A study on Migrating Flash files to HTML5/JavaScript

by

YOGESH Maheshwari, Y.Raghu Babu Reddy

in

Innovations in Software Engineering Conference, ISEC

Report No: IIIT/TR/2017/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
February 2017

A study on Migrating Flash files to HTML5/JavaScript

Yogesh Maheshwari
Software Engineering Research Center
IIIT Hyderabad, India
yogesh.maheshwari@research.iiit.ac.in

Y. Raghu Reddy
Software Engineering Research Center
IIIT Hyderabad, India
raghu.reddy@iiit.ac.in

ABSTRACT

Over the past few years, there is a significant shift from web to mobile devices. In addition, the success of HTML5 has negatively impacted the usage and support for Adobe Flash. This has resulted in unmaintained Flash assets and code bases. In this paper, we discuss the feasibility of a semi-automated technique to transform Flash based animations to HTML5 and JavaScript against re-writing the same animations from scratch. Writing animations from scratch in JavaScript is a time taking effort due to adherence to aesthetic details of the animation, domain knowledge and enumeration of all states. Our approach addresses these challenges by providing techniques to transform the Small Web Format (SWF) files to JavaScript. We validate our approach by conducting an experimental study to measure the efforts for a set of animations with respect to transformation from scratch, and transformation using our proposed approach. The results showed that our approach has the potential to significantly bridge the process of Flash to HTML5 migration.

Keywords

Flash, SWF, Program Transformation, Decompilation, Transpilation, ActionScript, JavaScript, HTML

1. INTRODUCTION

For years, Adobe Flash had been competing with Open Web and winning. The free (but not open source) Flash plugin was a *de facto* standard installation on nearly a billion desktop browsers. However the popularity of this platform for creating banner advertisments, rich internet application and browser games is on the decline. The primary reason for this decline is the lack of Flash support on devices running Google Android, iOS, etc. Over the past few years, some tools have been developed to address the absence of Flash player plugins. Flash players written in JavaScript like Mozilla Shumway[1], Smokescreen[2] and Gordon[3] were developed. However none of them could significantly support ActionScript (an object oriented programming language specifically designed for animations) to make this effort viable. Then came Google Swiffy[4], a Flash to HTML5 converter. Swiffy converted the Flash bytecode to JSON format, which was then executed via a JavaScript runtime [5]. It was not possible to modify the JSON content. However this converter was only targeted for banner advertisements and has now been shut down. Adobe itself has abandoned Flash and has started promoting HTML5 based tools. This leaves us with a large number of Flash applications and assets which will become completely obsolete in the years to come.

The migration of all Flash based applications that are still economically viable to HTML5 is a big challenge. Since SWF files require Flash plugin to run, one cannot write wrappers around it to make it run. The only alternative for a developer is to rewrite the entire application from scratch in HTML5/JavaScript. If FLA files (Flash source files) for given SWF Files are not present, this becomes a time taking effort, because:

- Recreation of Flash assets like images, vector graphics and animations while adhering to all the aesthetic details is a resource intensive effort.
- Developers may often lack the domain knowledge required for the particular animation. Therefore they will have to perform the additional step of enumerating all the animation states, before rewriting the entire logic in JavaScript which in itself is a huge task.
- Many HTML5 standards are still not completely implemented everywhere. Therefore one has to ensure browser compatibility.

Migrating the Flash files to Javascript requires a tool or set of tools to ease the process. In our previous work [4], we proposed a transformation process for converting Flash programs to HTML5/JavaScript. In this paper, we study the migration from flash programs to HTML5/JavaScript using our transformation approach [4] and compare it against rewriting from scratch in terms of effort for migration and further maintenance. By performing the study, we answer the following questions: (1) What kind of animations are better migrated using our approach than rewriting from scratch and vice versa?, and (2) How much effort is needed

ISEC '17, February 05-07, 2017, Jaipur, India

DOI: http://dx.doi.org/10.1145/3021460.3021472

[1] https://github.com/mozilla/shumway
[2] https://github.com/cesmoak/smokescreen
[3] https://github.com/tobeytailor/gordon
[4] https://developers.google.com/swiffy

