# Transforming Flash files to HTML5 and JavaScript

Thesis submitted in partial fulfillment
of the requirements for the degree of

*Masters of Science*
*in*
*Computer Science and Engineering by Research*

by

Yogesh Maheshwari
201202014
yogesh.maheshwari@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
August 2022

Copyright © Yogesh Maheshwari, 2022
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

# CERTIFICATE

It is certified that the work contained in this thesis, titled "**Transforming Flash files to HTML5 and JavaScript**" by **Yogesh Maheshwari**, has been carried out under my supervision and is not submitted elsewhere for a degree.

______________ Date

______________________________ Adviser: Prof. Y. Raghu Reddy

# Acknowledgments

Firstly, I thank Almighty earnestly for sealing my proposals with His divine consent. The successful completion of this thesis is a proof of His grace along with the love and guidance of those named below.

I am grateful to my advisor **Dr. Y. Raghu Reddy** who motivated and guided me to tread the Byzantine path of research in the field of Program Transformations. Working under him is a great learning experience. He promoted free thought, exploration and has pushed me to think out of the box.

I owe my sincere gratitude to all the members of Software Engineering Research Center for their valuable guidance, assistance and help during the course of my study. I am grateful to all my batchmates who have directly or indirectly assisted throughout my thesis work.

I also express my special thanks to all my family members for motivating and inspiring me all throughout my work and showing their undaunting love, and pride in my work. Last but not least, I would like to thank everyone who has been instrumental in my efforts towards the successful culmination of this academic pursuit.

# Abstract

Flash is a multimedia software platform used to produce animations, desktop applications, mobile games and applications. Flash graphics and animations are viewed in a browser through use of Flash Player, AIR and some third-party players are used for viewing desktop and mobile apps. For years, this free (but not open source) Flash plugin was a de- facto standard installation on nearly a billion desktop browsers. However, 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. To top this in July 2017, the official statement from Adobe stated, ”In collaboration with several of our technology partners - including Apple, Facebook, Google, Microsoft and Mozilla, Adobe is planning to end life of Flash. Specifically, we will stop updating and distributing the Flash Player at the end of 2020 and encourage content creators to migrate any existing Flash content to these new open formats.” This has caused developers to migrated towards open source platforms like HTML5 thereby resulting in unmaintained Flash assets and code bases. In our work, we propose SWF2JS - a tool that enables semi-automated migration of SWF files to JavsScript. We also discuss its feasibility to transform Flash based animations to HTML5 and JavaScript against re-writing the same animations from scratch.

# Contents

Chapter Page

Abstract . . . v

1 Introduction . . . 1
  1.1 Adobe Flash . . . 1
  1.2 HTML5 and Javascript . . . 1
  1.3 Motivation . . . 2
  1.4 Objective . . . 2
  1.5 Scope . . . 2
  1.6 Major Contributions . . . 2
  1.7 Outline . . . 3

2 Background . . . 4
  2.1 Program Transformation . . . 4
    2.1.1 Extent of Automation . . . 4
    2.1.2 Nature of Source and Target languages . . . 4
  2.2 SWFs . . . 6
  2.3 ActionScript . . . 7
  2.4 Open Source Tools Related to Flash . . . 8
    2.4.1 Decompiler . . . 8
    2.4.2 Transpiler . . . 8
  2.5 Technologies for HTML5 . . . 10
    2.5.1 SVG: Scalable Vector Graphics . . . 10
    2.5.2 Canvas . . . 11
    2.5.3 WebGL . . . 11

3 Literature Review . . . 12
  3.1 Move towards HTML5 . . . 12
  3.2 Existing Solutions . . . 13
    3.2.1 Flash Players in JavaScript . . . 14
    3.2.2 Migration to HTML5 . . . 14
      3.2.2.1 Fully Automated: Swiffy - Flash to HTML5 converter . . . 14
      3.2.2.2 Semi Automated Migrations . . . 17
      3.2.2.3 Manual Migration . . . 19

4 Proposed Approach . . . 20
4.1 Overview . . . 20
4.2 Proposed Approach . . . 21
4.2.1 Decompilation . . . 21
4.2.2 Project Construction . . . 22
4.2.3 Transpilation . . . 22
4.3 Example Walkthrough . . . 23

5 Validations . . . 27
5.1 Evaluating Migration Effort . . . 27
5.1.1 Animation Selection . . . 27
5.1.2 Metrics to objectify complexity of an animation . . . 28
5.1.3 Experiment Setup . . . 32
5.1.3.1 Transformation Process . . . 32
5.1.3.2 Rewriting from Scratch . . . 33
5.1.4 Results and Inferences . . . 34
5.2 Evaluation of Maintenance . . . 35
5.2.1 Experiment Design . . . 35
5.2.2 Results . . . 35
5.3 Threats to Validity . . . 36

6 Conclusions . . . 37
6.1 Problems with proposed approach . . . 37
6.2 Future Work . . . 37
6.2.1 Improvements in Approach . . . 37
6.2.2 More Comparitive Studies . . . 38
6.2.3 Migration to other languages . . . 38

Bibliography . . . 39

# List of Figures


| Figure | | Page |
|---|---|---|
| 1.1 | Adobe Flash Logo | 1 |
| 2.1 | Program Transformation Taxonomy by Visier et.al. | 6 |
| 3.1 | Decline in Flash Usage | 13 |
| 3.2 | Mozzilla Shumway Architecture | 15 |
| 3.3 | Swiffy Architecture | 16 |
| 3.4 | Generated Swiffy Object | 17 |
| 3.5 | Migration using HAXE | 18 |
| 4.1 | Option1: Exporting JS code | 20 |
| 4.2 | Option 2: Exporting AS3 and then transpiling to JS | 21 |
| 4.3 | Transformation Process | 21 |
| 4.4 | Rectangle | 23 |
| 5.1 | Kruskal | 28 |
| 5.2 | DFS BFS | 29 |
| 5.3 | Expression Tree | 29 |
| 5.4 | BST | 30 |
| 5.5 | Tensions | 30 |
| 5.6 | Orifice | 31 |
| 5.7 | Experimentation Order | 32 |

# List of Tables


Table | Page
2.1 SWF example for placing a simple shape . . . 7
2.2 Reverse Engineering tools for Flash . . . 8
2.3 Asset Export Capabilities of JPEXS Decompiler . . . 9
2.4 AS3 Export Capabilities of JPEXS Decompiler . . . 9
2.5 SVG code to create a black rectangle . . . 11
2.6 JS code for creating a rectangle via canvas . . . 11

3.1 AS3 code for creating a circle . . . 16

4.1 Trace for a simple blue rectangle drawn via AS3 . . . 23
4.2 AS3 code used to create a rectangle . . . 24
4.3 Expected exported AS3 output for Shape . . . 25
4.4 Transpiled output to JavaScript for the rectangle shape . . . 26

5.1 Set of Selected Animations . . . 32
5.2 Migration via Transformation Process . . . 33
5.3 Rewriting From Scratch . . . 34
5.4 Evaluation of Maintenance . . . 35
5.5 Viablity of Methods . . . 35

*Chapter 1*

# Introduction

## 1.1 Adobe Flash

Adobe Flash is an authoring application which is used to create animations, Rich Internet Applications(RIA), games and advertisements. It supports various features such as bitmaps, multimedia content, vector graphics, timelines, layers and has its own scripting language called ActionScript because of which it has gained huge popularity. The artifacts developed using Adobe Flash requires Flash player to run them. Flash players are analogous to JVM that executes JAVA bytecodes. They serve as an interface between OS and the underlying Flash code and hence abstracts out platform specific nuances. To execute the same on web browsers, an Adobe Flash player plugin is required. This plugin interfaces between web browsers and the existing Flash code. The word *Adobe Flash* and *Flash* will be used interchangeably throughout the thesis.

## 1.2 HTML5 and Javascript

HTML5 is a markup language which is used to structure and present information on webpages. Javascript is a scripting language which is used to provide interactivity to webpages. It is a dialect of ECMASCRIPT [33]. Today, the combination of HTML5 and Javascript is good alternative to Adobe Flash in terms of displaying content over web due to good video support, canvas, and introduction of high level programming constructs like packages, classes in Javascript. However, this was not the case when Flash had gained popularity.

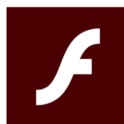

**Figure 1.1** Adobe Flash Logo

## 1.3 Motivation

For years, Adobe Flash has been competing with open source tools available for web development and has gaining enormous popularity. The free (but not open source) Flash plugin was a *de facto* standard installation on nearly a billion desktop browsers. However the demand of this platform started declining for it various use cases like creating banner advertisments, rich internet application and browser games. The primary reason for this decline is the lack of Flash support on devices running Google Android, iOS, etc. In April, 2010, Steve Jobs actually predicted this would happen [49]. He did not think it was a good idea to move forward with Adobe Flash because he believed that Flash Player was created for the PC era, but when it comes to mobile devices, it fell short due to the following reasons which are Openness, reliability, security, performance and substandardized quality of 3rd party development tools. Adobe made an announcement in 2017 that by 2020, they will no longer support the flash player plug-in. Thereafter, Google and Microsoft have announced that they will disable the plug-in by default in their browsers by early 2020[50]. Hence, browser support for Flash plugin no longer exists.

This limited usage and eventual shutdown of Adobe Flash in the mobile era due to the problems associated with it which has severely impacted large repositories of Flash content in form of games, e-learning courses, etc. All this raised concern for supporting the existing Flash content on browsers and mobile devices.

## 1.4 Objective

In the presence of source flash files migration to HTML5 is relative easier because we may not need to recreate the assets, however this is not always the case. Hence, the objective of the thesis is to aid in conversion of Flash files to HTML5 and JavaScript when no source files are present while ensuring that conversion is viable from the perspective of migration and maintainability.

## 1.5 Scope

The scope of this thesis is limited to Flash files where no source files are present. Since no source files are present the process of migration is bound to require human intervention because although flash files can be decompiled, the content is often obfuscated and it is difficult to establish relationships between code and assets. Not all Flash functionalities have a one to one mapping in HTML5 and hence custom libraries need to be written to support them. This thesis explores libraries which can support this migration, however implementing the Flash libraries in HTML5 is beyond the scope of this thesis.

## 1.6 Major Contributions

The contributions of this thesis are as follows:

- It presents an exhaustive survey on the following topics:
  - Tools to decompile SWF files: There exists lots of SWF decompilers, however not all are open sourced and also they support limited functionality.
  - Transpilers for AS3 code: We provide a survey of tools available to transpile AS3 code, along with output languages and support functionalities.
  - Survey of existing approaches and tools that aid migration of content to HTML5 and the problems associated with them.
- The novelty of this work is a semi automated approach to migrate SWF files to HTML5/JS.
- Methodology for evaluating the proposed approach against manually converting the content from scratch with respect to time taken and future maintenance.

## 1.7 Outline

The structure of the thesis is as follows. Chapter 2 explores Program Transformations, SWF file formats, Actionscript, and multiple tools for decompiling flash files and transpiling actionscript code. Chapter 3 is a survey of tools/solutions/work that aids in migrating Flash files to HTML5 and JS. Chapter 4 discusses why a fully automated transformation is not feasible and proposes a semi automated approach to do the same. Chapter 5, validates this approach in terms of time taken with respect to converting from scratch and also assess the maintainability of the transformations. Chapter 6 discusses the shortcomings of this approach and the future work in this area.

