Deconstructing Flash

Investigations into the SWF file format

Simon Wistow BEng III <smw3@doc.ic.ac.uk>
June, 2000

Project Supervisor: Ian Harries <ih@doc.ic.ac.uk>
Second Marker : Iain Phillips <iccp@doc.ic.ac.uk>

Abstract

Macromedia Flash is a web orientated vector animation format. Using it, web developers can easily place complex and attractive multimedia animations on their web pages. Additionally these animations scale well, due to their vector nature, have very small file sizes and are untroubled by browser incompatibilities.

Flash is stored in highly compressed binary files known as SWF (pronounced 'Swiff') files.

The major draw back to Flash is that although the format allows some interactivity, it is very limited. 'Action Scripting' allows the movie to react to key presses and button clicks of the user however it is not dynamic in the same way that a database driven web site is. These allow the information presented to a visitor to be freshly generated each time the page is viewed, reacting to up to date information without any outside influence.

This project aims to remedy this situation.

Acknowledgements

I would like to thank :

My supervisor, Ian Harries, and my second marker, Iain Phillips for their guidance and advice during this project.

Mark Fowler <mark@twoshortplanks.com>, who had the original idea for this project, for going through this document pointing out the obvious (and not so obvious) errors and omissions.

James Freeman and Victoria Knowles for exhaustively proof reading this report and saving me some quite serious embarrassment over my spelling and grammar.

Jon Peterson <jon@snowdrift.org>, for some constructive criticism.

The members of the London Perl Mongers and the (void) mailing list, in particular Leon Brocard <leon@astray.com>, Greg McCarroll <greg@mccarroll.demon.co.uk> , Dave Cross <dave@dave.org.uk>and Dave Cantrell<evildave@oven.com> for their help with the intricacies of Perl.

Introduction

Report Structure

In this report I aim to demonstrate the need for this project. To do this I will explain a little on the history and nature of the Flash format.

I will then go on to show how I planned to go about achieving my aims and then will demonstrate how they will actually achieved, the difficulties encountered and what I learnt in doing so.

Finally I will critique my achievements, give some suggestions for extending the project and also some final thoughts on the project.

Aims

Flash is a web orientated vector animation format.

Create SWF generation library

My major aim is to create a library to allow the dynamic, programmable generation of Flash/SWF movies that would overcome the limitations described in the previous section viz. that Flash content has to be (re) created by hand each time a change has to be made.

Investigate feasibility of a more comprehensive library

Whilst an SWF generation library is useful the project could be extended to be vastly more powerful.

A generic library for the representation of SWF files could be written such that any program could create abstract representations of SWF files and manipulate this structure from within themselves. This SWF object could then be saved out as either an SWF file or some other file format. Theoretically by providing a CORBA (Common Object Request Brokerage Architecture) interface these objects could even be shared between distinct applications dynamically.

To my knowledge nobody else has ever attempted to do anything quite like this before and therefore there will be little or no help available.

Study the SWF File Format

The SWF format is very compact and by attempting to understand it a greater understanding of compression and animation techniques will be gained.

Applications

CGI (Common Gateway Interface)

Two current vogues in Web publishing are database driven web sites and increasingly visual sites. The first allows information to be kept current without having to continuously edit and re-edit raw HTML. This is achieved either by using CGI programs, which can be written in almost any language but are most often done in Perl, or by using a server-side scripting language such as ASP, ColdFusion, or PHP. These are directly embedded in the HTML (Hyper Text Markup Language, the *lingua franca* of the web) document.

The second trend is for increasingly visual sites. Whilst Flash is perfect for this sort of application with current tools there is very little you can do to dynamically change the content in a movie once it has been initially designed.

However using this project, a site could be created that had a weather map on it. Every time the site was accessed a script would retrieve the latest weather forecast and then dynamically generate an animation showing the weather. Apart from a little design at the start and some fairly simple programming this would be quick and easy to achieve and with far more attractive (and informative) results than generating a bitmap picture (using something like the GD graphics creation library). Once it was set up it would be completely automated.

Auto Generation

There are some things that whilst easy to do by calculation, are difficult to do by hand. An example would be drawing a circle or even drawing objects arranged in a circle. Tasks like these are almost impossible to achieve with the current Flash user interface due to their complexity.

It would also be useful to able to do repetitive tasks with Flash. For example, a company wants to send out a number of individualised Flash movies to people with each movie bearing the recipient's name in some form. Currently this would have done by hand, each movie laboriously being edited and saved one by one by a person.

However the whole process, even the emailing, could be accomplished quickly and simply by using a list of names and addresses and writing a program that utilised this project.

Library

The final use that can be envisaged for this project would be as the base of another application designed around Flash. Theoretically a library, as long as it was sufficiently low level, could be used to write an Open Source alternative to Macromedia's authoring tool.

An application could be created with a drawing interface that used the Flash library and which stored, retrieved and updated structures within a Flash object depending on user input. It would then output the result as an SWF file (or some other format such as SVG, the Standard Vector Graphics Language).

Background

Flash History

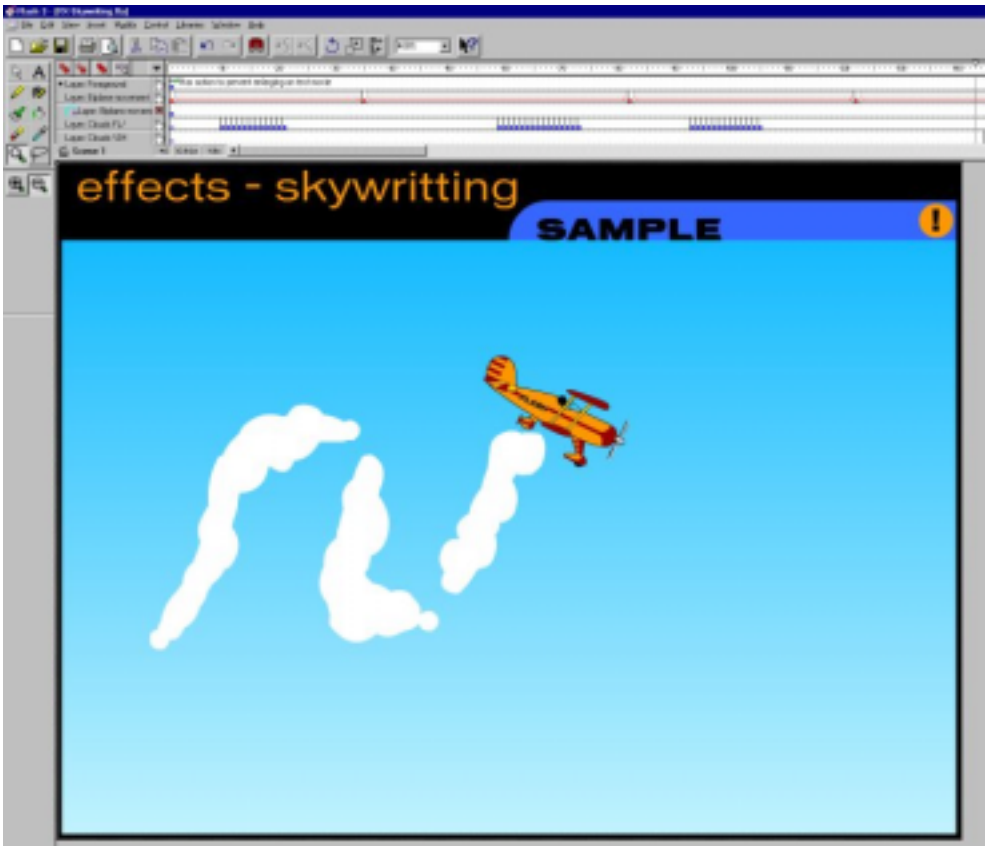
Futuresplash

In 1995 a company named Futurewave created a product named Futuresplash, a vector animation standard for the web. Despite having several merits and being critically acclaimed it was limited to a brief appearance on 'The Simpson's' web site and being used on Microsoft's doomed "Microsoft Network" content provider.