Table 1: Reverse Engineering tools for Flash

Tool	Availability	Extent of Support for Asset Export	Extent of Decompilation
JPEXS Decompiler	Open Source	Supports multiple export formats for every component	Provides AS3 code
SoThink Decompiler	Proprietary	Supports multiple export formats for every component	Provides AS3 code
Zoe	Free	Only Frames can be exported	Class Level Structure
JSwiff Investigator	Open Source	NO Support	Only Tag Level Information

in maintaining and enhancing the transformed animations as compared to animations rewritten from scratch?

The rest of the paper is structured as follows: Section 2 provides an overview of the SWF file structure and our migration process published preiviously [4]. Section 3 details the comparative study vis-a-vis effort needed for migration of animations by our process versus rewriting from scratch. Section compares effort needed in maintenance for both the approaches. Finally, Section 5 discusses the threats to validity and scope of future work.

2. BACKGROUND

2.1 Flash Files and its structure

Flash files are distributed in .swf format (binary format). Adobe Flash Professional is the commonly used proprietary tool to generate SWF files. The .fla format is used for saving the source flash files by Adobe Flash Professional. The structure of SWF file is described in the SWF file Format Specification [2].

A SWF file consists of header, followed by a number of tags. The header contains information on the content of the file, like number of frames, frame rate, whether tags are compressed or not, etc. The Tags are of two types: definition and control tags. Definition tags define the objects in SWF file like shapes, text, sounds, etc. Control tags render and manipulate instances inside the character dictionary and control the flow of the file.

ActionScript is the scripting language for Flash. ActionScript3 is a dialect of ECMAScript. ECMAScript is specified in ECMA-262 [1], a standard derived from an early version of JavaScript. Current versions of Flash Player embed two virtual machines as the virtual machine for ActionScript 3.0 (AS3) is incompatible with ActionScript 2.0.

2.2 Related Tools

Decompiler: Decompilation is an instance of reverse engineering where an object program is translated into higher level program. There are several reverse engineering tools available for SWF file. Table 1 is a table of most relevant tools and their features. We choose JPEXS Decompiler [5] because it supports export of all SWF assets and AS3 code.

Transpiler: Transpiler convert programs from one language another language.There are three open source transpilers that convert ActionScript3 to JavaScript: Jangaroo[6], FalconJx[7] and FlexJs[8]. After studying all the three tools, we chose Jangaroo for our transformation process as it supports more AS3 libraries.

2.3 Our Transformation Process

We propose a three step transformation process for migrating the .swf files to Javascript/HTML5. Firstly, the input SWF file is decompiled to extract assets and AS3 code. In the next step we construct an AS3 project from the decompiled content. The last step involves transpiling this project into JavaScript [4]. The entire process is summarized in figure 1.

2.3.1 Decompilation

Here the SWF file is taken as input by the decompiler to output the following content: 1) Sprites, Images, Buttons in PNG format. 2) Transformation matrices for each *named asset*. A *Named asset* is an asset created in Flash Professional tool which is referenced in AS3 code. 3) *DisplayList* of each named asset in terms of smaller asset. *DisplayList* contains display objects and precedence of how they are displayed in animations. 4) AS3 (ActionScript3) code after decompilation. 5) Each frame of MovieClip is exported as a PNG image and then converted into a spriteSheet.

2.3.2 Project Construction

This stage consists of 4 main steps. 1) Creation of wrapper AS3 class for each named asset. A script creates the class body by declaring each item in the DisplayList and we then manually embed the corresponding assets in this class using the Embed meta Tag defined in AS3. Each class is then invoked in the *Main* class of the project. 2) A new function called setStage is added in the Main Class where these assets are added on the displayList, with the Transformation Matrices defining their locations. 3) Lastly all the functions which are not implemented in transpilers Flash API are removed, or replaced as appropriate. 4) Implementing AS3 wrappers of any required JS library.