*Chapter 2*

# Background

## 2.1 Program Transformation

Program Transformation is the process of transforming one program into another without changing the underlying semantics of the program [75]. Researchers have worked on program transformations for several years in the past three decades. Program transformations have been categorized in many different ways depending upon the objective of the study. Stuikys et.al [70] presents an operation view on the same, where they categorize program transformations on the types of operations which can be performed to achieve it.

This thesis explores program transformations in two ways namely, Extent of Automation and Nature of Source and Target languages which are discussed in details in subsequent subsections.

### 2.1.1 Extent of Automation

Partsch [59] classifies program transformation into following categories:

- **Manual transformation** : The user is responsible or performs each and every transformation manually.
- **Semi Automatic transformation** : Large chunks of transformations can be done automatically, however user intervention is required for replacing libraries, de-obfuscation, restructuring, etc. The proposed approach which will be described in later chapter belongs to this category.
- **Fully automatic transformation** : End to End transformation can be done in an automated manner with no or very minimal inputs by user( eg.initial configuration).

### 2.1.2 Nature of Source and Target languages

The language of the program being transformed is the source language and the language of the transformed program is the target language. The source and target languages may be same or different from each other. Based on source and target languages Visier et. al. [74] classifies program transformation as depicted in fig 2.1. When the source language and the target language are same, the transformation

is called $Rephrasing$. When source and target languages are different, the transformation is called $Translation$. Program transformations tend to be algorithmic in nature, however in our work, we are proposing a series of tasks to transform the source program to the target program.

Rephrasing can be further subdivided into following categories:

- **Normalization:** This type of transformation reduces syntactic complexities of a program.
- **Optimizations:** This type of transformation improves space or time complexity of a program. It could involve function inlining, dead code removal, etc.
- **Refactoring:** This type of transformation involves restructuring the program in a way that is easy to understand while preserving the semantic complexity. This could include changing variable names, function names, following standard design patterns.
- **Obfuscation:** This type of transformation embraces upon making a program harder to understand to protect business logic. This includes renaming variables/funcitons, adding dead code, etc.
- **Rennovations:** This type of transformation involves changing behavioural aspects of a program like fixing an error or confirming to updated requirements.

Program translation is not a new idea and is popular since decades [30],[26],[76]. Based on the extent of semantic preservation and the levels of abstraction of the source and target program, program translation can further be categorized as:

- **Program Synthesis:** It reduces the abstraction of code and is of two types.
  - First is *refinement*, which is the process of implementation of code from its architecture.
  - Other is *compilation*, it is the process of conversion of high level language code into machine code or executable.
- **Program Migration:** The translation happens at the same level of abstraction. Transpilers from Java to JS [52], [51], WEB-IR proposed by Tomokazu, Shinya, Shota and Teruo [42] are some more recent examples of program migration. Our work aims to transpile ActionScript3 to JavaScript.
- **Reverse Engineering:** Reverse Engineering raises the level of abstraction. Reverse Engineering can be used to extract a high level program or specification or visualize the abstractions from a low level program. Reverse Engineering can be done at various levels: from byte code to language code, language code to models, from programs to architectures, etc. It is further classified into four types:
  - *Decompilation :* Decompilation is an instance of reverse engineering where an object program is translated to high level program. In our transformation process, we decompile the .swf files in order to extract ActionScript3 assets.

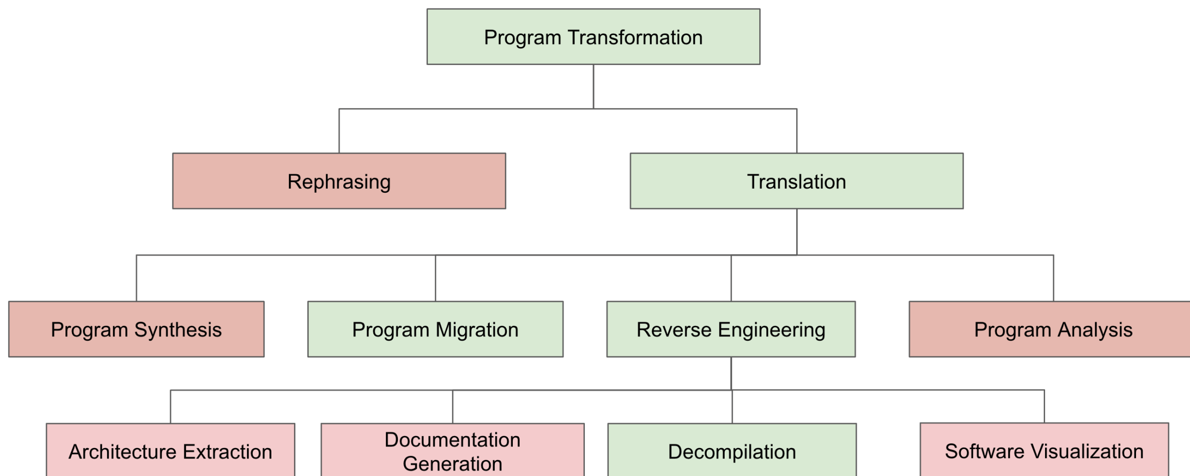


**Figure 2.1** Program Transformation Taxonomy by Visier et.al.

  - *Documentation Generation:* It produces documentation from program.
  - *Architecture extraction:* It extracts high level design from program.
  - *Software Visualizations:* It is used for depicting some aspect of program in an abstract way.

- **Program Analysis**: It reduces program to a particular aspect eg. control or data flow.

## 2.2 SWFs

The .swf or *swiff* file format is the binary format used to distribute Flash content on internet. Adobe Flash Professional is a developer tool which is commonly used to generate .swf files. This format requires a Flash player on desktop or flash player plugin on browsers to execute. Adobe Flash player and plugin are free and not open sourced. There are a few open sourced Flash players like Gnash [8] and LightSpark [12] but they support SWF files only partially.

As described in the SWF File Format Specification [46], a SWF file is made up 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. These objects are assigned character ids and are stored in a dictionary by Flash player. These tags by themselves do not cause anything to be rendered. An example of definition tag is $DefineShape$ tag used to define a shape. The aim of control tags is to render and manipulate instances inside the character dictionary and control the flow of the file. An example for control tag is $ShowFrame$ tag responsible for rendering of the frame. It also demarcates end of the current frame and beginning of new frame.

Let us take a simple SWF example as mentioned in table 2.1. We first see a Header that will have some predefined values like size, number of frames, framerate, etc. Next we see a DefineShape tag. It

| Tag Name | Value |
|---|---|
| Header | <some_predefined_values > |
| DefineShape | id:1, <vector_data > |
| PlaceObject | id:1, depth: 10, transform: (10,20) |
| ShowFrame | |
| End | |

**Table 2.1** SWF example for placing a simple shape

creates an object of id 1 which has some vector data which is a curve defining its shape. Till now the object is only defined but not placed on display. We then see a PlaceObject Tag which place this created object at depth 10 and positions it by move 10px right and 20px down. The depth 10 means this object will be placed below an object of depth 9 if that existed or above object of depth 11 if that existed. Even now no rendering is done. We now see a ShowFrame tag indicating that the Frame is to be rendered with an instance of object with id 1 at depth 10 at 10px,20px positions.

## 2.3 ActionScript

ActionScript[47] is the scripting language for Flash. It comes in three versions ActionScript 1.0, ActionScript 2.0 and ActionScript 3.0. ActionScript 2.0 can be compiled to ActionScript 1.0 by Adobe Flash Professional. Our transformation process is largely concerned with .swf files using ActionScript 3.0. It is mixture of function calls like ActionPlay and stack-based instructions like ActionPush and ActionPop. The bytecode for ActionScript 2.0 mainly resides in DoAction tag and DoInitAction tags. On the other hand ActionScript 3.0 resides in DoABC tag. DoABC consists of some optional headers and a list of traits. Traits can be functions, classes, namespaces, interfaces, getters, setters, constants or variables. Only one of these traits can be public, others are private.

ActionScript3 is a dialect of ECMAScript. ECMAScript is specified in ECMA-262 [32], a standard derived from an early version of JavaScript. Therefore ActionScript has a lot in common with JavaScript. Both languages feature prototype inheritance for instance. The major additions in ActionScript not present in JavaScript include dynamically and statically typed code, packages, classes, interfaces and type-safe conditional compilation. Usage of AS3 has declined substantially after decline of Flash and is no longer percieved as an introductory programming language for web developement [53].

Current versions of Flash Player embed two virtual machines as the virtual machine for ActionScript 3.0 is incompatible with ActionScript 2.0.

**Table 2.2** Reverse Engineering tools for Flash

| Tool | Availability | Extent of Support for Asset Export | Extent of Decompilation |
|---|---|---|---|
| JPEXS Decompiler[6] | Open Source | Supports multiple export formats for every component | Provides AS3 code |
| SoThink Decompiler [19] | Proprietary | Supports multiple export formats for every component | Provides AS3 code |
| Trillix Decompiler [20] | Proprietary | Supports multiple export formats for every component | Provides AS3 code |
| Zoe [23] | Free | Only Frames can be exported | Class Level Structure |
| JSwiff Investigator[11] | Open Source | NO Support | Only Tag Level Information |
| RABCDAsm[15] | Open Source | Allows Viewing and Editing AS3 tags | Only Tag Level Information for AS3 |

## 2.4 Open Source Tools Related to Flash

### 2.4.1 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 2.2 is a table of most relevant tools and their features. We choose JPEXS Decompiler [6] because it supports export of all SWF assets 2.3 and AS3 code 2.4.

### 2.4.2 Transpiler

Transpiler converts programs from one language to another language.There are three open source transpilers that convert ActionScript3 to JavaScript: Jangaroo[10] [40] [41], FalconJx[5] and FlexJs[7]. After studying all the three tools, we chose Jangaroo for our transformation process as it supports more AS3 libraries.

**More About Jangaroo**

This tool consists of two parts:

- **Language**:
  Jangaroo is a subset of the ActionScript 3 language. It supports the following concepts of ActionScript that go beyond ordinary JavaScript:
  - classes including private and static members
  - packages
  - interfaces

**Table 2.3** Asset Export Capabilities of JPEXS Decompiler

| Asset | Tag | Export Format Supported |
|---|---|---|
| Whole SWF | | FLA, XFL, and internal XML format |
| Shapes | DefineShape1-4 tags | SVG (limited), PNG, HTML 5 Canvas, BMP, SWF |
| MorphShapes | DefineMorphShape1-2 tags | SVG (limited), HTML 5 Canvas, SWF |
| Sprites | DefineSprite tag | PNGs, BMPs, GIF, AVI, SVG(limited), HTML 5 Canvas, PDF, SWF |
| Buttons | DefinButton1-2 | PNG, BMP, SVG, SWF |
| Texts | DefineText1-2, DefineEditText tags, No TLF support | Plain text, FFDec formatted text, SVG |
| Images | DefineBits, JPEGTables, DefineBitsJPEG2-4, DefineBitsLossless1-2 | JPEG, PNG, BMP |
| Fonts | DefineFont1-3, DefineCompactedFont | TrueType (TTF), Web font (WOFF) |
| Video | DefineVideoStream, VideoFrame | FLV format without audio |
| Sound | DefineSound, SoundStreamHead1-2, SoundStreamBlock | MP3, WAV, FLV |
| Frames | | PNGs, BMPs, GIF, AVI, SVG(limited), HTML 5 Canvas, PDF |

**Table 2.4** AS3 Export Capabilities of JPEXS Decompiler

| Asset | Tag | Export Format Supported |
|---|---|---|
| Scripts | DoABC, DoABCDefine, DoInitAction, DoAction, DefineButton tags, BUTTONCONDACTION, CLIPACTIONRECORD | ActionScript, P-code, P-code with hex, hex, P-code GraphViz, Hex, Constants, ActionScript method stubs optional all to single file export |

  - imports
  - type annotations, including Vector.<type >
  - field initializers
  - default values for optional method parameters
  - include directives
  - same-file helper classes

  The following concepts are not (yet) or only partially supported:

  - the with statement
  - package-scope functions
  - other member visibilities (internal, protected) and custom namespaces are syntactically accepted but ignored
  - XML initializers and query operators
  - annotations

- **Compiler**
  Jangaroo's central tool is the ActionScript-3-to-JavaScript compiler jooc. It takes source code written in a subset of ActionScript 3 and translates it into JavaScript 1.x that is understood by current browsers and other JavaScript engines like Rhino[16].