Flash/SWF

During the same period as Futuresplash was developing, Macromedia's Director, a multimedia-authoring tool, aimed at CD-ROMs and kiosks, was reinventing itself to cope with the collapse of the multimedia CD-ROM market and was becoming more Web focused. Designer refugees from this fallout, snapped up by the burgeoning web industry, were already familiar with the Director creation tools and began using them despite the fact that their bitmap core was unsuitable for the web.

Macromedia realised this and in 1997 bought the ailing Futurewave, renamed the product Flash and converted the technology into one of many the Shockwave plugin family for Macromedia's own Director and Freehand application. One by one the other plugins disappeared but the name Shockwave remained. Eventually a standalone Flash creation app was produced leaving only the MIME types "application/futuresplash" and "application/x-shockwave-flash" and the confusing file extension SWF (Shock Wave Flash) as a hint to Flash's convoluted past.



The Flash Authoring Tool

Vector versus Bitmap

In computer graphics there are two predominant ways of drawing a two dimensional picture, vector and bitmap. For example to draw a circle a vector graphic simply tells the display program to "draw a perfect circle exactly 100 pixels in diameter." A bitmap program maps out every pixel in that circle, which requires more information (and therefore a larger file size) often resulting in a poorly rendered circle. In addition to this, a vector-based image can be scaled to virtually any size with no impact on file size or quality

Bitmap *Vector*

This is a simulation of the differences between bitmap and vector graphics. The left circle demonstrates how a bitmap image is defined by a finite set of pixels. Enlarge it, and the circle becomes rough and granular. However, by increasing the resolution to counteract the "pixely" look, the file-size becomes much larger. Whereas, the vector graphic on the right could be enlarged 10 times and still look just as sharp.

Penetration

"Flash has become a standard, it really is out there"
- John Warnock, Chairman and CEO, Adobe Systems.

Before using Flash content on their sites, developers want to know what percentage of Web browsers will be able to see it. Because of the wide pre-installation of Flash and its ease of download, this percentage is very high.

In March 2000, NPD Research, the parent company of MediaMetrix, conducted a study to determine what percentage of Web browsers have Flash preinstalled. The results show that 89.9% of Web users can experience Flash content without having to download and install a player.

IDC Research estimates the total number of users online to be 248 million. Since Flash users represent 89.9 percent, then 222 million users have the Flash player already installed.

In December 1999, a similar study was conducted by NPD Online, showing Flash Player was installed in 85.6% of browsers. The March 2000 study by NPD, therefore, shows a rise of over 4% in Flash Player penetration.

There are Flash players available for Netscape and Internet Explorer under Windows and Macintosh and also for Linux and Solaris. There is also a Java player version that theoretically means that any platform that can run Java can view Flash movies.

Flash Capabilities

It is important to separate the features of the Flash authoring tool and the Flash file format. For example the Flash tool has the facility to morph between shapes, tween images and provide smooth paths for objects to travel along. However these are not built into the File format (for example, by providing two sets of co-ordinates and an image and expecting the client side player to generate all the images between).

Most of the Flash format revolves around defining shapes. Shapes can be constructed out of primitives such as free, straight line, vertical line, horizontal line and curves. These shapes can be filled using either gradient fills or bitmap images. Colours are stored as RGBA (Red, Green, Blue and Alpha) values.

Natively SWF allows the inclusion of several bitmap file formats in addition to the vector objects. Support is present for lossless JPEG (Joint Photographic Experts Group), lossy JPEG and Zlib compressed PNG (Portable Network Graphics) files, which are all common graphics formats

Once defined images and shapes can be manipulated by applying transformation matrices to them in the form of positional and colour transforms.

Sound can be stored either as ADPCM (Adaptive Differential Pulse Code Modulation) or MP3 (MPEG layer 3) sound formats with varying levels of compression and can either be triggered by an Action (explained later) or started at a particular frame in the movie.

Flash can define buttons within itself with various actions triggering different responses such as starting sounds, placing a particular object on the screen, skipping a number of frames or jumping to a different part of the movie.

The final, main feature of Flash is its ability to store what Macromedia term ‘sprites’. These are embedded, independent, slightly limited Flash movies and are useful for showing little animations in windows within the main movie. These can control, and be controlled from, the movie in the main window.

Open source

Macromedia published the specifications for SWF, up to and including the Flash 3 extensions, in April 1998 in the form of a document detailing the binary structure the file format. Through experimentation it was realised that many aspects of the format were documented incorrectly and an effort has been made to correct these parts. In addition parts of the more recent Flash 4 extensions have been reverse engineered.

Additionally a company named Middlesoft was commissioned by Macromedia to create a SDK (Software Development Kit) for writing SWF files. This was finally released in May this year.

Similar products

Macromedia Generator

Macromedia realised the potential of dynamically generated Flash quite early on and released their Generator program in 1998. It provides some simple manipulation of objects, some simple text replacement and generation of graphs. It works by combining SWT (Shockwave Template) files with a text configuration file and data from external sources like HTTP requests or database resources. It is currently on version two and is very expensive; the Enterprise edition, which implements some caching (and which could be done in this project using open source products such as Squid), costs upwards of $30000. The Developers edition costs $999 per processor and is not really suitable for hosting web sites on. It is only available for Windows NT, Linux, and Sun Solaris.

It is not possible to create files from scratch but can be accessed from ASPs (Active Server Pages) and ColdFusion generated pages.

Swift Generator

This attempts to recreate the functionality of Generator, even going so far as to using the same configuration file format. It is free as long as a logo is placed upon the web site using it.

It is available for several operating systems as a pre-compiled binary.

It also cannot generate movies from scratch and is not programmable in that the API (Application Programming Interface) is not available to external programs.

Turbine

Turbine is another attempt to provide the same functionality as Generator, available for several platforms. It costs around $400.

It is more powerful than Swift Generator and has some caching built in as well as some support for Flash 4 extensions and limited file size optimisation.

Libswf

Libswf is Paul Haeberli's attempt to produce a programmable Flash generation library. It has several bugs and is only available as a statically linked binary. It has not been updated since February 1999 and the author has not responded to any emails.

Middlesoft SDK

Macromedia commissioned a San Franciscan company named Middlesoft to write an SDK to aid in the creation of SWF movies. It is similar to libswf but is more comprehensive and comes with source code. Unfortunately there are some stiff licensing terms attached to it and the only available version is for Windows.

Swish

Swish is a tool for creating text effects in SWF such as exploding and expanding sentences. It is designed to allow you to import these effects into Flash and costs $30. It is only available for Windows and cannot be used to create dynamic Flash files for CGI purposes.

Saxess Wave Frame

Saxess, formerly Visiweb, is a tool written in Java to convert XML/XSL (eXtensible Markup Language / eXtensible Style Language) files to SWF files. It has a slightly different angle to Macromedia Generator *et al* but is essentially the same idea. It is free and runs under any Java capable platform.

Form2Flash

Form2Flash can change the text in any SWF movie be it text that appears on screen, URLs for the movie to jump to or frame labels and sequencing commands. It is free and available as source code files for Unix and binary executable for Windows.