2.3.3 Transpilation

Here the final AS3 project obtained from the construction phase is transpiled to JavaScript. This is the final step in transformation process after which we have a JavaScript code for the input SWF file which is callable via HTML5. For any future evolution of the program changes can be made either in AS3 project which can be transpiled to JS, or external JS wrapper can be written around this conversion. We however feel that it is more convenient to make changes in AS3 project and have AS3 wrappers for any external JS library required.

3. EVALUATING MIGRATION EFFORT

3.1 Data Selection

To evaluate the migration effort, we first select a set of Flash animations that need to be migrated to HTML5/JavaScript format. We compare the effort involved using our transformation process with the effort needed in rewriting the same set of animations from scratch. We picked six flash based experiments from Virtual Labs repository[9], a government

[5]https://www.free-decompiler.com
[6]http://www.jangaroo.net
[7]https://cwiki.apache.org/confluence/display/FLEX/FalconJx+Prototype
[8]https://cwiki.apache.org/confluence/display/FLEX/FlexJS
[9]https://vlabs.ac.in

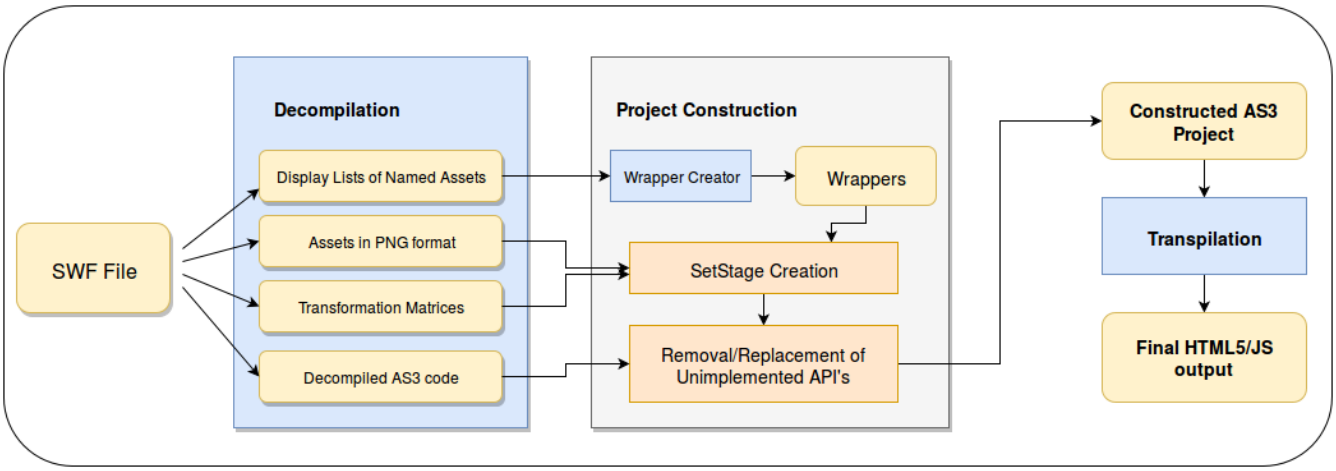

Figure 1: Transformation Process

of India initiative to supplement undergraduate engineering curriculum with labs. The first 4 experiments resemble rich internet appications (RIA) whereas last two, although RIA, resonate more with banner ads as they are largely MovieClips. As a reference point we record the following metrics for each animation (shown in Table 2):