  The compiler can generate debuggable code that is very similar to your ActionScript 3 source code, or deployment code that is more compact. We provide a Maven[13] plug-in, a command line version, and an Ant[2] task to help you integrate the compiler into your build process.

  A small runtime written in JavaScript takes care of interpreting the concise compiled class format.

## 2.5 Technologies for HTML5

There are 4 major technologies to render and add interactivity in HTML5. We will discuss each one of them in subsequent sections.

### 2.5.1 SVG: Scalable Vector Graphics

SVG [37] stands for Scalable vector graphics. It is an XML based format for drawing images. It is supported natively by all browsers and is easy to understand and follow. Interactivity to SVG objects can be done by adding DOM elements like <button >Tags or by using CSS. Since it creates a DOM object for every image, which is stored in memory, SVG is not suitable when one wishes to render large number of objects on screen. Listing 2.5 is SVG code to render a black rectangle.

```
<svg width='100' height='100'>
  <rect width='50' height='50' x='10' y='10' fill='black' />
</svg>
```

**Table 2.5** SVG code to create a black rectangle

```
const canvas = document.getElementById("myCanvasElement");
const context = canvas.getContext("2d");
context.rect(10, 10, 50, 50);
context.fillStyle="black";
context.fill();
```

**Table 2.6** JS code for creating a rectangle via canvas

### 2.5.2 Canvas

Canvas[39] is another tool to render graphics on web. It is supported in HTML5 via <canvas >Tag. Since canvas is an API it requires Javascript to draw using canvas. It is more efficient than SVG in terms of menory since Canvas produces rasterized images. Interactivity in Canvas elements can be added via JavaScript. Listing 2.6 is Canvas code to render a black rectangle. In the proposed approach we use canvas for rendering most of the content in HTML5.

### 2.5.3 WebGL

WebGL [58] is also used to render graphics with performance being key differentiating factor. It provides API to render rasterized images using low level code which executes on GPU instead of CPU which results in faster and more efficient rendering.

*Chapter 3*

# Literature Review

## 3.1 Move towards HTML5

Even prior to the decline of Flash, there have been effort to migrate towards open standards. In 2003, Concolato [31] presented a tool that converts Flash to SVG and a technique to add interactivity to the SVG using SMIL(Synchronized Multimedia Integration Language). Prior to that, Probets et al. [61] presented a comparison between Flash vector graphics and SVG. Their paper described how shapes could be converted to SVG (Scalable Vector Graphics). However there is no discussion about how to add interactivity in shapes. The primary advantage of using Flash is the interactivity and this seems to be lost during the conversion process.

Flash Player was created for the PC era, but when it comes to mobile devices, it fell short due to the following reasons: Openness, reliability, security, performance and substandardized quality of 3rd party development tools. After Steve Jobs open letter in 2010 [49], there was a sharp decline in usage of flash (graph 3.1 taken from Statistica[4]). Popular websites using Flash like Twitch and Youtube discontinued using Flash in 2016[21] and 2015[22] respectively.

As of 2020, any flash content (either built in or published as Flash) may no longer work[50]. Therefore developers are now forced to:

- Retire the app, course, program or other asset
- Convert the published content to HTML5 via Adobe Animate CC (Present name of Adobe Flash)
- Rebuild/redesign the app, course or other content from scratch.

In some cases, a course or other asset may have to be re-designed or revamped altogether, or you may be able to be leverage a portion of the content for your redesigned content. It really depends what's contained in the published content including how many any animations you have, and how many flash units are contained in your published app, course of other content.

Of course, converting the Flash content only applies to developers who have the source files. If you don't own the source files and have purchased flash content including an app, game, animation or course, for example, you may have to replace your content or program altogether.

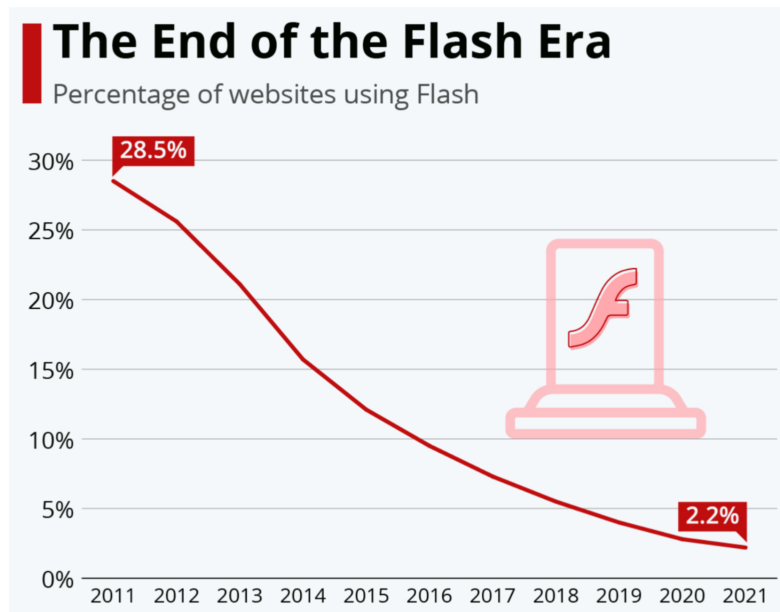


**Figure 3.1** Decline in Flash Usage

Given the number of Flash based applications, it is impossible to rewrite all these applications using HTML5. Repositories like NewGrounds have a about 8000+ flash games[38]. Just like a migration of a typical legacy system written in orphaned technology, one has to migrate all Flash based applications that are still economically viable to HTML5 to support evolution. However moving from Flash to HTML5 is a laborious task requiring lot of re-coding. This is because one cannot write wrappers around .swf (binary format) files to use it in HTML5. The entire application or website needs to be written from scratch. Rewriting the entire application in HTML5 from scratch is not ideal for the following reasons:

- Developers have a large number of assets like images, vector graphics and animations in Flash and rewriting them can be a resource intensive effort
- In many cases the source .fla files (Flash source files) for given .swf files may not be accessible.
- Tooling in HTML5 is immature as compared to Flash, so starting from scratch can take longer duration for developing the same application. A study by VESELÁ et.al. [73] in 2014 concluded that Flash is easier to use as compared to HTML5 and CSS3. The study however didn't compare ActionScript with JavaScript. The Java style class model used by ActionScript (scripting language for Flash) is more convenient and easier to maintain than JavaScript.

## 3.2 Existing Solutions

To facilitate this shift from Flash to HTML5, a tool or set of tools that automatically transform Flash programs to open standards is needed. The existing work falls under two categories:

- Flash players in JavaScript
- Tools for migration to HTML5

### 3.2.1 Flash Players in JavaScript

Smokescreen[18] and Gordon[9] are Flash players written in JavaScript. Gordon uses SVG for rendering and does not support ActionScript. Smokescreen partially supports ActionScript 2.0 and uses Canvas for rendering.

Shumway [17] [56] is Flash player Firefox extension created by Mozilla. It is written in TypeScript[27] and partially supports ActionScript3 APIs. The architecture of Shumway is shown in fig 3.2. The API support for this player comes from two places:

- AS3 Flash APIs and Builtin APIs which are compiled in form AS3 bytecode or .abc files
- Flash APIs implemented in TypeScript by Shumway

Shumway provides rendering support using both Canvas and WebGL.

**Limitations:**

Despite removing the need for Flash plugins, this does not help developers as they again depend on Adobe Flash for making changes to their animations. Another issue interpreting in JavaScript has its own performance overheads.

### 3.2.2 Migration to HTML5

There are various tools which aid migration to HTML5. Based on extent of manual efforts or inputs needed to complete the migration we classify the existing tools into three categories:

- Fully Automated
- Semi Automated
- Manual Migrations

#### 3.2.2.1 Fully Automated: Swiffy - Flash to HTML5 converter

In a fully automated migration very few or no inputs need to be provide by end user to carry out the end to end migrations. Google Swiffy is one such example that completely migrated most banner ads from Flash to HTML5 and JS.

Google Swiffy[65], that initially started of as a Masters thesis by P. A. M. Senster addresses the problem of converting a SWF file to HTML5 to a certain extent. As shown in figure 3.3 , Swiffy has two main components: a server-side compiler and a client-side interpreter. The server-side compiler is responsible for parsing the SWF file and transforming it into Swiffy bytecode. The bytecode will be sent to the browser in the form of JSON, which can be contained in a JavaScript file, an HTML file or a .json