Swifty Utilities

These are a suite of utilities for extracting sound, images and movies from SWF files. They are free but only available for windows.

Java/SWF wrappers

This comprises a number of Java classes that can read and write binary SWF, and export/import to/from an XML representation. The wrapper classes can also be programmatically constructed.

It is free and open source but was only started a few months ago and has not progressed very far.

Ming (PHP)

Ming (who was the enemy of Flash Gordon) is a module for the PHP server side scripting language. It is relatively new and allows you to programmatically create SWF files from within PHP scripts. It can only do text and solidly filled shapes.

It is free and open source.

Several of these products and projects have similar or overlapping goals but none has quite the same scope or flexibility that is envisaged for this project.

All of them suffer from one or more of the following faults

- Extremely limited in what they can achieve.
- They are closed source or restrictively licensed meaning that they cannot be built upon reliably or bugs fixed.
- They are only half completed

Therefore none of these can be used as a suitable base for this project.

Design

Basic Format of SWF file

An SWF file has two main components – the headers and the frames. The headers store information such as the size of the movie, the version, how many frames it has, its size and the frame rate.

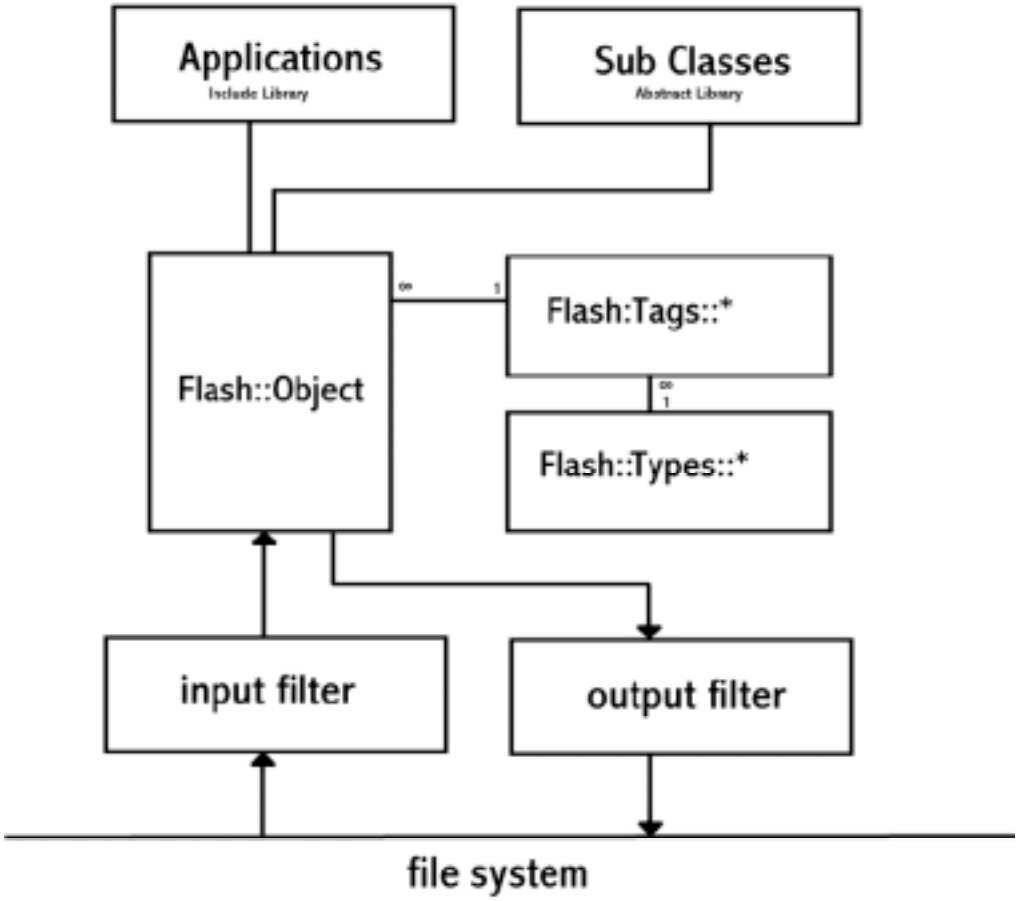
The rest of the file is taken up by frame information. Within each frame it is possible to define, place and manipulate objects by defining 'tags' but the frame is not actually shown until the end of the frame is reached (marked by the ShowFrame tag, id `0x01`). Tags are separated into two types; definition and control tags.

Definition tags define objects such as shapes, buttons, sprites (embedded Flash movies), text, sound and embedded images. These can be thought of as the actors in a movie.

Control tags control the flow of the movie. They are responsible for placing, modifying and removing the actors from the screen. There are also Actions scripts that define how the movie should react to user-generated events such as key presses and mouse clicks. These action scripts can start and stop sound, alter the frame of a movie or an embedded movie and launch a browser window.

Object Structure

After some consideration the following object structure was decided upon.



The container object, Flash::Object, is fairly simple. It contains an array of frames that are themselves arrays of Tags. It also contains some arrays so that font definitions can be stored and referenced.

It provides a few utility functions, the most important of which is add_tag that takes a Tag object and places it in the current frame array. If it is a ShowFrame tag then the frame number is advanced.

The other utility functions provide a variety of methods such as translating between Tag names (such as SetBackgroundColour) and Tag codes (such as the hex number 0x09 in the case of SetBackgroundColour).

Individual Tags are all represented as individual objects that can have any number of variables that can either be scalars, arrays hashes or Flash data types.

Data types are convenience objects that represent a particular group of data within an SWF movie. The simplest of these are things like Colours, and Transformation matrices. However they can get much more complicated with objects like Shape records being quite complex.

Why Object Orientated?

The structure of an SWF file is a series of frames each with a series of tags. Each tag can have any number of variables that could be a Flash data type. Each data type can, in turn have any number of variables that can also be Flash data types. These tags and data types will have to be manipulated and extended individually and so object structure was the easiest and most efficient design methodology, providing neat packaging of relevant variables and functions.

For example the module Flash::Types::Colour has four variables : r(ed), g(reen), b(lue) and a(lpha). This makes it easy to modify the colour and the transparency. However many functions and objects needed the hex value of the colour but it was simple to add the method `rgba_hex` to the Colour module such that:

```
sub rgba_hex
{
  my $self = shift;

  $self->a(0xff) if ($self->a eq "");
  return ($self->a << 24) | ($self->r << 16) | ($self->g << 8) | $self->b;


}
```

Role of Input and Output filters

One of the main features of the library is the ability to create a Flash object using arbitrary input and output features. These provide a way of creating and storing Flash movies in a number of formats.

Currently only one input and one output filter have been written. The first input filter was from a binary SWF file itself. This reads in an SWF file and loads a Flash::Object

with it. This object is then passed to the output filter, in this case one that prints out the data contained within in the same style as the reference parsing code from Macromedia.

However it should be possible to create more filters, for example to read and write a movie to XML or to a database and obviously to export an object back to an SWF file. Furthermore it would make the conversion of SWFs to other formats such as SVG and VML (Vector Markup Language).

The reason for this is that there will be certain jobs that are easier to do in one format than it is in another. For example a designer could draw the basis of a movie using the Flash authoring tool since this would be the best format for storing complicated graphics and sound. This could be loaded into a Flash::Object. However you could also then take some extra data that has been stored in an XML file because it is easily parseable, both by machine and by hand, and load that in after the data from the SWF data. The Flash::Object will automatically renumber every object ID so that there are no conflicts. In this way the best of both worlds can be achieved.