- **Number of Classes**: The number of AS3 classes found in the decompiled code. Higher number implies greater complexity of animations.
- **Line of Code(LOC)**: Higher number in decompiled code implies greater animation complexity.
- **Named Assets** : More *named assets* indicates higher interaction of AS3 with the assets and correlates to greater animation complexity. We further categorize these into - ***Simple Assets* (SA)**, ***Complex Assets* (CA)** and ***Partially Supported Assets* (PSA)**. *Simple Assets* consists of UI components like buttons and texts, which can be easily created in HTML5. From a strict functionality perspective aesthetics of these assets generally do not matter. *Complex Assets* are shapes, images and sprites which are difficult to recreate. Here aesthetics are important. *Partially Supported Assets* are MovieClips and MorphShapes. A lot of custom code needs to be written to accomodate such assets via our process. However they have good JavaScript API's.
- **Function Points** : Our working definition of Function Points is based on ISO/IEC 19761:2003 [3]. Function points are evaluated after enlisting the requirements from examining each animation. The Value Adjustment Factor (VAF) for these animations is set at 0.8 after examining the 14 General System Characteristics. We have rounded the values to nearest integer.

3.2 Experiment Design

The experiment was performed by a single subject. In order to remove familiarity bias and prevent skewing of results, we transformed Exp1 to Exp3 using our process, then Exp4 to Exp6 were rewritten from scratch. This was followed by rewriting Exp1 to Exp3 from scratch. In the end Exp4 to Exp6 were transformed using our process.

3.2.1 *Transformation Process*

Table 2: Set of Selected Animations

Id	Name	Number of Classes	LOC	Named Assets			Function Points
				SA	CA	PSA	
Exp1	Kruskal	13	2157	0	0	0	50
Exp2	DFS_BFS	11	2855	0	0	0	54
Exp3	ExpressionTree	29	4545	25	13	0	128
Exp4	Binary Search Tree	6	914	3	3	0	24
Exp5	Tension	2	56	1	0	1	12
Exp6	Orifice	2	61	1	0	1	16

Decompilation and Transpilation are completely automated processes and takes negligible time (in mins) and hence it is not recorded. The bulk of time is spent in the project construction phase which can be divided into three steps - Wrapper creation, SetStage Function Creation and modifying AS3 code. Hence following pareameters are recorded:

- **Wrapper Creation Time**: This is the total time taken to create wrappers for all the *named assets*. The manual task here is to embed the corresponding assets in wrapper code.
- **setStage Function Time**: This is the time taken to populate the stage with all the frame assets. It also involves assigning transformation matrices to every asset.
- **Modification Time**: This is the total time required to remove unimplemented Flash API calls and replacing them with alternatives available.
- **Total time taken**: Total time taken to get the final HTML5/JS as output.
- **Total LOC**: These are the total LOC (lines of code) in constructed AS3 project.
- **LOC changed**: It is sum total of lines of code which are added, removed or replaced.

3.2.2 *Rewriting from Scratch*

The following parameters were recorded while writing from scratch:

- **Time Taken for State Enumeration**: This is the total time needed to explore the animation and list down all it's distinct states, assets that will be required and their interactions with the user.

Table 3: Migration via Transformation Process

Id	Wrapper creation Time	SetStage Function Time	Time taken to remove/ Replace AS3 code	Total time Taken	Total LOC	LOC changed
Exp1	0	20 mins	1 hr 10 mins	1 hr 30 mins	2224	122
Exp2	0	5 mins	2 hr	2 hr 5 mins	2959	133
Exp3	3 hrs	1 hr	3 hrs	7 hr	5279	800
Exp4	1 hr	2 hrs	1.5 hrs	4.5 hrs	985	81
Exp5	30 mins	1 hr	4 hrs	5 hrs 30 mins	266	234
Exp6	30 mins	1 hr	1 hrs	2 hrs 30 mins	289	247

- **Asset Creation time**: It involves time spent in recreating all the assets like images, buttons, etc.
- **Time for HTML5/JS code**: This includes the total time for writing the JS logic for animation and HTML5/CSS code to place the assets and to create the main containers where the animation resides.
- **Total Time**: The total time required in the process.
- **Total LOC**: Total lines of code written.

3.3 Results and Observations

Table 3 and Table 4 show the measurements from the experiment. From the results, we can see that our process outperforms rewriting from scratch in terms of time taken. While rewriting from scratch, bulk of the time is spent in mapping HTML5/JS features to state requirements and trying to maintain the same look and feel of animations. It was also difficult to find the related API for getting similar effects. Time was also spent in getting acquainted with new JavaScript API's for every animation.

