The design and implementation of Google Swiffy: a Flash to HTML5 converter

Pieter Albertus Mathijs Senster

The design and implementation of Google Swiffy: a Flash to HTML5 converter

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Pieter Albertus Mathijs Senster
born in Den Helder, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Google UK Ltd
Belgrave House 76
Buckingham Palace Road
London, United Kingdom
www.google.com

The design and implementation of Google Swiffy: a Flash to HTML5 converter

Author: Pieter Albertus Mathijs Senster
Student id: [redacted]
Email: psenster@google.com

Abstract

As the web shifts towards mobile devices without support for Adobe Flash, developers need to use new technologies to bring the type of animated, interactive content they used to develop in Flash to those mobile devices. In this thesis, we present the design and implementation of a tool that eases this transition by automatically converting Flash to HTML5.

We propose a new type of transformation using a server-side compiler and client-side interpreter that benefits the performance, file size overhead and interoperability of the conversion.

The converter, named Swiffy, is evaluated on a dataset of Flash advertisements by measuring the percentage of files that are fully supported, the accuracy of the conversion and the performance of the output on desktop browsers and mobile devices.

Swiffy provides Flash to HTML5 conversion in Google AdWords and is available for anyone to use as an extension to Adobe Flash Professional or using an online conversion service. Since the public release, millions of SWF files have been converted and Swiffy is now used across the Internet.

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Company supervisor:	Dr. S. Spence, Google
Committee member:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee member:	Prof. Dr. K.G. Langendoen, Faculty EEMCS, TU Delft

Preface

In July 2010, I started an internship at Google London to explore how Flash animations could be displayed on mobile devices. This internship eventually resulted in Swiffy: a Flash to HTML5 converter. I was given the opportunity to continue at Google right after the internship as the technical lead of a new team continuing development on Swiffy. At that point, I approached Eelco Visser with the proposal for this thesis.

This thesis would not have been possible without the support of my many great colleagues at Google. I especially want to thank Branimir, Esteban, Graeme, Marcel, Ot and Stephen for their contributions and feedback on drafts of this thesis. Many thanks to my supervisor Eelco Visser for his support throughout the project, and for his insights in our discussions on Swiffy. Finally, I would like to thank my parents, friends and family for their support.

Pieter Senster
London, United Kingdom
December, 2012

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1

2 Analysis 3
- 2.1 Known approaches . . . 3
- 2.2 Requirements . . . 4
- 2.3 Research questions . . . 5

3 Background 7
- 3.1 The SWF file format . . . 7
- 3.2 Related web standards . . . 9

4 Architecture 13
- 4.1 Design decisions . . . 13
- 4.2 Architectural overview . . . 17
- 4.3 Server-side compiler . . . 17
- 4.4 Client-side interpreter . . . 21

5 Implementation details 25
- 5.1 Transformation of vector graphics . . . 25
- 5.2 Transformation of fonts and text . . . 31
- 5.3 Context for errors and warnings . . . 32

6 ActionScript **37**
6.1 History . . . 37
6.2 Interpreting ActionScript in JavaScript . . . 38
6.3 Compiling ActionScript bytecode to JavaScript . . . 41
6.4 ActionScript runtime libraries . . . 41

7 Evaluation methodology **43**
7.1 Metrics . . . 43
7.2 Dataset . . . 44
7.3 Evaluating performance . . . 45

8 Experimental results **47**
8.1 Coverage . . . 47
8.2 Accuracy . . . 48
8.3 File size overhead . . . 49
8.4 Performance . . . 50

9 Applications **53**
9.1 Flash to HTML5 conversion in Google AdWords . . . 53
9.2 Extension for Adobe Flash Professional . . . 54
9.3 Online conversion service . . . 54

10 Conclusions and future work **57**
10.1 Summary and conclusions . . . 57
10.2 Future work . . . 58

Bibliography **61**

List of Figures

4.1 System architecture. 17
4.2 Overview of the server-side compiler. 18
4.3 Overview of the top-level objects created by JSwiff. 19
4.4 Example of Swiffy bytecode serialised as a JSON object. 20
4.5 Overview of the client-side interpreter. 22
4.6 Correspondence between scene graph and renderers. 23