Filters can be used to increase the speed of parsing. Whilst a binary compressed file is useful when transferring movies across the Internet. However when reading files locally the complexity of parsing them shifts the bottleneck from the bandwidth of the connection to the time it takes to parse the movie. As such it would be much better to read the movie in from a binary file and then save it out as something far easier to parse, such as an SWF file.

Another use of input and output filters is to extract limited amounts of information. For example you may want to manipulate only the sound or the text within an SWF file in which case it is pointless to parse the shape data. This can be achieved by simply writing a reduced set of filters.

Language choice

The language of implementation is extremely important as the strengths and weaknesses of the language chosen will be reflected in the project. It was decided that there were 3 main choices:

C/C++

C is a language designed by Dennis Ritchie and Brian Kernighan in order to write the Unix operating system. It is an imperative, strongly typed, compiled language that is very fast but has poor handling of strings. The language has been designed to be as portable resulting in the definition of an ANSI standard to ensure compatibility. However care must be taken with programs written in C to ensure that they will compile on as many platforms as possible.

C++ is an Object Orientated version of C written by Bjarn Stoustrup. It has some very powerful object orientated features but can be very complicated to learn and can also be non-standardised between implementations, even on the same platform.

GNU Objective-C language is a cleaner, object orientated superset of ANSI C and provides classes and message passing similar to Smalltalk. It is much easier to learn but there is relatively little support for the language at this time.

Since all the example code for SWF manipulation is in C or C++, coupled with the execution speed benefit of programs written in one of these two languages, would make it logical to use either of them. However they have been rejected for several reasons.

Pure C is not object orientated and it had already been decided that an Object Orientated approach would provide the greatest flexibility for extending the library when the base had been done. On the other hand I was not familiar with C++ or Objective C.

Secondly, since the exact file format was still unknown it was anticipated that there would be a large amount of experimentation. C's rigidity would have made it unsuitable for rapid prototyping.

Thirdly, one of the main applications of this project would be the dynamic generation of web content. Whilst some CGI programming is done in C, its poor string handling makes it rather unsuitable for anything that requires a lot of text processing.

Perl

Larry Wall's language, Perl (Practical Extraction and Report Language) is a weakly typed interpreted language with excellent string manipulation capabilities. It is the most commonly used language for writing CGI scripts and has a vast, centralised library of add-on modules in the form of the CPAN (Comprehensive Perl Archive Network). It also has a very close knit and friendly community.

The features of Perl that are attractive are that the programmer does not have to worry about the details of memory allocation (all arrays are dynamically sized and all garbage collection is automatic). Additionally, whilst it is high level enough to allow rapid prototyping it is also powerful enough to not be a limiting factor.

It also has some powerful Object Orientated features such as multiple inheritance and polymorphism.

As mentioned before, Perl is also the *de facto* CGI scripting language and is extremely good for batch processing making it a natural choice when considering two of the projected applications of this project.

The Perl language allows you to write extremely cross platform code. The Perl interpreter has been ported to many different platforms such as virtually all Unix variants to Apple Macintoshes, Windows machines and even Psion personal organisers. Scripts written on one machine run on another, usually with no modification. Additionally, the low level features of Perl that depend on particular platforms are clearly marked in the documentation and only deal with areas that will not concern this project (such as interprocess communication).

On the other hand Perl is not as fast as C especially when it comes to complicated programs. The overhead from having to fire up the interpreter each time your script is run can be quite large, although this may be offset by using caching technology such as the mod_perl Apache module which can pre-compile and cache the program and all the modules it uses.

The weak typing in Perl may cause difficulties when trying to ensure that the right types are passed to each object.

Java

Java is a programming language developed by Sun Microsystems designed to be 'platform-independent'. It achieves this by compiling bytecode. This means that a Java program can be run on any operating system that has a Java Virtual Machine (JVM) and Java Runtime Environment (JRE) implemented for it. Generally, the JVM is incorporated into a web browser (such as Netscape or Internet Explorer) and allows the user to download and run Java code straight from the Internet. This means that the user does not have to download and install the additional Java Runtime Environment (JRE), a large application with a somewhat complicated installation procedure. Furthermore, sections of code are only downloaded as and when they are necessary, meaning that download-size is often reduced.

The flexible but simple object orientated aspect of Java are strong points in its favour since it would have forced an Object Structure to develop very early on. However this was also a limiting factor since more flexibility was needed when prototyping.

Although there is already a Flash player written in Java that would have been interesting to interface with the project, it was felt that this would mean it would be constrained by the needs of the player.

Also, although using technology known as servlets, Java can be used as a server side scripting language, the technology is not especially mature and I had no experience in it, which would make it difficult to realise one the major aspects of my project.

Finally, with a large project, compilation of .java files to .class files can be quite time consuming. The final, compiled code is also not much faster than interpreted Perl code (especially when it is run under mod_perl).

Final choice

I decided to use Perl to implement the project as it was felt that whilst it certainly had its downsides it would be easiest language in which to implement a prototype. It was also reassuring to know that there was access large support base in the form of the London Perl Mongers programming group. It was also the language I was most practised in.

Implementation

Perl Semantics

Perl programs are generally known as scripts since they are not compiled. To run them the Perl interpreter is first loaded and then the script is loaded into it, compiled to an internal byte code format and then executed.

Since version 5 Perl has been extendable by libraries, known as modules. These modules are generally written in an object-orientated manner so that to use their features it is necessary to an instantiate an object. This is done for scoping reasons; object orientating your module prevents the name space of the main program from being polluted with, possibly conflicting, function names. However it is also possible to write modules such that the features can be accessed as normal, inline functions.

Modules are arranged hierarchically into families. The naming of modules depends on what family they are in and is read like standard file paths only using '::' instead of a delimiter instead of '\' or '/'.

So for example the File::Lock and File::Copy modules both belong to the same 'File' family.

The production cycle

File::Binary

The first thing that was required was some way of reading the binary SWF files from within Perl. Although there are functions to read and write individual bytes sequentially from files within the language, there are no functions or add on modules to handle the reading of more fine grained, or even larger, binary data types such as signed and unsigned bits and 16 bit and 32 bit words.

In order to overcome this obstacle it was necessary to write a module for manipulating these files which, using the nomenclature of Perl, was named File::Binary.

As mentioned before Perl modules are traditionally object orientated. For example 'normal' file handles can be opened as objects as well as the more usual scalar variables.

```
my $filehandle = new IO::Handle;
$filehandle->open($filename);
print $filehandle->getline;
$filehandle->print("Some text\n");
$filehandle->close;
```

This module was written to be object orientated too.

To use it the user creates a new File::Binary object

```
my $file = new File::Binary($filename)
       or die "Cannot read file $filename : $!\n";
```

At which point the constructor of the module opens up the filename that has been passed to it, returning a new object if it succeeds or an error if it does not.

The object keeps track of several variables inside it, such as the current position in the file and the current bit offset.