The time taken by our method increases as the number of named assets increases. This is because every named asset requires a wrapper, has to placed in the stage and needs to be checked for API feature instances that may not be supported. In case of Complex Assets, the time required to create a wrapper is significantly lesser than time take to rewrite the asset. However for Simple Assets like button and texts, time taken to create wrapper is almost similar to time taken for declaring these in HTML5. In such scenarios, if lines of code in decompiled script is very less, it is better to rewrite from scratch.

Time for Exp5 is larger than expected because it required implementation of a small subset of MovieClip API. The same API has been reused in the next animation (Exp6). However for animations that have multiple such assets relying completely on MovieClip, we have to ensure other conditions like synchronized motions of all assets with each other and other time dependent entities like tweens. This requires implementation of more of the API features and is not viable for a developer.

For Exp5 and Exp6 we find that our transformation process requires time in reimplementing subset of movieclip API, whereas bulk of time during rewriting goes in recreation of assets. This indicates that for such animations a better alternative could be to partly use our approach to extract assets and then recode the logic in JavaScript which contains API's similar to MovieClip to animate them, thereby taking lesser time. For further discussion, we will call this a approach a **Hybrid Approach**. Table 6 summarizes viable approaches in each scenario. In case of high LOC, we recommend using our approach. However when this is not the case and animations have more SA (simple assets), it is better to rewrite from scratch as these are easily reproducible in HTML5. Another reason is that our approach involves creating wrappers for each of these assets that only increases LOC and reduces code readability. When more PSA are present in animations it is best to use the hybrid approach.

The study also indicates that a great chunk of time is spent in recreating the assets from scratch. However when similar animations need migration, these assets can be reused, thereby saving that time. In case of similar animations there also exists a lot of boilerplate code like classes for defining animation themes and templates, which needn't be coded again. This can result in reduction of time taken for rewriting from scratch.

4. EVALUATION OF MAINTENANCE

4.1 Experiment Design

To evaluate maintainability, we add 2 features to Exp1 - Exp4 in our transformed code and rewritten code. The last two experiments are not included because: a) they are very simple in terms of requirements b) they are largely movieClips or set of images and contain minimal AS3 code. As in previous experiment familiarity bias is removed by first adding features to Exp1 and Exp2 of our transformed code, then Exp3 and Exp4 of rewritten code, then Exp1 and Exp2 of rewritten code and lastly Exp3 and Exp4 of transformed code. For every feature, we record the Enhancement Function Points it consists of considering a VAF score of 0.8 as stated earlier. At the same time we measure the total time taken and Lines of Code changed.

4.2 Results

From table 5 we observe that the mean time taken for enhancements is more when they are performed on our Transformed code as compared to the code that was rewritten from scratch. The difference in time taken can be attributed to the following factors: (1) The subject is not acquainted with the transformed code logic, as he merely adds assets and removes unimplemented Flash API calls in previous experiment. This is not the case with rewritten JS code, (2) The subject may not be comfortable with existing code structure, and (3) Transformed code consists of local variables, whose names could not be retrieved in decompilation process thereby reducing readibility of the code.

The tabulation also helps us to calculate the average effort in terms of time taken for a unit function point. We observe that effort required to implement unit Function Point on our transformed code is **30 min** whereas it is **24 min** on a code rewritten from scratch.