5.1 Example shape to illustrate path splitting. 26
5.2 Representation of Figure 5.1(a) in a SWF file. 26
5.3 Illustrated example of splitting a SWF shape in SVG paths. 28
5.4 Phase one of the path splitting algorithm. 29
5.5 Phase two of the path splitting algorithm. 29
5.6 Representation of a shape in the JSON object. 30
5.7 SVG generated to display the bytecode from Figure 5.6. 31
5.8 SVG generated to display static text using an embedded font. 31
5.9 SVG generated to display dynamic text using an embedded font. 32
5.10 Reference graph for a simple SWF file. 33
5.11 Reference graph for a SWF file containing a sprite. 35

6.1 Example of ActionScript bytecode as represented in the JSON object. 38
6.2 Mapping from instruction names to the numbers used in the JSON object. . . . 38
6.3 Translation of the JSON representation into function calls. 39
6.4 Main loop of the interpreter. 39
6.5 Implementation of the `add`, `trace` and `push_value` instructions. 40
6.6 Implementation of the `if` instruction. 40

8.1 Comparison of input and output file size. 49

9.1 Flash to HTML5 conversion in Google AdWords. 53
9.2 Swiffy Extension for Adobe Flash Professional. 54
9.3 Conversion feedback for the Swiffy Extension for Flash Professional. 54

9.4 Swiffy online conversion service. 55
9.5 Swiffy combined with other HTML5 techniques. 55

List of Tables

3.1 Example of a SWF file containing a single frame. 8
3.2 Example of a SWF file using animation and ActionScript. 8

7.1 ActionScript versions used in the ads dataset. 44
7.2 SWF versions used in the ads dataset. 44

8.1 Coverage of Swiffy on the ads dataset. 47
8.2 Coverage of Swiffy on the ads dataset per major browser. 48
8.3 Accuracy of Swiffy on the ads dataset. 48
8.4 Performance of Swiffy output on desktop browsers. 50
8.5 Performance of Swiffy output on various mobile devices. 51

Chapter 1

Introduction

When Macromedia Flash (now known as Adobe Flash) was released in 1996, it provided new functionality to web designers such as support for vector graphics, animation on a timeline and, most importantly, an editor that could be used by designers to create those graphics and animations. For more than ten years, web standards and browser implementations would not be able to match the functionality of this plugin-based technology.

In recent years, the rise in usage of mobile devices has changed the landscape. The Flash player is not available on any of the major mobile platforms, while web standards and their implementations have caught up. This gives an incentive to developers and designers to develop their content using those web standards. However, this is not always convenient. Tooling support for web standards is immature compared to tooling for Flash, and many designers only have experience in developing for Flash, or have existing assets in Flash that need to be reused.

To ease the transition for developers, we have developed a tool that automatically converts Flash to HTML5. The developer can then deliver the HTML5 version to modern browsers, including those on mobile devices, while the Flash version can still be used on older browsers.

In this thesis we present the design and implementation of this tool, which we have called Swiffy (a play on the pronunciation of the file format it operates on: SWF or "swiff" files).

This thesis is structured as follows. In the following chapter, we analyse the problem and present our research questions. The technologies most relevant to this thesis, the SWF file format and several web standards, will be introduced in Chapter 3. We present the architecture of Swiffy in Chapter 4, followed by the details of the implementation in Chapter 5. The handling of ActionScript is discussed in Chapter 6.

In Chapter 7, we describe the evaluation methodology used to validate Swiffy against the requirements. The experimental results using this methodology are then presented in Chapter 8. Several real-world applications of Swiffy are presented in Chapter 9, such as the use of Swiffy in Google AdWords. Finally, in Chapter 10, we draw conclusions and provide directions for future work.

Chapter 2

Analysis

Our goal in this thesis is to design and implement a Flash to HTML5 converter. We do not attempt to support all application domains of Flash, which nowadays includes 3D gaming and rich internet applications. Instead, we will focus on the type of content used in online display advertising, sometimes referred to as *banner ads*.

We describe the known approaches for Flash conversion in the first section of this chapter. In the following two sections, we specify the requirements and research questions that guide the design decisions and architecture for our Flash to HTML5 converter.

2.1 Known approaches

A comparison between SWF and SVG was first published in 2001 by Probets et al. [26]. Their paper describes how SWF represents shapes and animation, and how they can be converted to SVG. There is no discussion of more advanced features, such as how interactivity and ActionScript can be converted.