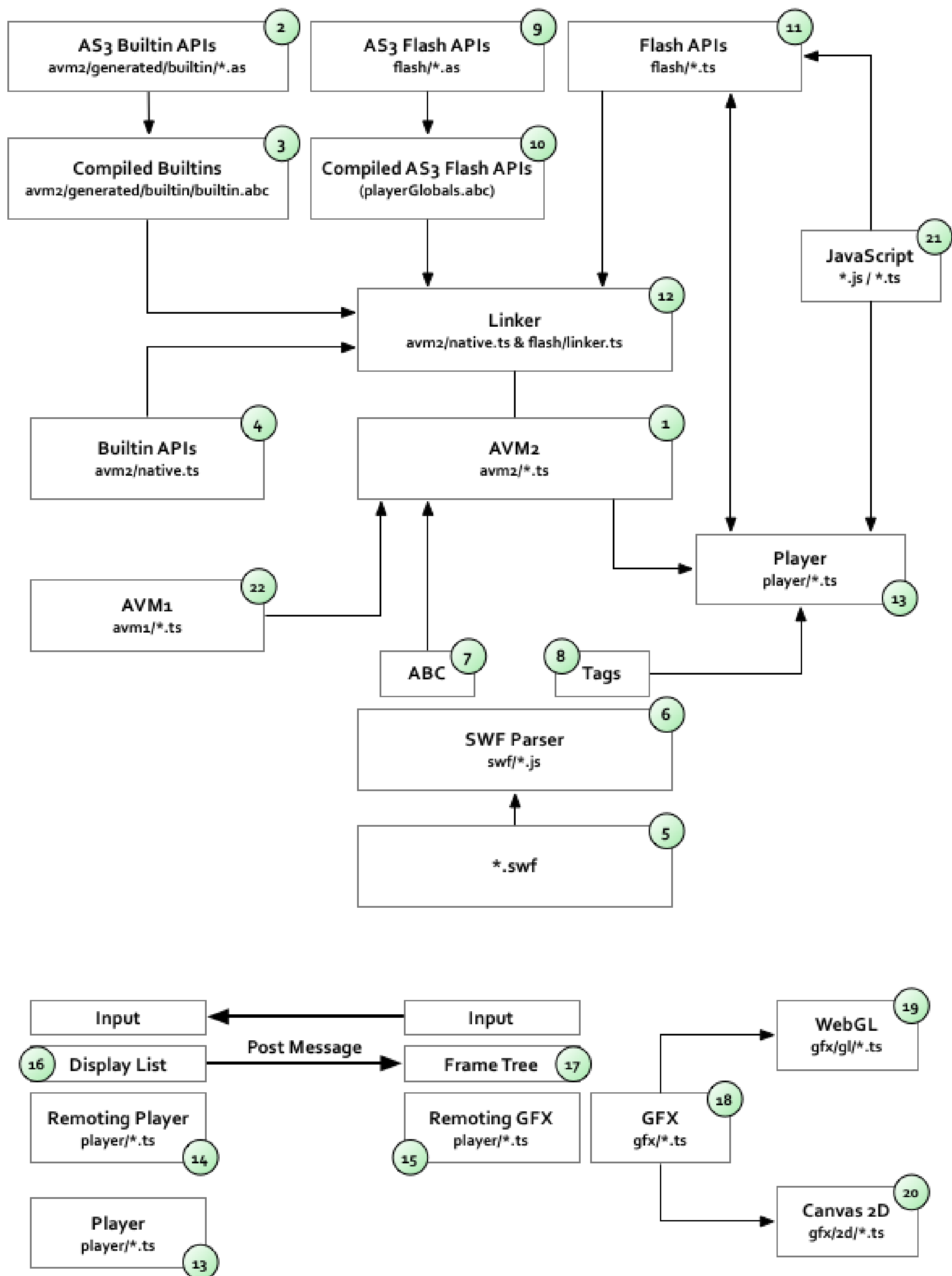


**Figure 3.2** Mozzilla Shumway Architecture

```
var circle:Shape = new Shape();
circle.graphics.clear();
circle.graphics.lineStyle(1,0x000000);
circle.graphics.beginFill(0xD6D6D6);
circle.graphics.drawCircle(0,0,100);
circle.graphics.endFill();
addChild(circle);
circle.x = stage.stageWidth / 2;
circle.y = stage.stageHeight/ 2;
```

**Table 3.1** AS3 code for creating a circle

file. The client-side interpreter is a JavaScript runtime library. The runtime library consists of Bytecode Interpreter, Rendering Engine, ActionScript VM and ActionScript Runtime Libraries. Swiffy uses

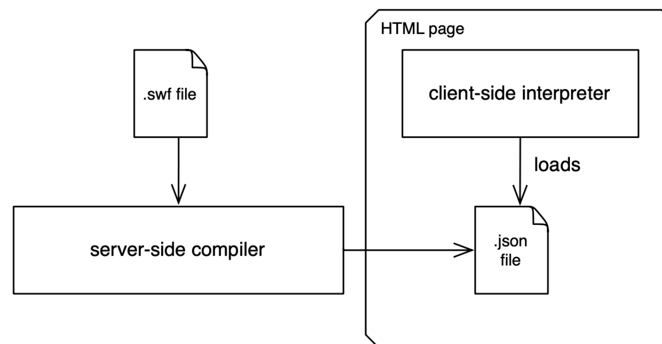


**Figure 3.3** Swiffy Architecture

a Javascript interpreter to execute the ActionScript bytecode present in JSON format. This interpretive approach suffers from two major limitations:

The first limitation is that the final output is not editable. For example consider the SWF file created by the ActionScript3 (AS3) code shown in Listing 3.1. The AS3 code draws a circle of radius 100, at the center of the drawing stage. The properties of the circle in the Swiffy output cannot be changed unless they are changed in the source AS3 files. If a change is to be made, the SWF file is regenerated and again converted by Swiffy. This is because the generated Swiffy Object is a JSON bytecode which is not human readable due to two reasons: (1) The bytecode specification is not provided. (2) It consists of low-level instructions which are difficult to comprehend. Hence making changes in Swiffy Object is very difficult. Figure 3.4 is a screen-shot of the Swiffy object. This blocks the developer from editing and extending the output, which could have otherwise been useful for integrating it with other JavaScript code.

**Is a fully automated transformation possible?** Although theoretically it is possible to completely automate the migration from Flash to HTML5. There are certains factors that limit this practically:

```
    <script>
      swiffyobject =
{"internedStrings":["STATIC_PROTECTED","PACKAGE","PRIVATE"],"tags":[{"frames":[],"scenes":[{"name":"Sce
ne
1","offset":0}],"type":23},{"classes":[{"ns":3,"cinit":0,"ctraits":[],"name":1,"init":1,"traits":[],"su
pertype":2}],"methods":[{"exceptions":[],"locals":1,"traits":[],"params":[],"code":"0DBH","type":0,"opt
ionals":[]},{"exceptions":[],"locals":1,"traits":[],"params":[],"code":"0DDQSQDQXQNKAwBoBNBmBGYFTwYA0GY
EZgUkASQATwcC0GYEZgUtAU8IAdBmBGYFJAAqJGRPCQPQZgRmBU8KAF0L0GYETwsB0GYEYAxmDSQCo2EO0GYEYAxmDyQCo2EQRw
\u003d
\u003d","type":0,"optionals":[]},{"exceptions":[],"locals":1,"traits":[],"params":[],"code":"0DBlAGARMG
ASMGATMGAUMGAVMGAWMGACMGACWAAdHR0dHR0daAFH","type":0,"optionals":[]}],"scripts":[{"init":2,"traits":[{"
writable":true,"name":1,"value":0,"slot":1,"kind":"classes"}]}],"multinames":[{"ns":1,"name":2,"kind":7
},{"ns":2,"name":4,"kind":7},{"ns":2,"name":5,"kind":7},{"ns":1,"name":6,"kind":9},{"ns":1,"name":14,"k
ind":9},{"ns":1,"name":15,"kind":9},{"ns":1,"name":16,"kind":9},{"ns":1,"name":17,"kind":9},{"ns":1,"na
me":18,"kind":9},{"ns":1,"name":19,"kind":9},{"ns":1,"name":20,"kind":7},{"ns":1,"name":21,"kind":7},{"
ns":1,"name":22,"kind":7},{"ns":1,"name":23,"kind":9},{"ns":1,"name":24,"kind":7},{"ns":1,"name":25,"ki
nd":9},{"ns":1,"name":26,"kind":7},{"ns":15,"name":28,"kind":7},{"ns":2,"name":29,"kind":7},{"ns":2,"na
me":30,"kind":7},{"ns":2,"name":31,"kind":7},{"ns":2,"name":32,"kind":7}],"namespacesets":[[4,5,1,6,7,3
,8,9,10,11,12,13,14]],"strings":["","","Main","flash.display","MovieClip","Shape","circle","http://adob
e.com/AS3/2006/builtin","flash.display:MovieClip","flash.display:Sprite","flash.display:DisplayObjectCo
ntainer","flash.display:InteractiveObject","flash.display:DisplayObject","flash.events:EventDispatcher"
,"graphics","clear","lineStyle","beginFill","drawCircle","endFill","addChild","stage","stageWidth","x",
"stageHeight","y","Object","flash.events","EventDispatcher","DisplayObject","InteractiveObject","Displa
yObjectContainer","Sprite"],"type":18,"uints":[0],"ints":[0,14079702],"namespaces":[{"name":1,"kind":"#
1"},{"name":3,"kind":"#1"},{"name":2,"kind":"PROTECTED"},{"name":0,"kind":"#2"},{"name":0,"kind":"#
2"},{"name":1,"kind":"PACKAGE_INTERNAL"},{"name":7,"kind":"NAMESPACE"},{"name":2,"kind":"#
0"},{"name":8,"kind":"#0"},{"name":9,"kind":"#0"},{"name":10,"kind":"#0"},{"name":11,"kind":"#
0"},{"name":12,"kind":"#0"},{"name":13,"kind":"#0"},{"name":27,"kind":"#
1"}]},{"references":[{"id":0,"name":"Main"}],"type":19},{"type":2}],"fileSize":1053,"v":"7.2.0","backgr
oundColor":-1,"frameSize":{"ymin":0,"ymax":4800,"xmin":0,"xmax":6400},"as3":true,"frameCount":1,"frameR
ate":30,"version":15};
    </script>
```


**Figure 3.4** Generated Swiffy Object

- **Differences in Flash and HTML5** A number of Flash features like movieClips, stages, morphshapes though can be emulated in HTML5 have overheads and are missing in many APIs. Instead HTML5 relies on spriteSheets for similar animations. Porting from the former to the later is quite a tedious task and requires human intervention.

- **Issues in Decompilation** Although Flash files are easily decompiled, its content can still be obfuscated, and in many cases its difficult to establish relationship between the code and assets, again requiring human intervention.

- **Open Flash Like APIs are not complete:** The open Flash like APIs are incomplete which poses another challenge in migrating all the functionality.

#### 3.2.2.2 Semi Automated Migrations

In semi automated migrations, users needs to perform some actions manually withing the process of migrations. However tools are available to do most of the heavy lifting like export assets, transpiling code to a different language. User however needs to ensure mapping between code and assets, mapping between APIs of source and target language, support missing APIs, etc. In case of video and banner advertisements, the viable content that needs migration is the assets like the video, sprites, images and not the code logic because it may be trivial and minimal in terms of effort needed to create the assets. There are a number of tools available online to extract videos from SWF files and export it to suitable formats. Google Swiffy performed really well for flash advertisements. Automation makes sense only if the number of assets to migrated are huge or the logic that needs to be migrated is non trivial in terms of effort needed or source code (FLA files) is absent.
Below are a list of use cases where semi-automated migration become viable:

- Open web games

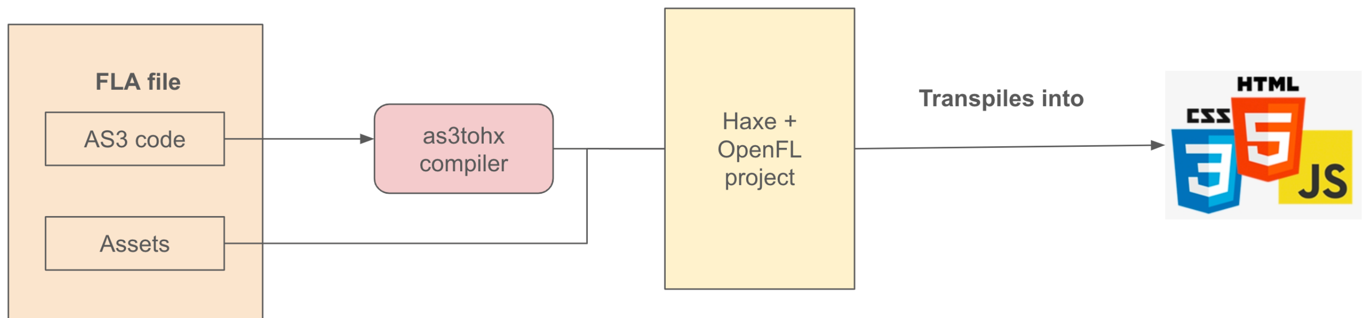


**Figure 3.5** Migration using HAXE

- Real-time apps with WebSockets: Flash TCP sockets have been commonly used to create real-time multiuser apps such as chat apps, but they can be replaced with WebSockets.
- Clipboard access: Flash's clipboard API used to be the only available means for creating advanced clipboard functionality, but this is now available in web standards.
- Flexible user interfaces with CSS: Adobe Flex used to provide a way to effectively create flexible user interfaces on the web, but the open web platform now boasts powerful CSS layout systems like Grid and Flexbox to close the gap.
- Camera/Microphone access with WebRTC: Developers used to rely exclusively on Flash for accessing the user's camera and microphone, but this can now be achieved with WebRTC and related technologies.