Table 4: Rewriting From Scratch

Id	Time Taken For State Enumeration	Asset Creation Time	Time For HTML/JS Coding	Total Time	Total LOC
Exp1	4 hrs	0	35 hrs	39 hrs	752
Exp2	3 hrs	0	30 hrs	33 hrs	1156
Exp3	1 hrs	8 hrs	42 hrs	51 hrs	863
Exp4	10 mins	4 hrs	25 hrs	29hrs 10 min	349
Exp5	15 mins	3 hrs	1 hr	4hrs 15min	67
Exp6	15 mins	2 hrs	1 hr	3 hr 15min	67

Table 5: Evaluation of Maintenance

			Transformation Process			Rewriting From Scratch		
Id	Feature to be added	Function Points	Time Taken	LOC Changed	Mean Time	Time Taken	LOC Changed	Mean Time
Exp1	Add Edge Weights	5	3 hrs	120	4 hrs	2 hrs	54	5.5 hrs
	Modify weights and restart simulation	11	5 hrs	220		9hrs	253	
Exp2	Populate Instructions Button	3	2 hrs	57	3 hrs	30 mins	12	1.75 hrs
	Modify graph and restart simulation	8	4 hrs	130		3 hrs	78	
Exp3	Add New operation button	6	3.5 hrs	48	4.25 hrs	2 hrs	44	3 hrs
	Accept Postfix expression as input	9	6 hrs	53		4 hrs	45	
Exp4	Add undo operation	8	4 hrs	81	5 hrs	2 hr	30	3.5 hrs
	Feature to insert a node in animated tree	18	6 hrs	109		5 hrs	68	

Table 6: Viablity of Methods

Animation consists of	Viable Approach
SA, CA, High LOC(>200 per class)	Transformation Process
High SA, Less CA(<5), Low LOC	Rewrite from Scratch
SA, CA, Less PSA(<5), High LOC	Transformation Process
SA, CA, More PSA, Low LOC	Hybrid Approach
More PSA, High LOC	Hybrid Approach/Transformation Process

5. THREATS TO VALIDITY AND FUTURE WORK

In our study, we try to compare the effort for enhancement in code bases between AS3 and JavaScript. Although both are same ECMA-262 standard, they are still different languages. Hence comparing lines of code related parameters may not be appropriate. We also realise that we can have other starting points for migrating our code, other than starting from scratch. As pointed in the first study, we can migrate the assets via the decompiler and rewrite the code logic in JS. We can also take hints for logic and structure from the decompiled code. With help of JS libraries which are similar to Flash for example CreateJS, one can also attempt to manually transpile the decompiled AS3 code to JS, replacing Flash API calls with CreateJS.

Although we have tried to mitigare familiarity bias, there still remains a bias in second experiment where user is familiar with rewritten JavaScript code from first experiment. We therefore do not account the time taken by the subject to comprehend the code bases for comparison. Other factors like subject familiarity with languages, chosen VAF factor pose some threats to the study.

Other threats to the study lie in the limitations of our approach. The limitations to our approach are: (1) The user is limited to Flash APIs presently implemented in the transpiler. Usage of other open source JavaScript API requires writing AS3 wrappers which is not always possible, especially for large API's. (2) Modifying transpiled JavaScript code is difficult. (3) Given that AS3 is not a popular choice, the output language may need to be different in future.

The work can extended by creating a transpiler to output JavaScript code that can be modified conveniently as compared to present final output. Given that the output languages will now be the same, it creates a scope of new comparative study with aim of determining the effort to maintain codebases obtained after both techniques. Other languages like Dart and TypeScript can be explored for migrating the decompiled AS3 code. Dart's StageXL library has API's similar to Flash, whereas one can use Shumway's Flash API's for TypeScript or even write their TypeScript wrappers for CreateJS API.

6. REFERENCES

[1] ECMA. Ecmascript® 2015 Language Specification, 6th edition, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. June 2015.

[2] A. S. Incorporated. Swf FILE FORMAT SPECIFICATION VERSION 19, http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf-file-format-spec.pdf. 2012.

[3] I. ISO. Iec 19761 software engineering-cosmic-ffp-a functional size measurement method. *International Organization for Standardization_ISO, Geneva*, 70, 2003.

[4] Y. Maheshwari and Y. R. Reddy. Transformation of flash files to html5 and javascript. In *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*, pages 23–27. ACM, 2015.

[5] P. Senster. The Design and Implementation of Google Swiffy: A Flash to HTML5 Converter. Master's thesis, TU Delft, Delft University of Technology, 2012.