In 2003, Concolato et al. presented a technique to represent 2D cartoons using SVG and SMIL [5], and a tool that converts Flash files into SVG. Although the tool only supported a limited set of Flash features, their research demonstrated that a simple frame-based Flash animation can be represented using SMIL. Concolato later described a mechanism to structure SVG documents for SWF-like streaming [4].

Between 2010 and 2012, five tools were released that convert Flash to HTML5: Gordon, Smokescreen, Mozilla Shumway, Adobe Wallaby and the Adobe Toolkit for CreateJS. Gordon [27], Smokescreen and Mozilla Shumway are SWF players written in JavaScript. They load, parse and render the binary SWF file in the browser using JavaScript.

Smokescreen comes with basic support for the core ActionScript 2.0 language, but the ActionScript runtime libraries are not supported. Gordon does not support ActionScript at all. Mozilla Shumway has basic support for all ActionScript versions and is the only SWF player in JavaScript actively being developed as of 2012.

Wallaby is an experimental tool developed by Adobe to convert FLA (the source format of a SWF file) files to a combination of HTML, SVG and CSS. Since the source file is

needed, the tool is mainly designed to be used by content developers who would like to reuse assets from a FLA file in HTML. Wallaby does not support ActionScript.

The Toolkit for CreateJS is an extension to Adobe Flash Professional, which allows developers to convert SWF files into JavaScript code. The converted code requires the CreateJS JavaScript libraries to run. CreateJS uses the Canvas element for rendering and does not support ActionScript.

2.2 Requirements

In this section we enumerate the requirements for the converter developed as part of this thesis. Each requirement will be followed by the reasoning behind it.

Requirement 1: The converter should operate on SWF files.

Our converter needs to operate in environments where only the SWF file is available. The user who uploads the SWF file does not always have access to the original FLA file. Furthermore, we want to be able to convert existing content as well. Existing tools such as Adobe Wallaby and Adobe Toolkit for CreateJS use the FLA as the source of the compilation.

Requirement 2: The output should be supported by all major modern browsers.

The conversion output should function on all major modern browsers with HTML5 support, which we define to be the most recent versions of Google Chrome, Safari, Mozilla Firefox and Internet Explorer.

Requirement 3: The Gzip-compressed file size overhead should be less than 50%.

Swiffy output is expected to work well on mobile devices, where bandwidth is typically more constrained than on desktops. It may not be possible to achieve the same file size efficiency as the binary SWF file format, but the converted output should be less than 50% larger on average.

Requirement 4: Rendering performance should be acceptable on recent mobile devices.

To work well on mobile devices the output should not only be functionally correct, but the animation is expected to run smoothly as well. We will evaluate the rendering performance on three popular mobile devices released in 2011: the iPhone 4S, the iPad 2 and the Samsung Galaxy Nexus. In Section 7.3, we will present our definition of acceptable performance.

Requirement 5: The user should receive accurate feedback.

Not every SWF feature will be supported by Swiffy, either due to incompatibilities with web standards or limitations of Swiffy itself. In those cases, the user should receive an error or warning that accurately describes the source of the problem.

Requirement 6: The conversion process should be accurate: at least 95% of files without errors or warnings should have no defects.

In the intended use case for Swiffy the developer or advertiser will assess the quality of the conversion output. 100% conversion accuracy is therefore not required, especially for defects that are consistent across browsers and platforms. However defects might not always be immediately obvious, and the user may lose confidence in Swiffy if it fails to raise a warning about defects. We expect 95% of files for which no errors or warnings are raised by Swiffy to work correctly (functionally and visually) on all supported platforms.

2.3 Research questions

These requirements lead us to several research questions which we cover in this thesis. Since our ultimate goal is to design and implement a converter from SWF files to HTML5, our main research question is:

How can SWF files be transformed into HTML5?

We will answer the following secondary research questions that assist us in achieving the main goal of this thesis:

- What is the best architecture for a SWF to HTML5 converter?
- Can the type of graphics and animation used in Flash be described declaratively using HTML5?
- How can the similarities between ActionScript and JavaScript be used by the converter?
- Which web technologies are suitable as the target of the conversion?
- How can HTML5 animations be represented efficiently in order to limit the file size overhead?

Chapter 3

Background

In this chapter we discuss the source and target technologies of the converter. First, we discuss the SWF file format in Section 3.1, followed by a discussion of HTML5 and other relevant web standards in Section 3.2.

