base class.

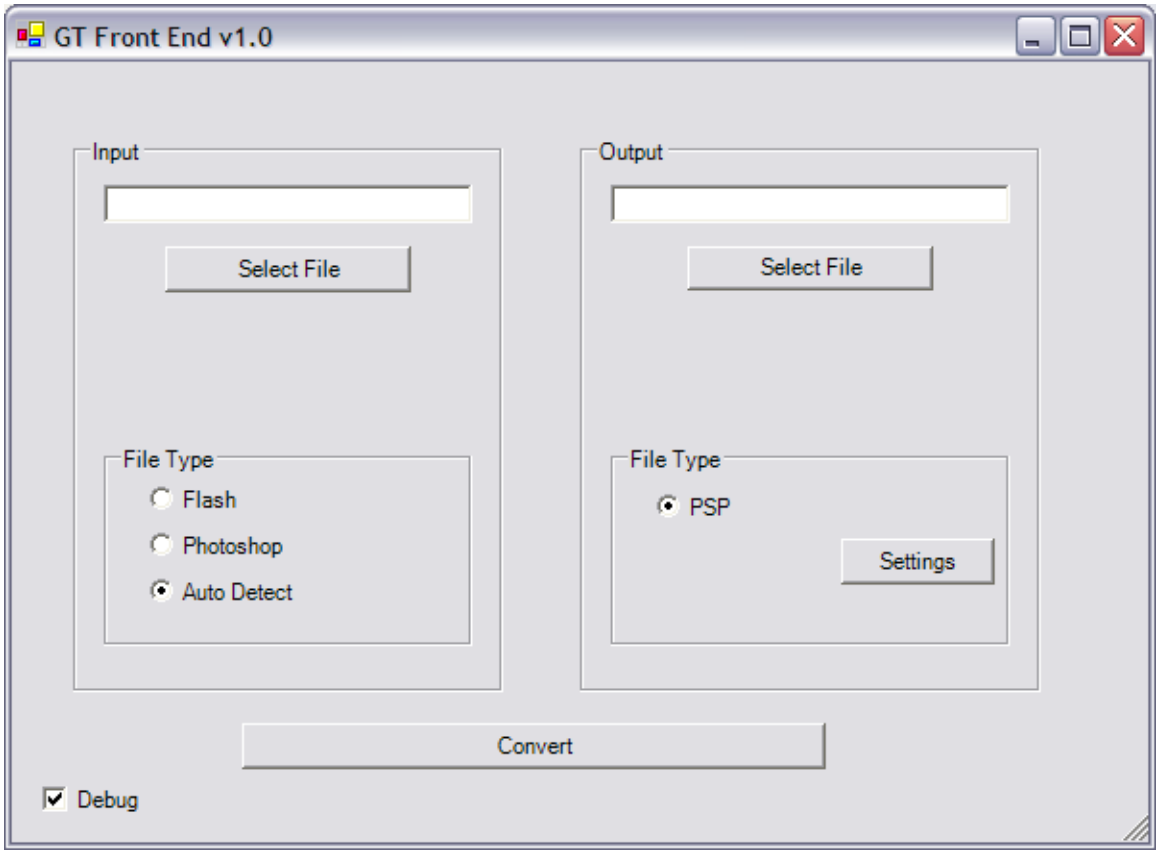
OutputPlugin.cpp
OutputPlugin.h
These files define the base class for all output plug-ins. All output plug-ins must inherit from this base class.

Psd.cpp
Psd.h
These files define the class for the Photoshop file format decoding. This class inherits from InputPlugin. It takes all layers in an Adobe Photoshop file and saves them off as PNG files named as their layer names. ImageMagick's *convert* creates the PNG files, and then the PNG files are named according to their layer names which are pulled directly from the PSD file.

PspOut.cpp
PspOut.h
These files define the class for the Playstation Portable output plug-in. This class inherits from OutputPlugin. This class takes the intermediate files created by the input plug-in and creates a Playstation Portable executable image.

GT Front End VB.Net Code Walkthrough

This software is a Graphical User Interface to make running the Game Teleporter usable by anyone. A screen shot of this application is shown below:

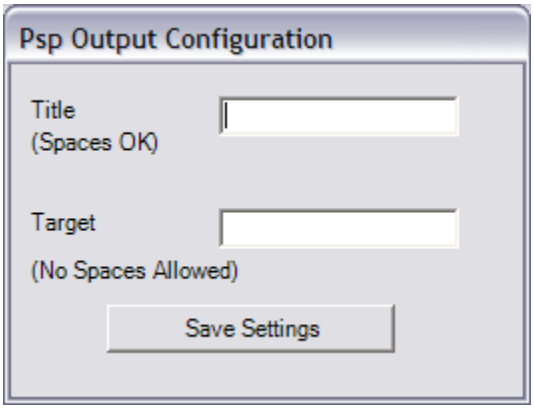


Form1.vb

This file contains the main code for the entire interface. This includes allowing the user to select the input file and the output file, along with the type of each. When the Debug check box is selected, the output of the GameTeleporter is shown in a Dos Window only for displaying Debug information.

PspOut.vb

This file contains the code for the user to set up specific Playstation Portable build information.



This setup screen contains two entries: Title and Target. Title is the very nice title which will be displayed on the selection screen when the user runs the application. The Target is the directory name that the executable code will be

stored on the Playstation Portable and must not contain any spaces.

Plug-in Directory Walkthrough

The demonstrated plug-ins are organized into the following

```
\plugins
\--\Input
\--\--\Psd
\--\Intermediate
\--\Output
\--\--\psp
\--\--\--\final
\--\--\--\nochange
```

Psd Input Plug-in

By default, the Psd input plug-in directory is empty. The Psd plug-in uses this directory as a temporary scratch storage area and all files are removed after running.

Intermediate

This directory is written to by the input plug-in and read from by the output plug-in. This directory will be empty when the input plug-in is run, and upon completion of the input plug-in running it will have files ready for the output plug-in. The contents of the *intermediate* directory will contain any images needed, any header files created, and a file called *code.gtc*. This file will contain the commands that the input plug-in needs executed. It will be up to the output plug-in to read this file and convert the commands inside of it into commands the target will understand.

Psp Output Plug-in

This directory contains several files required for the Playstation Portable Output Plug-in. First off, the files contained in the *nochange* directory are files required to build all Psp applications. The *final* directory contains the completed source code for the project as wells as any assets needed, in this basic case images are the only assets. Only two files are modified by the Psp Output Plug-in, main.c and Makefile. The Makefile contains tags to enter the Target and Title of the application. The main.c file contains tags for entering additional header files, additional variable definitions, and a tag for the main loop of the application.

Conclusion

This paper has demonstrated how the application can translate one file format into another. It shows how easy new plug-ins can be created for either a new input or a new output format. The application was designed with the intention of it to be used by everyone at all programming skill levels. A younger student who desires to create a game can do so by simply drawing pictures and running it through the proper plug-ins. A higher level student can write more plug-ins, or modify the existing ones to demonstrate programming abilities, or to learn new ones. This broad range of users and uses makes this tool a great part of the education process. The ease of adding new plug-ins will keep students very interested, as well as allowing this piece of software to evolve with time.

References:

Adobe Photoshop 6.0 File Formats Specification Version 6.0 Release 2 November 2000. Copyright 1991-2000 Adobe Systems Incorporated

Macromedia Flash (SWF) and Flash Video (FLV) File Format Specification Version 8 Copyright 2005 Adobe Systems Incorporated