Reading bytes simply calls the in built file handle function ‘read’

```
sub get_bytes {
  my ($self, $bytes) = @_;

  # set the default to one byte
  $bytes = 1 unless (defined $bytes);

   my $data;

  $self->{filehandle}->read($data, $bytes, 0);
  return $data;
}
```

When wanting to read a number of bits the object first checks to see if there are enough bits already in the bitbuffer. If not, it reads in a byte and tacks it on to the current bitbuffer and then reads the bits.

```
sub get_bits {
  my ($self, $bits) = @_;

  # set the default to one bit
  $bits = 1 unless (defined $bits);

  my $data = 0; # the return value

  for (;;) {

    # we want to know if we should use the whole byte.
    my $s =  $bits - $self->{_bitPos};

    if ( $s > 0 ) {

      # all these bits are ours
      $data |= $self->{_bitBuf} << $s;
      $bits -= $self->{_bitPos};

      # get the next buffer
      $self->{_bitBuf} = unpack("C", $self->get_bytes(1));
      $self->{_bitPos} = 8;

    } else {

      # this is our last byte, take only the bits we need
      $data |= $self->{_bitBuf} >> ( -$s );
      $self->{_bitPos} -= $bits;

      # mask off the consumed bits
      $self->{_bitBuf} &= 0xff >> (8 - $self->{_bitPos});
      return $data;
    }
  }
}
```

There are also a number of other convenience methods.

```
$file->where()
```

returns the current file location

```
$file->seek($pos)
```

allows arbitrary seeking to a position within the file.

The function get_word returns a 16 bit word

```
sub get_word {
  my $self = shift;

  return  unpack("C", $self->get_bytes(1)) | unpack("C", $self->get_bytes(1)) << 8;
}
```

The only problem with this module has been the handling of signed bits. Since Perl is quite high level it performs some internal trickery to handle signed bits and hides the fact that they are signed for the user's convenience. As yet, no way has been found of stopping this. This has prevented the reading of negative signed numbers correctly.

It is hoped that this will be fixed correctly in the next version since there is interest in including this in the list of standard Perl modules included with Perl itself (providing copyright issues with college can be resolved).

Reverse Engineering

Once a satisfactory method of obtaining binary data reliably in Perl had been achieved it was possible to investigate SWF files further. Using the specification it was possible to write a small program that would run through a SWF file, first extracting the header information and then finding the code and length of each tag. It then printed that information out onto the screen before seeking to the next tag.

From there began the fairly laborious task of writing a function to parse each tag. In many cases the specification released by Macromedia was incomplete or wrong and the only way to test this was to create a simple SWF file with the required tag in it and alter the parameters one by one until the correct values could be confirmed.

A simple example of a conversion from the specification to actual working code is given below.

```
ButtonReserved UB[4]
Reserved bits - always 0

ButtonStateHitTest UB[1]
Button state hit test flag

ButtonStateDown UB[1]
Button state down flag

ButtonStateOver UB[1]
Button state over flag

ButtonStateUp UB[1] Button
state up flag

ButtonCharacter UI16
Button character ID

ButtonLayer UI16 Button
character layer

ButtonMatrix Matrix Button
character matrix

ColorTransform  CXForm
Character color transform
```

=>

```
sub button_record
{
    my ($byte,$colour_matrix) = @_;

    my $record = new Flash::Types::ButtonRecord;

    my $pad = $byte >> 4;
    $record->state_hittest ( ($byte & 0x8) );
    $record->state_down    ( ($byte & 0x4) );
    $record->state_over    ( ($byte & 0x2) );
    $record->state_up      ( ($byte & 0x1) );

    $record->character     ( $bin->get_word );
    $record->layer         ( $bin->get_word );

    $record->matrix        ( get_matrix () );


    # bit confused about this one the spec
    # mutters about nCharactersInButton
    # and getting as many Cxforms
    # as that but it always seems to be one :/
    $record->cxform ( get_cxform )
                      if ($colour_matrix);

    return $record;
}
```

Of particular difficulty were the tags that contained data for external data formats such as JPEG images and MP3s.

First prototype

Once it was considered that all errors had been discovered and corrected a first prototype was constructed. In the quest for optimisation the designers of the file format have made use of some clever tricks such as having global counters, lists of offsets in the file pointing to specific information and lists of pointers to shapes.

Text, for example, is not stored as a list of letters, but as a pointer to a list of pointers to shapes.

It was therefore necessary, to preserve these ‘global’ variables otherwise the output filter would not have all the relevant information.

It was at this time that various data types were identified and isolated and a preliminary version of the object structure was attempted.

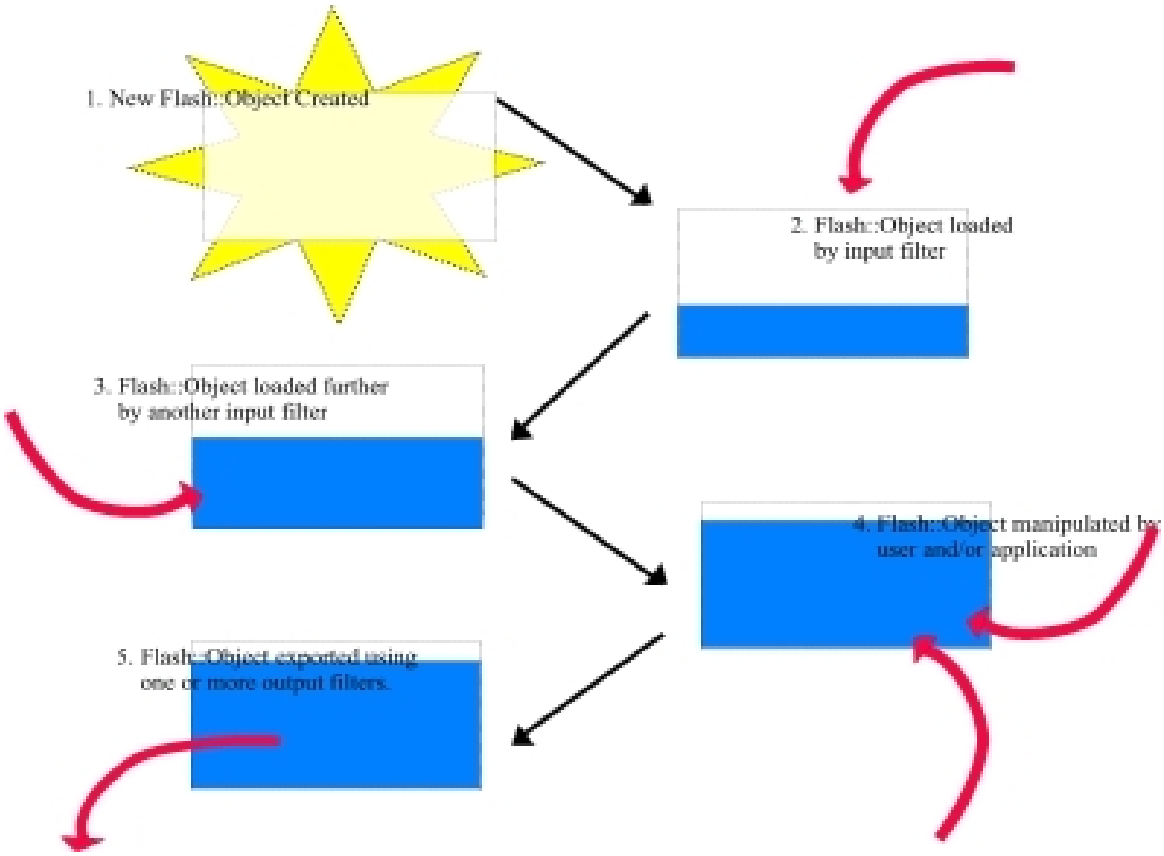
Second prototype

The second, and final, prototype took the lessons learnt in making the first prototype and used them to create a proper, formal object structure. The previous version had used lists of hash-indexed arrays. This was because the preliminary object structure was changing

rapidly as new lessons were learnt and having to go through and change all the formal definitions would have been extremely time consuming.