3.1 The SWF file format

The SWF file format is the binary format used to distribute Flash animations on the Internet. Since 2008 the specification of the format has been publicly available [19]. A variety of tools exist to generate SWF files, most of them originating from Adobe. The most common way of generating SWF files is to create them in an authoring tool called Adobe Flash Professional.

A plugin is needed to play a SWF file in a browser, although some browser vendors bundle this plugin with their installation. The traditional plugin is the Adobe Flash Player, which is available free of charge for a variety of platforms. Several open-source projects aim to build SWF-compatible players, such as Gnash [13] and Lightspark [25].

A SWF file consists of an ordered list of structured data blocks known as *tags* in the specification. Those tags define the graphics, animation and interactivity of a SWF file. On a high level, a tag is either a *definition* tag or a *control* tag. Definition tags will not be rendered directly, but they define a shape, image, sound or other object for later use. Each definition tag has an identifier that allows the definition to be referenced by other tags. A control tag results in an instance of a definition being placed, modified or removed from the screen. For example, the ShowFrame control tag defines where the definition of one frame ends and a new one begins.

Interactivity may be added to a SWF file using ActionScript, which can be included in the file in various formats. For ActionScript 1.0 and 2.0, the bytecode instructions may reside in button definitions or action tags such as the DoAction and DoInitAction tags. ActionScript 3.0 bytecode can only be included in a DoABC tag (ABC stands for ActionScript Byte Code). Further background on ActionScript will be given in Chapter 6.

Tag name	Data
Header	
DefineShape	id: 1, [vector data]
PlaceObject	id: 1, depth: 1, transform: (10, 10)
ShowFrame	
End	

Table 3.1: Example of a SWF file containing a single frame.

SWF file structure example

The structure of a simple SWF file that draws a shape on the screen is shown in Table 3.1. A SWF file starts with a header that specifies the SWF version and properties of the animation such as the frame rate. The header is followed by a list of tags. In this example the first tag is a DefineShape tag, which contains definitions of fill and line styles and vector data. A PlaceObject control tag places this definition on the screen at a certain position by specifying a transformation matrix. The z-ordering of instances is defined using the *depth* value, where the instance with the highest depth is drawn on top of instances at lower depths. A ShowFrame tag indicates that the definition of the frame is complete and that the frame can be rendered.

Tag name	Data
Header	
DefineShape	id: 1, [vector data]
PlaceObject	id: 1, depth: 3, transform: (10, 10)
PlaceObject	id: 1, depth: 2, transform: (20, 10)
ShowFrame	
PlaceObject	depth: 2, transform: (20, 20)
DoAction	ActionStop
ShowFrame	
End	

Table 3.2: Example of a SWF file using animation and ActionScript.

An extension of this example is shown in Table 3.2, which places the shape on the screen twice and animates one instance. The PlaceObject tag in the second frame modifies the current stage. Despite its name, the tag does not create an instance in this case, but modifies an existing instance. The instance can be uniquely identified by the depth it is placed on.

The following tag is a DoAction tag, which defines compiled ActionScript 2.0 code as a sequence of bytecode instructions. In this example the only instruction is the ActionStop instruction, which prevents the animation from looping.

A reusable timeline can be defined using the DefineSprite definition tag, which contains a list of control tags to place other definitions on a timeline. This concept is called a *MovieClip* in Adobe Flash Professional. MovieClips are the building blocks of the *scene graph*: a collection of graphical objects organised as a tree.

3.2 Related web standards

HTML5 is a proposed standard for structuring information on the Internet, developed by the Web Hypertext Application Technology Working Group (WHATWG) and the W3C. It introduces several new features with respect to HTML 4.01 and XHTML 1.1, such as the `<video>` and `<canvas>` elements and new APIs. A wide variety of related web technologies are generally included in the term HTML5, such as Scalable Vector Graphics, Web Workers and CSS3. Although technically incorrect, we will follow common usage and use HTML5 as an umbrella term for these web standards as well. We will briefly discuss the technologies most relevant to Swiffy: Canvas, SVG, SMIL and CSS animation.

3.2.1 Canvas

The Canvas element offers an HTML5 API to draw onto a 2D bitmap in the browser. It allows a developer to have exact control over every pixel, and provides convenience functions to draw lines and basic shapes.