We will discuss two major tools here:

- **Haxe: An Alternative for Flash Developers:** Haxe[29] is a cross-platform toolkit which is growing in popularity. Regarded as open source flash [44], it is similar to ActionScript and hence it is one of the easiest options for conversion, in particular if used through OpenFL[14]. Haxe can compile to several languages including JavaScript. Hence, it is popularly used as a write once, transpile everywhere language enabling developers encapsulate business logic in a single place [25]. Being a multiplatform language, it is widely used for migration of C/C++ or Java code to mobile platforms [34], [35], [55] [36], [57], [28].
  In order to convert Flash content to HTML5 one can first convert their AS3 code to Haxe. Some automatic conversion tools are currently available, such as as3hx and as3tohx[3], which can convert a significant part of the code. Some modifications will be needed to map the AS3 libraries with OpenFL libraries in Haxe. Also the assets will need to be set manually in a haxe file. Post this one can compile the entire project to JavaScript, which is the desired goal. This is summarized in figure 3.5
  Following are **limitations** of this approach:
  - It requires source FLA files that are not always accessible.

  - as3tohx transpilers are not complete
  - Mapping AS3 APIs can be difficult and they may often not exist.

- **Using Adobe Animate CC:** Adobe animate CC [1] allows exporting FLA files to HTML5, JS, CSS. It allows exporting Sprites and Movieclips as spritesheets, which can then be animated via CreateJS toolkit. However it does not export AS3 code, but comments it in the output. It aids user by providing them with code snippets of JS code corresponding to AS3 code.
  **Limitations:**
  However source FLA files are not always available. A lot of time such migrations are infeasible when source animations use DisplayLists and other non AS3 features[64].

#### 3.2.2.3 Manual Migration

Rewriting in HTML5 requires a lot of effort for the following reasons:

- One needs to explore the animation and list down all it's distinct states, assets that will be required and their interactions with the user.
- When source files are unavailable or asset can't be exported in a desired format, effort is needed in recreating all the assets like images, buttons, etc.
- Writing the JS logic for animation and HTML5/CSS code to place the assets and to create the main containers where the animation resides is both skill and time intensive.

Many a times the asset that requires migration may have minimal to no support in HTML5 and may require wrapping flash widgets[60] or building respective libraries in JS. Identification of equivalent tooling HTML5 is also an intensive process [62]. Also adoption of new HTML5 frameworks often requires skill sets that are difficult to find[24]. Development in Adobe Flash was significantly superior than HTML5 in terms of User Interface, animations, drag and drop and debug-ability[63]. Multiple other works [66], [43] arrive at a similar conclusion.

After the decline of Flash, lots of HTML5 tools have come up that offer Flash like features and mobile support. These are however very specific to eLearning Solutions.
Adobe Animate CC is a proprietary solution for making games and PhaserJS is another popular open source option.

*Chapter 4*

# Proposed Approach

## 4.1 Overview

To address the problems of readability and efficient execution we use decompilation and transcompilation as part of our transformation process. Decompilation of SWF bytecode allows us to extract AS3 code, along with various assets like sprites, images, fonts, text, etc. The extracted AS3 code and its transcompilation to JavaScript will thus be human readable. Since code will be transcompiled to JavaScript it will not have interpretation overheads that normally exist in interpreting AS3 components.

In our approach we can directly export the contents of decompiled SWF file to JavaScript as shown in 4.1 or chain decompiler's AS3 output to a transcompiler as shown in 4.2. We chose the latter approach because:

- It provides the developer flexibilty to edit his work in both ActionScript and JavaScript. The ability to edit work in ActionScript is advantageous because it allows one to use the features present in ActionScript which are not present in JavaScript, like dynamically and statically typed code, packages, classes and interfaces, etc. This makes more complex applications easier, less error-prone and more cost-effective to extend.

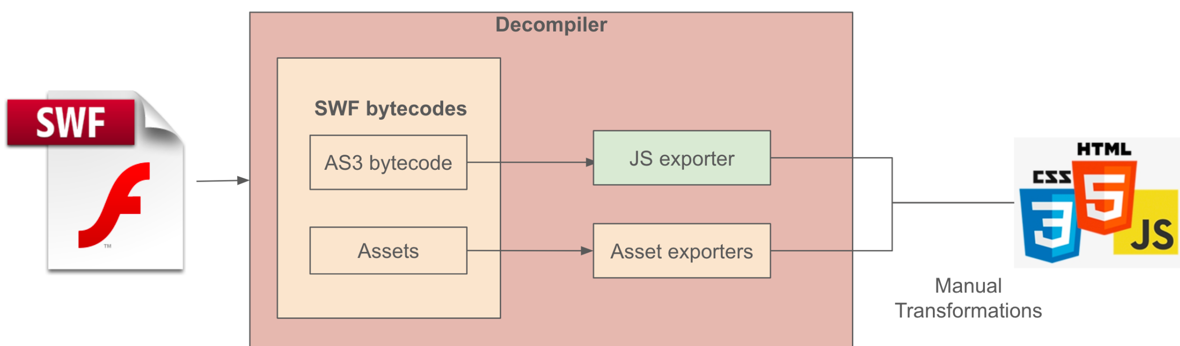


**Figure 4.1** Option1: Exporting JS code

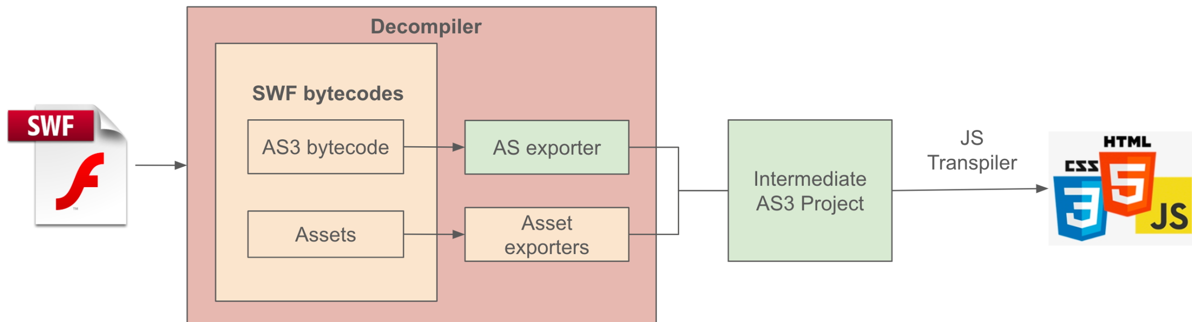


**Figure 4.2** Option 2: Exporting AS3 and then transpiling to JS

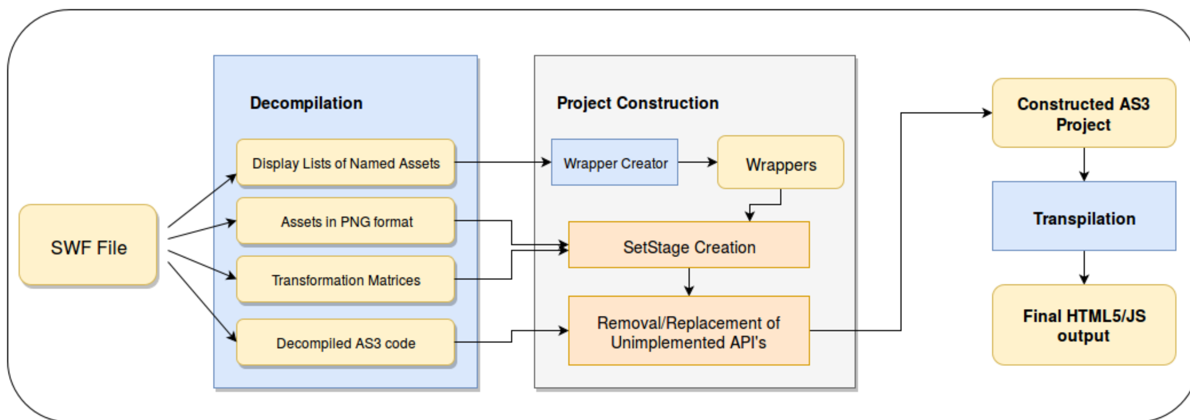


**Figure 4.3** Transformation Process

- It also helps those who only wish to migrate their ActionScript assets from Adobe Flash.
- It gives us flexibilty to change our transcompiler when desired.

## 4.2 Proposed Approach

We propose a tool: SWF2JS that aids in the following three step transformation process for migrating the .swf files to HTML5/Javascript. 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. The entire process is summarized in figure 4.3.

### 4.2.1 Decompilation

Here the SWF file is taken as input by the decompiler to output the following content:

- Sprites, Images, Buttons in PNG format. We choose this format since it is efficient compared to Canvas and Svg options in terms of runtimes and overheads.

- Transformation matrices for each *named asset*. A *Named asset* is an asset created in Flash Professional tool which is referenced in AS3 code.
- *DisplayList* of each named asset in terms of smaller asset. *DisplayList* contains display objects and precedence of how they are displayed in animations.
- AS3 (ActionScript3) code after decompilation.
- Each frame of MovieClip is exported as a PNG image and then converted into a spriteSheet.

### 4.2.2 Project Construction

After exporting all the components to AS3 format, it is necessary to append a bootscript to it. This because SWF file is declarative in nature. It defines what has to done and not how it is to be done. The purpose of bootscript / runtime script is to initiate execution of the exported components in desired order. This bootscript/runtime will be an AS3 file called Main.as and will be common for all SWF files that are to be transformed. This gives us the final AS3 project. The AS3 code, exported assets and their arrangements will be linked to a setStage function in the Main.as file. At this point still need to ensure if all functions referenced in AS3 code are present in transpilers implementation, if any required asset is missing or incompatible etc. We have organised these actions into 4 major steps as below.

- 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.
- 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.
- Lastly all the functions which are not implemented in transpilers Flash API are removed, or replaced as appropriate.
- Implementing AS3 wrappers of any required JS library.

The constructed project uses Maven for building and deploying since Jangaroo has easy integrations with that tool and also it improves structure and organisation of intermediate project.

### 4.2.3 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

| Tag Name | Value |
| --- | --- |
| Header | <some_predefined_values > |
| DoABC | <abc_bytecode > |
| ShowFrame | |
| End | |

**Table 4.1** Trace for a simple blue rectangle drawn via AS3

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.

## 4.3 Example Walkthrough

**Figure 4.4** Rectangle

Consider an example of a SWF file that draws a blue rectangle as shown in Figure 4.4 using AS3 code shown in table 4.2. The AS3 code creates a blue rectangle by first setting the color of the point. Then it moves pointer to pixel coordinates (298,173) and draws four lines along coordinates (133,173), (133,73), (298,73), (298,173) after deducting 165 pixels along horizontally, followed by 100 vertically, followed by adding 165 pixels, and followed by 100 vertically completing the rectangle. The trace for SWF file(table 4.1) declares a $header$ with values for number of tags, framerate, frames, etc and a $DoABC$ tag with some <abc_bytecode >and finally a $ShowFrame$ tag.