Unfortunately by the time the object structure had been finally formalised, a suitable output filter had been written and the whole structure and parser debugged there was little time left to write further input and output filters.

The overall flow of the application was finalised to resemble



Steps 2 and 3 are optional.

XS wrapping of libswf

During the design and implementation of the object structure a parallel effort was underway to create a Perl library to generate SWF files. The reasons for this were two fold. Firstly it would provide a simple iterative interface for generation of simple SWF files using primitive building blocks. Whilst theoretically it would be possible to create almost any SWF file using this library in practice having to define each shape line by line would be time consuming at best.

Secondly, by interfacing this primitives library with the object structure however, it would be possible to build up complicated shapes by importing them from an SWF file generated using the Flash authoring tools.

At first it was hoped that this library could be written entirely in Perl, but it was soon realised that this was unfeasible due to time constraints. Instead it was decided to ‘wrap’ an existing C library, Paul Haeberli’s libswf, with what is known as an XS wrapper.

XS is a language used to create an extension interface between Perl and some C library that one wishes to use with Perl. The XS interface is combined with the library to create a new library that can be linked to Perl. An XSUB is a function in the XS language and is the core component of the Perl application interface.

The XS compiler is called xsubpp. This compiler will embed the constructs necessary to let an XSUB, which is really a C function in disguise, manipulate Perl values and creates the glue necessary to let Perl access the XSUB. The compiler uses typemaps to determine how to map C function parameters and variables to Perl values. The default typemap handles many common C types. A supplement typemap must be created to handle special structures and types for the library being linked.

To wrap the C library it was necessary to create an interface definition in the XS language. For example for the function described in libswf's header file as

```
void swf_getfontinfo(float *aheight, float *xheight);
```

is wrapped using the glue code

```
Void
SWF_get_font_info()
   PPCODE:
   {
      float ah,xh;
      swf_getfontinfo(&ah,&xh);
      EXTEND(sp,2);
      PUSHs(sv_2mortal(newSVnv(ah)));
      PUSHs(sv_2mortal(newSVnv(xh)));
   }
```

the function can then be called from within Perl

```
my ($Aheight, $xheight) = get_font_info();
```

This wrapping was largely successful and simple SWF files can be generated using text, shapes, full alpha blending, sound (ADPCM and WAV) and images (GIF, JPEG, PNG and BMP).

There are, however, some problems with this approach namely that. the library it is based around is closed source and has a number of bugs. For example according to the documentation that came with the library it should be possible to write the generated SWF movie directly to STDOUT by calling the open function with the reserved word 'STDOUT'. However this causes the library to segfault. Clearly this is not suitable for

CGI generation of SWFs (which need to be written directly to STDOUT) and so a workaround was created. When a file was opened using the reserved word a flag was set and a new filename was generated based on the current system time and the process id. A temporary file was then created with this name and, whilst it is still theoretical that a filename collision could occur, it is extremely unlikely.

When the file is finally closed the temporary file is copied to STDOUT and then the temporary file is deleted.

Furthermore, libswf cannot take insert graphics and sounds as native SWF format in the form of a variable. Instead it must be passed the name of a file, which it then opens and inserts into the movie. This means that when creating an SWF file from an object structure you must first save images and sounds as temporary files and then passes the temporary file names to the library. To say that this is merely 'time consuming' is a huge understatement since reading and writing from disk is vastly slower than using memory alone. It could also produce more temporary-file name collisions and errors.

Libswf is only available as a pre-compiled binary for three platforms: Linux, FreeBSD and Irix. It has not been updated since February 1999 and emails to the author have gone unanswered. This does not make it suitable as the base of a proper library.

However, this said, the implementation does provide and good proof of concept and allows a base upon which can be built upon at a future date.

Evaluation

Key Features

Successes

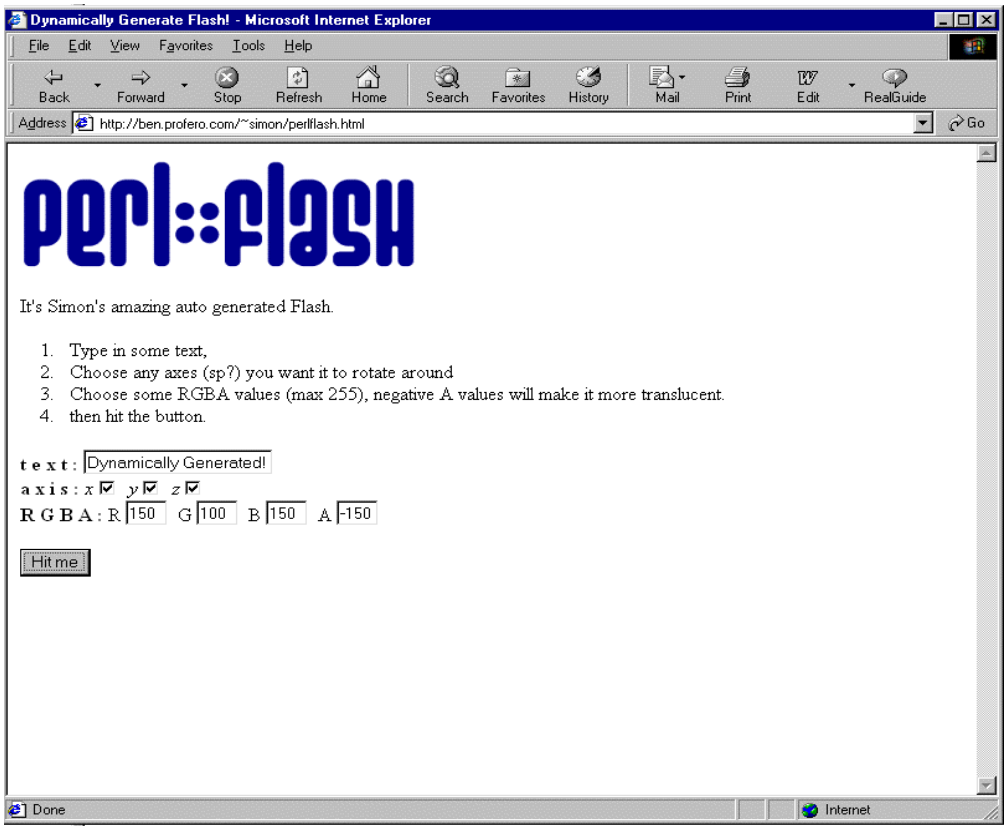
A successful binary file manipulation module has been written for Perl that may be included in the standard Perl libraries in the future.