The following code snippet illustrates how a red and a green square can be drawn on the screen using Canvas.

```
1 var canvas = document.createElement("canvas");
2 var context = canvas.getContext("2d");
3 context.fillStyle = "rgb(255, 0, 0)";
4 context.fillRect (0, 0, 20, 20);
5
6 context.fillStyle = "rgb(0, 255, 0)";
7 context.fillRect (20, 0, 20, 20);
```

This example demonstrates that Canvas does not expose a scene graph: there is no way to establish a hierarchy among graphical objects. Furthermore, it is not possible to attach event handlers to parts of the scene. To achieve interactivity, developers using the Canvas element will have to write custom hit testing code.

3.2.2 Scalable Vector Graphics

The Scalable Vector Graphics (SVG) language is an XML-based language to define vector graphics on the web. In contrast to the Canvas element, SVG maintains a scene graph. SVG is a declarative language: Instead of instructing the renderer to write specific pixels into a buffer, the developer specifies the end result in terms of graphical primitives. The SVG renderer decides when and how to render the scene, taking the effort of invalidating and repainting out of the developer's hands.

The following code snippet draws a red and green square in SVG.

```
<svg>
  <rect x="0" y="0" width="20" height="20" fill="red" />
  <rect x="20" y="0" width="20" height="20" fill="green" />
</svg>
```

An SVG DOM tree can be created and modified using JavaScript. When the DOM is modified, the browser will invalidate the affected regions and queue a repaint event. The following JavaScript code snippet generates the same SVG as the snippet of XML shown above.

```
var svgNode = document.createElementNS(
    "http://www.w3.org/2000/svg", "svg");
var redRect = document.createElementNS(
    "http://www.w3.org/2000/svg", "rect");
redRect.setAttribute("x", "0");
redRect.setAttribute("y", "0");
redRect.setAttribute("width", "20");
redRect.setAttribute("height", "20");
redRect.setAttribute("fill", "red");
svgNode.appendChild(redRect);

var greenRect = document.createElementNS(
    "http://www.w3.org/2000/svg", "rect");
greenRect.setAttribute("x", "20");
greenRect.setAttribute("y", "0");
greenRect.setAttribute("width", "20");
greenRect.setAttribute("height", "20");
greenRect.setAttribute("fill", "green");
svgNode.appendChild(greenRect);
```

3.2.3 Synchronized Multimedia Integration Language

Synchronized Multimedia Integration Language (SMIL) is a markup language originally designed to create multimedia presentations for the web [17]. SVG is one of the types of content that can be animated using SMIL. For example, the SMIL markup embedded in the following SVG transitions the colour of a rectangle from red to green in three seconds.

```
<svg>
  <rect x="0" y="0" width="20" height="20" fill="red">
     <animate attributeName="fill" attributeType="CSS"
          from="red" to="green" begin="0s" dur="3s" />
  </rect>
</svg>
```

SMIL allows the developer to specify animation declaratively, as distinct from the more commonly used imperative style of web animation using JavaScript.

3.2.4 CSS animation

CSS animation is a working draft for an extension to CSS that allows developers to specify animation declaratively. In the following example, a CSS rule is defined that applies an

animation with the name `example-transition` to an element with the identifier `example`. This transition is defined using the `@keyframes` keyword and specifies a transition from a red fill to a green fill, similar to the SMIL example above.

```
<svg>
  <rect id="example" x="0" y="0" width="20" height="20" fill="red" />
</svg>

<style type="text/css">
#example {
  animation-duration: 3s;
  animation-name: example-transition;
}

@keyframes example-transition {
  from {
    fill: red;
  }
  to {
    fill: green;
  }
}
</style>
```

CSS animation is a single animation model for both HTML and SVG content with a syntax that will be familiar to most web developers. Animations defined using CSS are hardware-accelerated in most modern browsers.

Chapter 4

Architecture

In this chapter we describe the architecture of Swiffy. First, we discuss the design decisions that determined the architecture. We then give the architectural overview of our chosen design: A server-side compiler and client-side interpreter working together using a custom bytecode to achieve the transformation from SWF to HTML5. Finally we describe the compiler and interpreter in more detail in Section 4.3 and Section 4.4, respectively.

4.1 Design decisions

We present three design decisions that determined the architecture of Swiffy. First, we discuss whether the transformation should be executed server-side, client-side, or using a combination of both models. We propose a new model that uses an intermediate representation we refer to as Swiffy bytecode, improving upon existing Flash to HTML5 converters in several areas. In Section 4.1.2, we discuss the rendering and animation technology used to display the output in the browser. The programming paradigm most suitable for the transformation is discussed in Section 4.1.3 by characterising the scope, direction and staging of the transformations.