Table 4.3 shows the AS3 expected exported output. Few notable differences in the output are:

- presence of SWF metadata that is embeded in AS3 code. This is provided as an output of decompiler and the values are in turn derived from $Header$ tag from SWF file.

- import of MovieClip object. MovieClips are complex SWF Sprites that support multiple frames and hence are a proxy of a mini SWF file of it own. We leverage this property of movieClip to create Main.as in every AS3 project, where movieClip seres as a starting point of the animation.

```
package {
        import Flash.display.Shape;
        public class Main {
                var rect:Shape = new Shape();
                public function Main() {
                        // constructor code
                        addChild(this.rect);
                        this.rect.graphics.beginFill(0x0066CC,1);
                        this.rect.graphics.lineStyle(1, 0x000000);
                        this.rect.graphics.moveTo(298,173);
                        this.rect.graphics.lineTo(133,173);
                        this.rect.graphics.lineTo(133,73);
                        this.rect.graphics.lineTo(298,73);
                        this.rect.graphics.lineTo(298,173);
                        this.rect.graphics.endFill();
                }
        }
}
```

**Table 4.2** AS3 code used to create a rectangle

- we notice that the decompiler is able to retrieve *rect* variable name since it is a class member and not a local variable.

Table 4.4 shows the final transpiled JS output for the corresponding AS3 code. We notice that JS output is readable but it has a lot of boilerplate code around it since, the AS3 objects are converted to jangaroo classloader objects.

```
package {
        import Flash.display.MovieClip;
        import Flash.display.Shape;
        [SWF(
             backgroundColor='0xFFFFFF',
             frameRate='30',
             width='320',
             height='240')]
        public class Main extends MovieClip {

                var rect:Shape = new Shape();
                public function Main() {
                        // constructor code
                        addChild(this.rect);
                        this.rect.graphics.beginFill(0x0066CC,1);
                        this.rect.graphics.lineStyle(1, 0x000000);
                        this.rect.graphics.moveTo(298,173);
                        this.rect.graphics.lineTo(133,173);
                        this.rect.graphics.lineTo(133,73);
                        this.rect.graphics.lineTo(298,73);
                        this.rect.graphics.lineTo(298,173);
                        this.rect.graphics.endFill();
                }
        }
}
```

**Table 4.3** Expected exported AS3 output for Shape

```
joo.classLoader.prepare("package",/*  {
        import Flash.display.MovieClip;
        import Flash.display.Shape;*/
        {SWF:{
              backgroundColor:'0xFFFFFF',
              frameRate:'30',
              width:'320',
              height:'240'}},
        "public class Main extends Flash.display.MovieClip",
        7,function($$private)
        {;return[
                "var",{ rect/*:Shape*/ :function()
                        {return( new Flash.display.Shape());
                        }
                       },
                "public function Main",function Main() {
                   Flash.display.MovieClip.call(this);
                   if(0===0){this.rect=this.rect();
                }
                        // constructor code
                        this.addChild(this.rect);

                        this.rect.graphics.beginFill(0x0066CC,1);
                        this.rect.graphics.lineStyle(1, 0x000000);
                        this.rect.graphics.moveTo(298,173);
                        this.rect.graphics.lineTo(133,173);
                        this.rect.graphics.lineTo(133,73);
                        this.rect.graphics.lineTo(298,73);
                        this.rect.graphics.lineTo(298,173);
                        this.rect.graphics.endFill();
                },
        undefined];},[],
        ["Flash.display.MovieClip","Flash.display.Shape"], "0.8.0", "0.9.9"

);
```

**Table 4.4** Transpiled output to JavaScript for the rectangle shape

*Chapter 5*

# Validations

We aim to evaluate the effectiveness of solution in terms of effort needed for migrating and maintaining it. A lot of work has been done on comparing performances in terms of runtimes and compile times. Stepasyuk et.al. [68] evaluated performance between Haxe-compiled and target-specific language code in their work. While Iakovenko et.al. [45] had a focus on programming time and performance, only performance was evaluated and programming time was assumed to be trivially less. Similar Vstrekelj et.al. [69] only tested performance times of developed games on three different hardware configurations, with three different complexity settings, and directly stated that programming time was low. P.A.M Senster [65] in their work measured performance, accuracy and runtimes of their conversions.

## 5.1 Evaluating Migration Effort

### 5.1.1 Animation Selection

For all the below evaluations, 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. Later, we will try to implement a few features on the migrated animations to compare them from perspective of maintenance. We picked six flash based experiments(animations) from Virtual Labs repository[71], a government 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.
Below is a brief description of the selected animations.

- Kruskal: This animation emulates the kruskal's algorithm. It takes user inputs in form of nodes, then edges, then weights via UI button interactions. Then the MST is algorithm is emulated and shown to user. Refer fig 5.1 for UI.

- DFS_BFS: This animation emulates the DFS and BFS algorithm. It takes user inputs in form of nodes, then edges, then weights, and finally the algorithm to be emulated via UI button in-

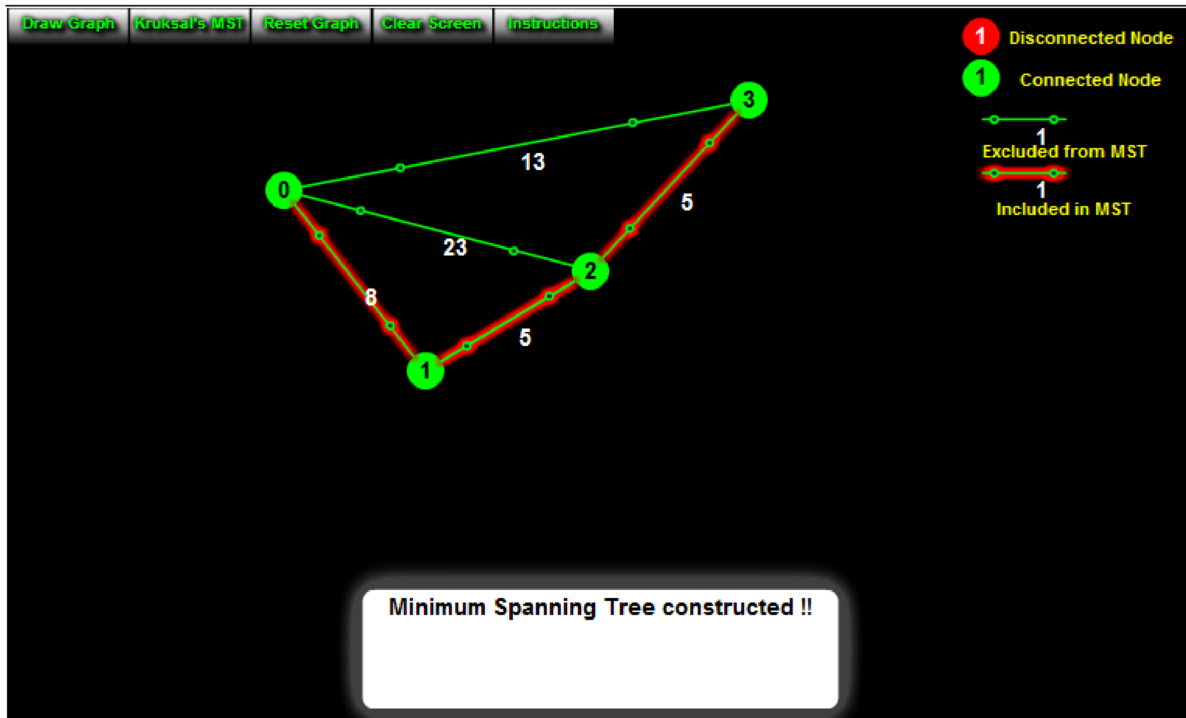


**Figure 5.1** Kruskal

teractions. Then the selected is algorithm is emulated and shown to user. Refer fig 5.2for UI.

- Expression Tree: This animation emulates expression tree algorithms, basically Prefix, InFix and PostFix algorthims. It takes user inputs in form of text and form button and later emulates the selected algorithm forming the expression tree of the same. Refer figure 5.3 for UI.
- BST: This algorithm emulates binary search tree algorithm. It takes user inputs in form of text and form button and later emulates the binary search tree. Refer figure 5.4 for UI.
- Tensions: This is a static emulation of tension and takes input only in form of Start button. Refer figure 5.5 for UI.
- Orifice: This is a static emulation of fluid running through orifices and takes input only in form of Start button. Refer figure 5.6 for UI.

### 5.1.2 Metrics to objectify complexity of an animation

Since animations are abstract in nature it is necessary to objectify them using some metrics in order to measure complexity for any evaluation. Hence, as a reference point we record the following metrics for each animation (shown in Table 5.1):