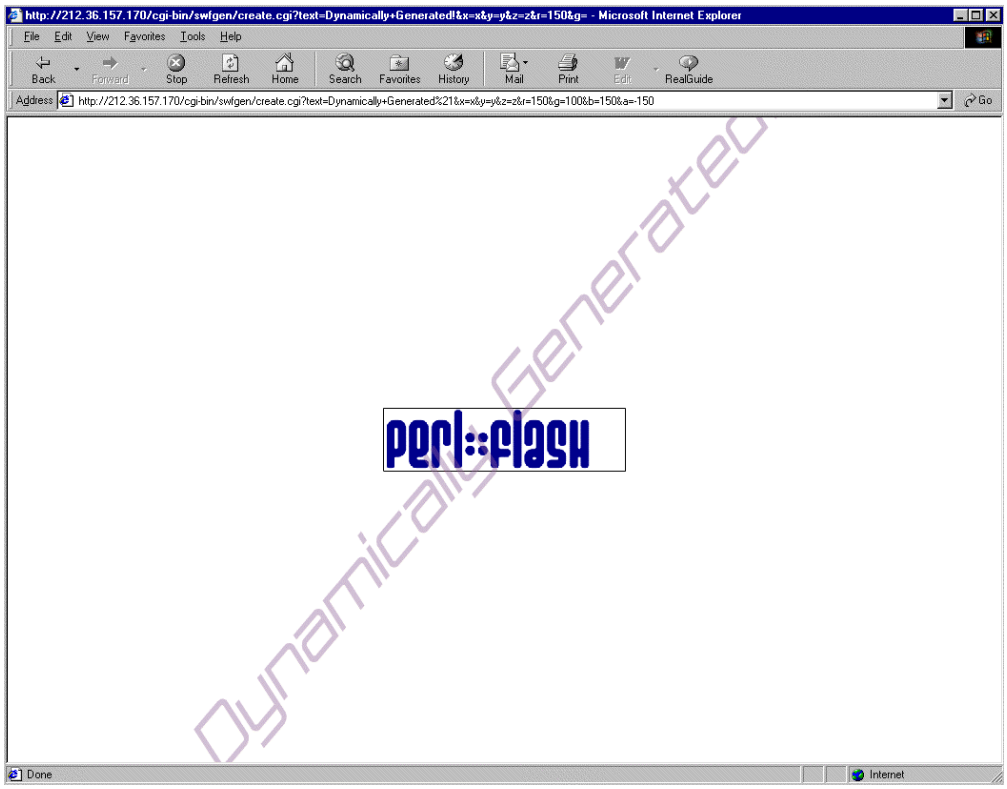
There is a comprehensive object structure representing an SWF file and the framework for arbitrary input and output filters is complete. This has come with a far greater understanding of the SWF file format that should be able to act as a reference model to other people working on other projects in the same field.

The decompiler covers every single one of the tags and data types in the file format. Whilst there is obviously room for improvement this makes it easier for anybody else to add further import and output filters. It also means that there should be minimal work involved in getting to the stage where it is possible to decompile an SWF file into an object structure, alter it slightly and recompile it again.

A comprehensive but primitive SWF generation library has been completed for Perl. it is capable of generating SWF files with Text, Shapes, Alpha transparency, Action scripts, Buttons, Sprites, Images and Sounds.

An example of this can be shown below.





The result

The Perl script that generated this took the parameters passed to it from the form and generated the SWF file with an image loaded behind the text and the text being rotated 1 degree in the x, y and z axes every frame. There are 360 frames at 30 frames per second.

The script is 50 lines long including white space, blank lines, comments and CGI initialisation routines.

Notice the image in the background and the fact that the text has been made semi-translucent. What cannot be heard is the looping sound in the background.

Uncompleted

- ✘ No interface between the object structure and the generation library
- ✘ Minor bugs still remaining in parser.
- ✘ No XML input and output filters done. Although this would not take very long there was simply insufficient time.
- ✘ Text replacement not completed.

Difficulties Encountered

This section will try and describe some of the difficulties encountered whilst undertaking this project.

Lack of knowledgeable people

There are very few people in the world who are that knowledgeable about the SWF file format, less than 100 certainly and probably less than 50. Many of these people work for companies such as Macromedia, Corel and Adobe and are not able to talk about the format, or their work, due to NDAs (Non Disclosure Agreements). The initial open source community that sprang up around the original release of the specs is now fragmented and a response to questions on the mailing list is rarely forthcoming.

As such it was difficult to get any help from people when difficulties were encountered and even fairly simple problems sometimes took far longer than would otherwise have been necessary.

Lack of binary file libraries for Perl

The lack of any binary file manipulation modules for Perl was somewhat surprising since there are many modules for far for obscure purposes. Lacking knowledge of the Perl internals, it was quite difficult to understand why certain values were not evaluating correctly since Perl does a lot of “deep magic” on internal variables to make it “easy” for the end user.

Incorrect File Format

As mentioned before the file format was incorrect and incomplete. For example, it was not mentioned that each scan line in an 8-bit Zlib compressed image, must be padded out to be aligned to a 32-bit value, (Hence up to three bytes padding need to added at the end of each scan line). The width recorded in the tag, however, remains as the actual image width.

File Format complexity

The file format was far more complex than first thought. The specifications released only hinted at some of the more complicated optimisations. This made it very difficult to judge the amount of time needed to complete the project and to plan for any sort of formal structure early on.

As mentioned before it was discovered that text was stored in a complicated but ingenious way, as pointers to lists of shapes. The shapes define individual characters, or glyphs, for a font. It was defined in this manner to allow for the possibility of a font required by the movie not being available on the client system. The downside of this is that text replacement becomes difficult because if a letter required in the new text is not present in the old text then the glyph will not be present in the DefineFont tags.

There are two ways of circumventing this problem:

Firstly you could have the entire font set for each font defined in the SWF movie available on the server. Whenever some character that has not been defined before is needed that character is read from the font definition and converted into a shape and inserted into the font table of the file. This may however be quite slow.

Secondly the entire font set could be stored in the SWF file whether the character is needed or not. This is the technique Macromedia used when creating SWT files for their Generator product. This has the disadvantage of bloating the file size.

Embedded formats

On top of understanding the SWF file format there were a number of embedded file formats within SWF files that also needed decoding and extracting. As mentioned before these were JPEG, PNG, ADPCM and MP3. However, due to the constraints of the SWF file these are sometimes altered internally from their published specifications (as in the ZLIB image mentioned above) but these changes were not necessarily documented. This made it difficult to ensure that these formats were being parsed correctly.

What has been learned?

The project was possibly a little over ambitious

The complexity of the SWF file was vastly underestimated and the length if time it would take to produce a working decompiler and object structure was badly miscalculated. As a reference it was expected that that section of the project would be completed by Christmas last year however, due to group project, coursework, and later, exam commitments that stage wasn't actually finished until June.

Whilst part of this can be blamed on the incorrect specifications over enthusiasm also played a part.

Perl is possibly too slow

Whilst it was appreciated from the start that Perl, as an interpreted language, would be slower than compiled C it was not apparent quite how much slower it would actually be.

The slow down can be attributed to the fact that the reference code does not need to create an object structure, (instead printing out what its findings immediately) this does not explain why the equivalent Perl code can be 50 times slower.

It would be interesting to find out whether attempting to run this under mod_perl, which would remove the time that the Perl interpreter takes to launch and parse the script and would implement some caching of the code, would make the process any faster.

Similarly it would be nice to analyse the execution of the script using the standard Perl debugging tools to see where any bottlenecks are. These could then be optimised to see if a speed increase could be achieved.

The basic premise is sound

Most importantly it has been determined that the project is definitely feasible even if it was not quite completed here due to time constraints.

Extensions

Text Replacement

Due to the unforeseen complications in the representation of text in an SWF file it was not possible to implement text replacement. However the framework to do so is already in place and work on this matter continues as this document is being finalised and it is expected that this feature will be implemented shortly.

In order to implement text substitution auxiliary methods in the DefineText and DefineText2 tags would have to be implemented. These would provide the actual value of the text defined in the tag by analysing pointers to the glyphs tables defined in the DefineFont family of tags. If passed some new text they would then update these pointers. Additionally the relevant DefineFont would be have updated by adding new glyphs to the glyph tables. This would be necessary if the correct glyph for a particular character in a particular font was not already present.