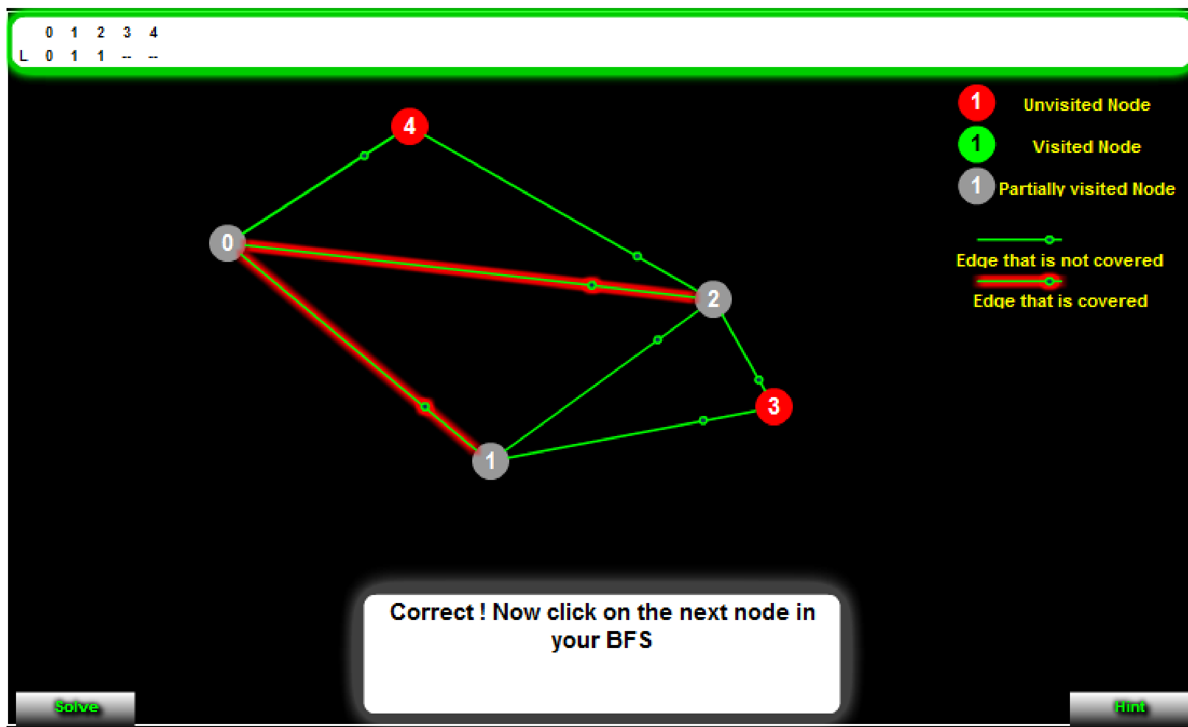


**Figure 5.2** DFS BFS

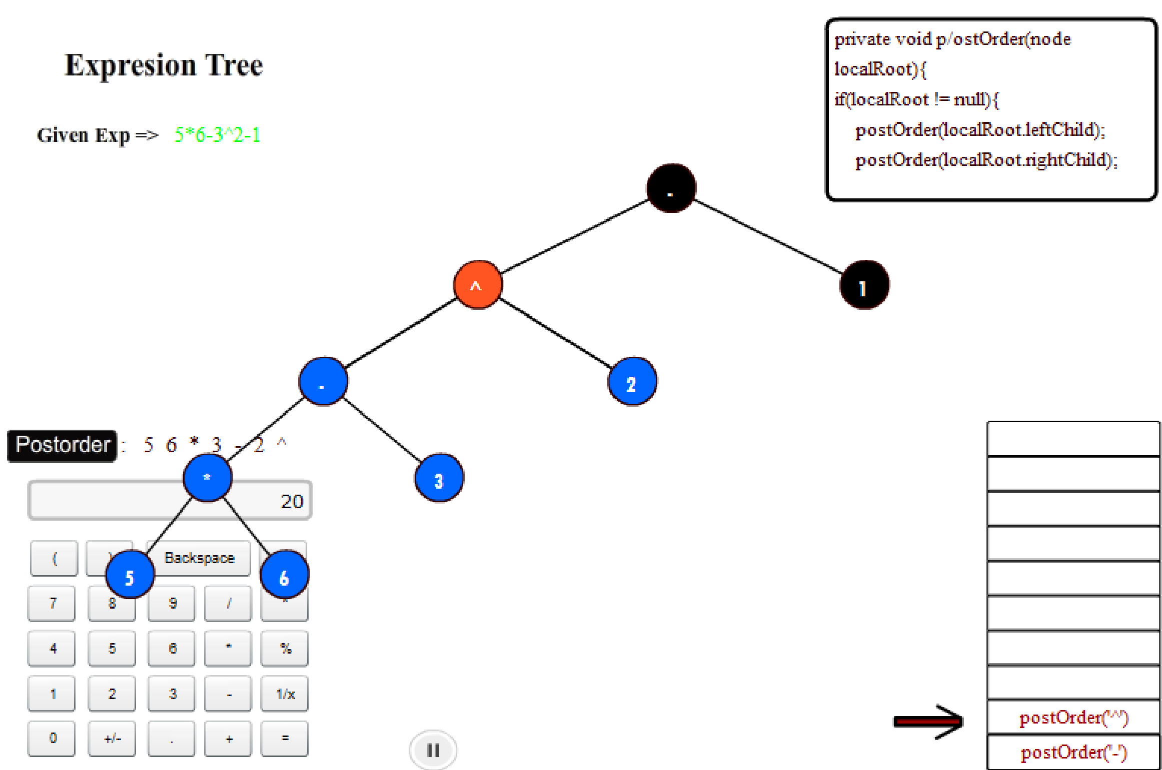


**Figure 5.3** Expression Tree

Insert Delete Search

9 is found at the position
5

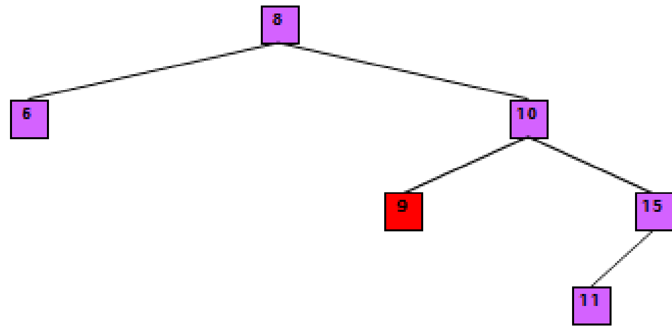


**Figure 5.4** BST

Tension Test

Start

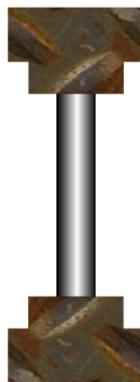

**Figure 5.5** Tensions

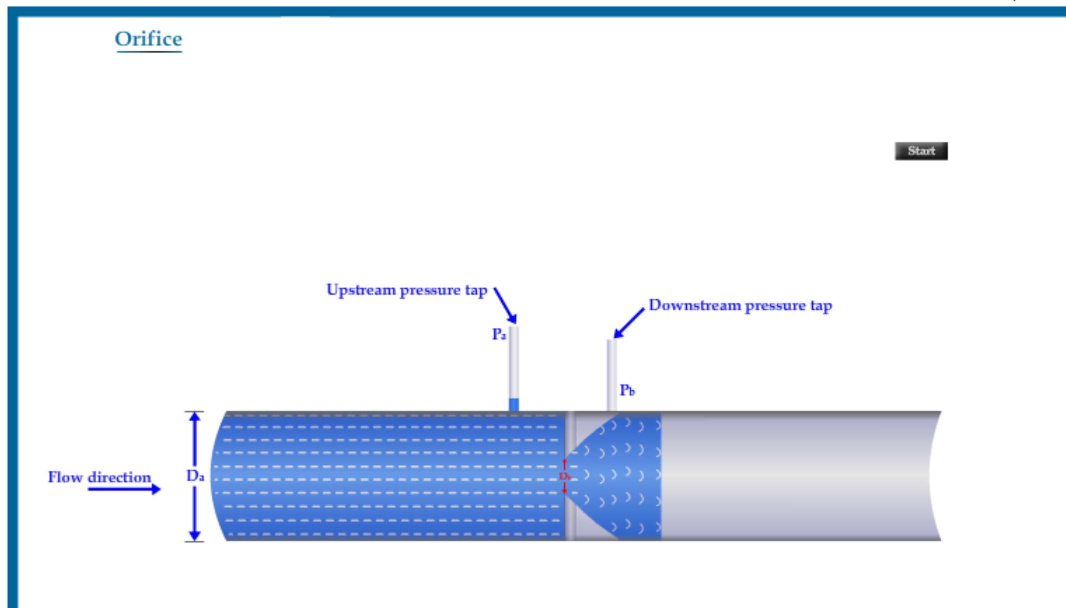


**Figure 5.6** Orifice

- **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.

- **Function Points** : Our working definition of Function Points is based on ISO/IEC 19761:2003 [48]. 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.

- **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.

From the above LOC and number of classes are standard parameters, however these can't be compared across languages. To deal with this we used function points to determine complexity of animations. However these too do not give a complete picture of animations, since the number of assets can vary from animations to animations. Also the complexity of assets can vary. Hence we came up with

**Table 5.1** 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 |

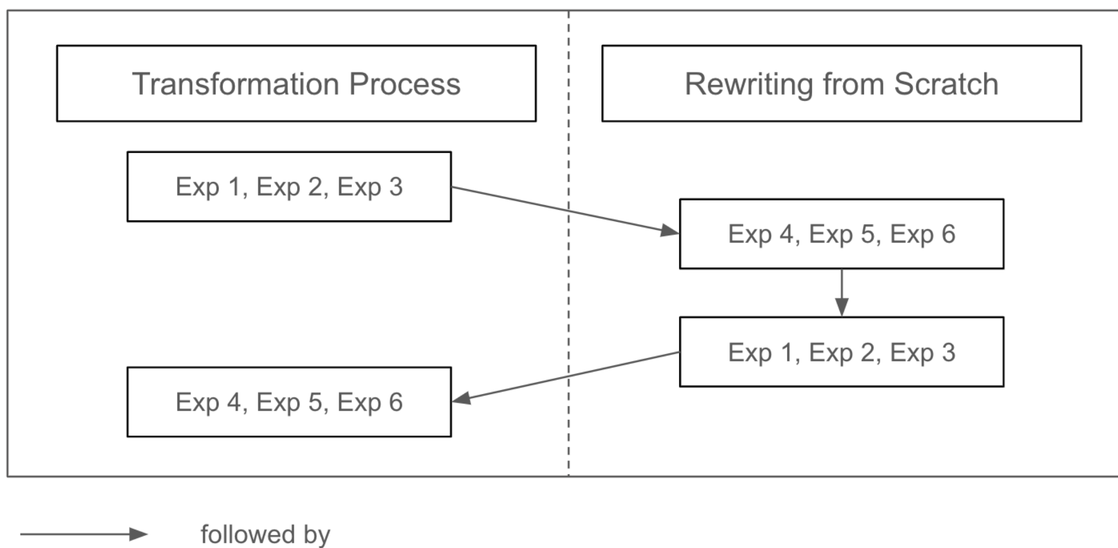


**Figure 5.7** Experimentation Order

a parameter of Named Assets which are assets that will interact with code. Based on extent animation complexity we have further classified these assets.

### 5.1.3 Experiment Setup

The experiment was performed by a single subject. In order to remove familiarity bias and prevent skewing of results, we transformed Exp1, Exp2, Exp3 using our process, then Exp4, Exp5, Exp6 were rewritten from scratch. This was followed by rewriting Exp1, Exp2, Exp3 from scratch. In the end Exp4, Exp5, Exp6 were transformed using our process 5.7.

#### 5.1.3.1 Transformation Process

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

**Table 5.2** 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 |

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.

#### 5.1.3.2 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.
- **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.

**Table 5.3** 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 |

### 5.1.4 Results and Inferences

Table 5.2 and Table 5.3 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 5.5 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. Cases where similar assets are reused repeatedly one might benefit from creating a custom tool with these assets [71], [72].

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

**Table 5.4** Evaluation of Maintenance

| Id | Feature to be added | Function Points | Transformation Process | | | Rewriting From Scratch | | |
|---|---|---|---|---|---|---|---|---|
| | | | 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 5.5** 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 |

themes and templates, which needn't be coded again. This can result in reduction of time taken for rewriting from scratch.

## 5.2 Evaluation of Maintenance

### 5.2.1 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.

### 5.2.2 Results

From table 5.4 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.

## 5.3 Threats to Validity

In our study, we try to compare the effort for enhancement in code bases between AS3 and JavaScript. Although both are same ECMA-262[32] 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 mitigate 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.

*Chapter 6*

# Conclusions

## 6.1 Problems with proposed approach

There are various problems associated with our approach:

- The developer is limited to the transpiler's implementation of Flash APIs. Since the number of Flash APIs is huge and their implementations are non trivial, complete flash functionality wouldn't be provided by the transpiler. However even basic Flash applications end up using some non popular Flash features that are not commonly used. Therefore the user either needs to replace this in the project construction phase by using third party AS3 libraries or implementing this functionality on his own.

- Although we have the final output in JavaScript, modifying this code is difficult as it is still representing the AS3 code as class objects in JS to be interpreted by JSClassLoader. Therefore code debugging is easy as type semantics corresponding to AS3 are present in the code however modification is difficult in such a format.

- The approach encourages code modifications and maintenance in AS3, which is no more a popular choice when it comes to developing rich content for browsers.

- Declaring the assets in AS3 code becomes tedious when the number of assets grow. This is because AS3 is a scripting language and not declarative language. Therefore declaring assets in AS3 increases lines of code and reduces readability.

## 6.2 Future Work

### 6.2.1 Improvements in Approach

To address the above shortcomings we can take inspiration from the availability many open flash like APIs. Some examples include CreateJS [54] in JS, StageXL for Dart [67], LayaAir SDK for AS3, TypeScript and JS, etc. This and the lack of popularity of AS3 motivates us to modify our transpiler by:

- Producing an output code that is actually JavaScript and not merely an object code to be used via ClassLoaders, thereby making it actually modifiable and more usable.
- Decoupling the FlashAPI implementations from the transpiler, and instead mapping these API calls to the open source APIs that are available. Therefore the developer is not completely dependent on the transpiler and can then choose to rely on these open APIs to replace Flash functionality.

We can also create a new exporter in the decompilation phase to export the assets in HTML timeline format, which can be referred in decompiled AS3 code and transpiled JS by using the 'id' attribute of the asset.

### 6.2.2 More Comparitive Studies

We can perform the comparitive studies with larger set of candidates to get a better read on ease of modifications and maintenance. Furthermore, new studies need to be done to evaluate effectiveness of direct JS migration of AS3 code if approach mentioned above is implemented.

### 6.2.3 Migration to other languages

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.

# Bibliography


[1] Adobe animate creatibe cloud. https://www.adobe.com/in/products/animate.html. Accessed: 2022-10-03.

[2] Ant. https://ant.apache.org/. Accessed: 2022-10-03.

[3] As3 to haxe converter. https://github.com/innogames/ax3. Accessed: 2022-10-03.

[4] End of flash era. https://www.statista.com/chart/3796/websites-using-flash/. Accessed: 2022-10-03.

[5] Jswiff open-source project. https://cwiki.apache.org/confluence/display/FLEX/FalconJx+Prototype. Accessed: 2022-10-03.

[6] Jpexs, free flash decompiler. https://www.free-decompiler.com. Accessed: 2022-10-03.

[7] Flexjs - mxml/as3 to html5/js transpiler. https://cwiki.apache.org/confluence/display/FLEX/FlexJS. Accessed: 2022-10-03.

[8] Gnu project free software foundation. gnash open-source project. https://www.gnu.org/software/gnash/. Accessed: 2022-10-03.

[9] Gordon - a flash player written in javascript. https://github.com/tobytailor/gordon. Accessed: 2022-10-03.

[10] Jangaroo tool. https://github.com/rsippl/jswiff. Accessed: 2022-10-03.

[11] Jswiff open-source project. https://github.com/rsippl/jswiff. Accessed: 2022-10-03.

[12] Alessandro pignotti. lightspark. https://sourceforge.net/projects/lightspark/. Accessed: 2022-10-03.

[13] Maven. https://maven.apache.org/. Accessed: 2022-10-03.

[14] Openfl - the open flash library for creative expression on the web, desktop, mobile and consoles. https://github.com/openfl/openfl. Accessed: 2022-10-03.

[15] Rabcdasm, robust abc (actionscript bytecode) [dis-]assembler. https://github.com/CyberShadow/RABCDAsm. Accessed: 2022-10-03.

[16] Rhino: Javascript in java. http://mozilla.github.io/rhino/. Accessed: 2022-10-03.

[17] Mozilla. shumway: Html5 technology experiment that explores building a faithful and efficient renderer for the swf file format without native code assistance. URL https://github.com/mozilla/shumway.

[18] Smokescreen - a flash player written in javascript. https://github.com/cesmoak/smokescreen. Accessed: 2022-10-03.

[19] Sothink swf decompiler. https://www.sothink.com/product/flashdecompiler. Accessed: 2022-10-03.

[20] Flash decompiler trillix. https://www.eltima.com/products/flashdecompiler. Accessed: 2022-10-03.

[21] Html5 video player beta begins today. https://blog.twitch.tv/en/2016/07/14/html5-video-player-beta-begins-today-135d1b7baa65. Accessed: 2022-10-03.

[22] Youtube drops flash for html5 video as default. https://www.theverge.com/2015/1/27/7926001/youtube-drops-flash-for-html5-video-default. Accessed: 2022-10-03.

[23] Zoe, generates spritesheets from swf. https://www.softpedia.com/get/Internet/WEB-Design/Flash/Zoe.shtml. Accessed: 2022-10-03.

[24] M. Ahmed and S. S. U. Rahman. A framework for smooth adoption of emerging technologies for rich internet application (ria) development focusing on html5. *Journal of Independent Studies and Research*, 10(2):18, 2012.

[25] B. F. Andrés and M. Pérez. Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE, 2017.

[26] E. Ashcroft and Z. Manna. The translation of'go to'programs to'while'programs. In *Classics in software engineering*, pages 49–61. 1979.

[27] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[28] P. N. D. Bilotto and L. Favre. Migrating java to mobile platforms through haxe: An mdd approach. In *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, pages 240–268. IGI Global, 2016.

[29] N. Cannasse. Haxe. *Too good to be True*, 2014.

[30] W. Chu. A re-engineering approach to program translation. In *1993 Conference on Software Maintenance*, pages 42–50, 1993. doi: 10.1109/ICSM.1993.366957.

[31] C. Concolato, J. Moissinac, and J. Dufourd. Representing 2d Cartoons using SVG. *Proceedings of SMIL Europe*, 2003.

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

[33] S. Ecma. Ecma-262 ecmascript language specification. *JavaScript Specif*, 16:1–252, 2009.

[34] L. Favre. A framework for modernizing non-mobile software: A model-driven engineering approach. In *Research Anthology on Recent Trends, Tools, and Implications of Computer Programming*, pages 320–345. IGI Global, 2021.

[35] L. M. Favre. Migrating software towards mobile technologies. In *Encyclopedia of Information Science and Technology, Fifth Edition*, pages 887–903. IGI Global, 2021.

[36] L. M. Favre. Non-mobile software modernization in accordance with the principles of model-driven engineering. In *IoT Protocols and Applications for Improving Industry, Environment, and Society*, pages 29–60. IGI Global, 2021.

[37] J. Ferraiolo, F. Jun, and D. Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington, 2000.

[38] M. Fiadotau. Growing old on newgrounds: The hopes and quandaries of flash game preservation. *First Monday*, 2020.

[39] S. Fulton and J. Fulton. *HTML5 canvas: native interactivity and animation for the web*. " O'Reilly Media, Inc.", 2013.

[40] A. Gawecki and F. Wienberg. Enterprise javascript with jangaroo. 2011.

[41] A. Gawecki and F. Wienberg. Enterprise javascript with jangaroo: using actionscript 3 for javascript programming in the large. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, pages 33–38, 2011.

[42] T. Hayakawa, S. Hasegawa, S. Yoshika, and T. Hikita. Maintaining web applications by translating among different ria technologies. *GSTF Journal on Computing (JoC)*, 2(1), 2014.

[43] J. Hoivik. From flash to html5: making a mobile web application for library with jquery mobile. In *E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher*

*Education*, pages 140–145. Association for the Advancement of Computing in Education (AACE), 2013.

[44] M. Hughes. Exploring open source flash: Whats available. In *The Essential Guide to Open Source Flash Development*, pages 7–17. Springer, 2008.

[45] A. Iakovenko-Marinitch. A haxe implementation of essential opengl examples using duell tool. 2016.

[46] 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.

[47] A. S. Incorporated. May 2007. URL `http://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf`.

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

[49] S. Jobs. Thoughts on flash. *Apple, Inc*, 2010.

[50] G. Keizer. Faq: How apple, google, microsoft and mozilla will eliminate adobe flash. *ComputerWorld, July*, 31, 2017.

[51] D. Leopoldseder. *Graal AOT JS: A Java to JavaScript Compiler*. PhD thesis, Universität Linz, 2015.

[52] D. Leopoldseder, L. Stadler, C. Wimmer, and H. Mössenböck. Java-to-javascript translation via structured control flow reconstruction of compiler ir. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, page 91103, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336901. doi: 10.1145/2816707.2816715. URL `https://doi.org/10.1145/2816707.2816715`.

[53] J. Li. *Can ActionScript find a place in introductory programming language: a systematic literature review*. PhD thesis, School of Computer Science and Statistics Can ActionScript find a place in , 2021.

[54] B. Manderscheid. Getting to know createjs. In *Beginning HTML5 Games with CreateJS*, pages 1–14. Springer, 2014.

[55] L. Martinez, C. Pereira, and L. Favre. Migrating c/c++ software to mobile platforms in the adm context. 2017.

[56] C. McAnlis, P. Lubbers, B. Jones, D. Tebbs, A. Manzur, S. Bennett, F. dErfurth, B. Garcia, S. Lin, I. Popelyshev, et al. Importing flash assets. In *HTML5 Game Development Insights*, pages 99–103. Springer, 2014.

[57] M. Papadrakakis, V. Papadopoulos, G. Stefanou, and V. Plevris. Modernizing software in science and engineering: From c/c++ applications to mobile platforms.

[58] T. Parisi. *WebGL: up and running*. " O'Reilly Media, Inc.", 2012.

[59] H. A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Springer Science & Business Media, 2012.

[60] J. A. Pereira, S. Sanz, I. Perurena, J. Gutiérrez, and I. Luengo. An experience migrating a cairngorm based rich internet application from flex to html5.

[61] S. Probets, J. Mong, D. Evans, and D. Brailsford. Vector graphics: from PostScript and Flash to SVG. In *Proceedings of the 2001 ACM Symposium on Document engineering*, pages 135–143. ACM, 2001.

[62] M. Ruottu. Tools and technologies for interactive elements and svg animations in html5-based e-learning. 2016.

[63] M. Rusnák. Software transition from flash to html5.

[64] M. I. Santally, R. K. Sungkur, N. Sooltangos, P. Mounier, and R. Mira-Bahadoor. Flash to html5 adaptations: A case-study for uptoten. com. In *2016 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pages 375–380. IEEE, 2016.

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

[66] M. T. Sheeba and M. S. H. Begum. Comparative study of developing interactive multimedia applications using adobe flash and html/css. *International Journal of Advanced Research in Computer Science and Electronics Engineering*, 7(5):1–6, 2018.

[67] M. Sikora. *Dart Essentials*. Packt Publishing Ltd, 2015.

[68] S. Stepasyuk and Y. Paunov. Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages. 2015.

[69] D. Štrekelj, H. Leventić, and I. Galić. Performance overhead of haxe programming language for cross-platform game development. *International journal of electrical and computer engineering systems*, 6(1):9–13, 2015.

[70] V. Štuikys and R. Damaševičius. Taxonomy of the program transformation processes. *Information Technology & Control*, (1):22, 2002.

[71] S. Swain, L. M. S., V. Choppella, and Y. Reddy. Model driven approach for virtual lab authoring - chemical sciences labs. In *2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT)*, pages 241–243, 2018. doi: 10.1109/ICALT.2018.00062.

[72] D. Syme, A. Granicz, and A. Cisternino. Building mobile web applications. In *Expert F# 3.0*, pages 391–426. Springer, 2012.

[73] V. VESELÁ, M. Krbeček, and Z. Prokopová. Comparative analysis of web animation creation methods. In *Recent Advances in Electrical Engineering and Educational Technologies: Proceedings of the 2nd International Conference on Systems, Control and Informatics (SCI 2014)*, pages 165–168, 2014.

[74] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[75] M. Ward. *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1989.

[76] R. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988. doi: 10.1109/32.7629.