The easiest way to perform the substitution would be to generate a hash table of variable values indexed by variable name. When a writer came to a tag that had some sort of substitutable text in. Examples of these are the obvious text definition tags, frame labels or an action command such as ‘jump to frame label x’ or ‘load URL y’. It would then attempt to substitute the text using a RegExp (Regular Expression) such as:

```
$value =~  s
{
       (\\*)(\${(.+)})
}{
       "\\" x (length($1) / 2) . (!(length($1) % 2) # this bit allows
                                                    # variable names to be
                                                    # escaped using ‘\’(and ‘\’s
                                                    # can be escaped themselves)

       && ($return = get_value($3)) ? $return : $2) # if only if the get_value
                                                     # function gets a replacement
}xeg;
```

so that if

```
       get_value(‘foo’) == ‘bar’
```

then

```
       $value = “hello my word is ${foo}”
```

becomes

```
       $value = “hello my word is bar”
```

but

```
       $value = “hello my word is \${foo}”
```

becomes

```
       $value = “hello my word is ${foo}”
```

and

```
       $value = “hello my word is \\${foo}”
```

becomes

```
       $value = “hello my word is \bar”
```

Write more input and output filters

Now that the object structure is in place writing input and output filters should be relatively easy, especially if the current filters are used as references.

In particular XML parsing and generating should be fairly easy since there are already a number of Perl modules for manipulating XML Document Objects.

It would be useful to write an output filter that extracted all the images and sounds of an SWF file.

Obviously, writing the glue between the generation library and the object structure would be a huge leap forward.

Write more subclasses

It would be useful to write some subclasses of the Flash::Object to show what can be done with it.

It would also be interesting to write a quick application using a Perl accessible graphical toolkit, such as GTK or QT that would allow you to draw shapes on screen and then save them as an SWF file.

Furthermore subclasses could be implemented that replicated the functionality of some of the other similar products mentioned before, in particular ones that provided Swish like text effects or could read generator configuration files and produce the same results.

Write utility methods

At the moment the tag and data type objects can only be altered by changing the individual parameters one by one. Whilst this works and is very flexible it is not the most convenient programmer interface.

More methods along the lines of the ‘rgb_hex’ function in the Colour module mentioned earlier would make performing some tasks a lot easier.

Work in sanity checking

At the moment the library assumes that the programmer is going to be well mannered and not input illegal values.

There are three types of illegal values :

Values can be out of range, for example depths cannot be less than 1 or larger than 2^{16} (65536).

Values can refer to non-existent objects. For example text cannot be defined using fontid 15 if font 15 has not been defined and PlaceObject cannot place objectid 15 if it has not been created.

Values can contradict each other. For example a ShapeRecord cannot be of type vline and hline. Also the extensive use of flags is rich ground for contradictory values. For example:

```
$record->fontid   ( $bin->get_word() ) if ($record->flags & text_hasfont);
$record->colour   ( get_colour       ) if ($record->flags & text_hascolour);
$record->x_offset ( $bin->get_word() ) if ($record->flags & text_hasxoffset);
$record->y_offset ( $bin->get_word() ) if ($record->flags & text_hasyoffset);
$record->height   ( $bin->get_word() ) if ($record->flags & text_hasfont);
```

At the moment `$record->flags` is kept along with `$record->fontid`. Theoretically, however, a user could set `$record->fontid` independently and not update `$record->flags`. It would be better that when reading the value of `$record->flags` the value is calculated by checking to see whether `fontid, colour, x_offset, y_offset` and `height` are actually defined.

Rewrite parser in C and wrap it

As mentioned before, the decompilation process is quite slow. Whilst this can be offset by decompiling to an easier to parse format offline first, and then using this easier format in any real time applications, it would still be beneficial to have a faster parser. As such it might be possible to write the parser in C and wrap it in the same way that libswf was wrapped.

This however could be quite difficult since there would be a need to pass around complex data structures or many individual values and convert the types between Perl and C.

Some experimentation would be needed to see whether the complexity of such a wrapping would be worth the speed increases.

Rewrite the SWF Generator

The fact that as the base of the SWF generator, libswf is not ideal has been detailed extensively elsewhere in this report. There are three main options to remedy this problem.

1) Rewrite the generator in Perl. This means that there would be no cross platform problems, no difficulties in wrapping C libraries and no copyright problems. However the possibility exists that this generator would be too slow for practical use.

2) Patch the more powerful Macromedia/Middlesoft Flash SDKs and write a Makefile to get them to compile under a variant of Unix such as Solaris, BSD or Linux. Then wrap them in a similar manner to libswf. Whilst this would be relatively speedy (since little code would have to be written) and the generation library itself would be fast there could be problems with not having control over the code base and copyright issues.

3) Write a new C library from scratch, possibly using code from the SDK as a starting point, then wrap it in XS. This would probably take longer but would mean that complete control was kept over the code and would circumvent any copyright problems.

Conclusion

The potential scope for this project, should it be fully realised, is huge. The ability to combine current Flash authoring techniques and then be able to manipulate them at every level, dynamically and programmatically, is very exciting especially in today's climate of high visual impact, database driven, dynamic web sites and cross browser incompatibilities.

There has already been some interest from industry to see this developed further and several companies have been in contact about continuation of the work.

In addition members of the Open Source community have expressed a desire expand upon this project and I would be very interested in taking it further.

Bibliography

Relevant Books

Programming Perl
August 1996, O'Reilly UK; ISBN: 1565921496
Larry Wall, et al.

Perl Cookbook
August 1998, O'Reilly UK; ISBN: 1565922433
Tom Christiansen, et al.

Mastering Regular expressions
January 1997, O'Reilly UK; ISBN: 1565922573
Jeffrey E. Friedl

Professional XML
February 2000, Wrox Press; ISBN: 1861003110
Mark Birbeck, et al.

Director 7 and Lingo Authorized
August 1997, Macromedia Press; ISBN: 0201354160
by Phil Gross, et al.

Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP
Addison Wesley Longman Publishing Co; ISBN: 0201604434
by John Miano.

SWF information

Macromedia
http://www.macromedia.com

The Open SWF organisation
http://openswf.org/

SWF file format specs (revised)
http://homepages.tig.com.au/~dkl/swf/SWFfilereference.html

Miscellanous Information

Perl
http://www.perl.com

JPEG
http://www.jpeg.org/

What is MP3?
http://www-scf.usc.edu/~huan/mp3.html

MPEG Audio headers guide - A description of MPEG audio frame headers.
Predrag Supurovic
http://www.dv.co.yu/mpgscript/mpeghdr.htm

Mpeg Audio FAQ
http://www.tnt.uni-hannover.de/project/mpeg/audio/faq/

Zlib homepage
http://artpacks.acid.org/pub/infozip/zlib/

PNG homepage
http://artpacks.acid.org/pub/png/

Freetype Open source True Type font library
http://www.freetype.org/

Similar Projects

Java XML/XSL
http://www.anotherbigidea.com/

Phillip Jacob
http://www.whirlycott.com/services/projects.html

Michael Wallace
http://www.sabren.com/code/flashperl/

Paul Haeberli - Libswf
http://reality.sgi.com/grafica/flash/

Olivier Debon - Netscape Flash Plugin and Swift tools
http://www.swift-tools.com/

The Labs
http://www.the-labs.com/

David Cantrell
http://www.cantrell.org.uk/david/tech/bits/flashperl/

Turbine
http://www.blue-pac.com/

Middlesoft
http://www.middlesoft.com

Swish
http://www.swishzone.com/

VisWeb
http://saxgate.saxess.com/visweb/

Form2Flash
http://www.kessels.com/Form2Flash/index.html

Swifty Utils
http://buraks.com/swifty/

Ming / PHP
http://www.opaque.net/ming/