4.1.1 Client versus server-side

Existing Flash to HTML5 converters use either a pure client-side or pure server-side model. Client-side converters such as Gordon and Smokescreen interpret the SWF file itself in JavaScript without pre-processing the file. In this model, the SWF file is required on the client alongside a SWF player written in JavaScript.

In pure server-side converters, the SWF or FLA file is only processed on the server. That server generates HTML, CSS, SVG and/or JavaScript code that does not require an interpreter. The most prominent example of this model is Adobe Wallaby.

We propose a hybrid of the two models, introducing a server-side compiler that transforms the SWF file into an efficient intermediate representation that can be conveniently handled by a client-side interpreter. This intermediate representation will be referred to as *Swiffy bytecode*, since it can be regarded as instruction code for graphics and animation that

will be interpreted client-side. The hybrid model based on this bytecode improves upon existing Flash to HTML5 converters in the following areas:

Performance In the pure client-side model, computationally expensive operations, such as transcoding images, sound and video, need to be executed on the client. In our model, these transformations are executed server-side to obtain a bytecode that can be efficiently handled on the client.

File size Code generated directly by SWF converters is generally significantly larger than the original SWF file, due to the verbosity of HTML and SVG. By serialising Swiffy bytecode using an efficient representation such as JavaScript Object Notation (JSON), we are able to achieve a file size ratio that is well within our requirements.

Run-time rendering decisions A pure server-side converter generates code that targets a single rendering technology such as SVG or Canvas. Our client-side interpreter can make rendering decisions at run-time based on the objects being drawn and characteristics of the client platform. This allows it, for example, to combine the strengths of SVG and Canvas to achieve efficient rendering.

Interoperability The bytecode can be processed or rendered using alternative client-side interpreters, for example one that uses Google Native Client (NaCl). Furthermore, other converters could target the same bytecode. For example, one could develop a compiler from Microsoft Silverlight files to Swiffy bytecode.

Streaming The SWF file format allows for streaming of the file by ordering the tags in the byte stream per frame. This allows the player to start rendering frame n while still fetching $n+1$ from the network. Swiffy bytecode preserves this behaviour. Once the client-side interpreter is loaded, it can start streaming the bytecode using `XMLHttpRequest`.

A disadvantage of the bytecode is the lack of interoperability with existing developer tools and workflows. Code generated by a pure server-side converter can be imported into HTML editing tools for further development. To modify the output of the Swiffy compiler, the developer needs to be familiar with Swiffy bytecode, which may not accommodate the developer's needs.

4.1.2 Rendering technology

In Chapter 3.2 we introduced the major styles of rendering and animation available on the web: Canvas, SVG, CSS animation and SMIL. In this section we will decide upon the technology most suitable for our application domain.

Our design goal is to use a technology that allows us to delegate as much work and responsibility to the browser as possible. This simplifies the client-side interpreter and benefits the performance of the output. For example, a commonly used feature in Flash

is the application of filter effects. If such an expensive operation were implemented using JavaScript, the browser would be unable to use optimised (and potentially hardware-accelerated) code to implement the effect. On the other hand, if a declarative language was used to describe the intended effect of the filter (a drop shadow, for example), the browser would be able to do this.

The similarities between the SVG language and the SWF file format are extensive: Both are vector graphics based, use a hierarchical tree of objects (a scene graph) and both provide similar primitives for filter effects, clipping and masking. In areas where SVG does not provide the required functionality it can be complemented with other web technologies using an SVG feature known as *foreign objects*.

The declarative nature of SVG has advantages in terms of accessibility as well. For example, a screen reader is able to read text for disabled users when text is declared using an SVG text element. If Canvas were used, a screen reader would be unable to access this text.

The benefits of a declarative language leads us to use SVG as the main rendering technology.

Animation The choice of SVG for rendering leaves us with several options to animate the SVG content. At first sight, both CSS animation and SMIL, introduced in Chapter 3.2, adhere to our design goal of using a declarative representation when possible. However, both technologies provide insufficient control over the animation when support for ActionScript scripting is required. For example, although the concept of keyframes exists in CSS animation, it does not provide a mechanism to synchronise the JavaScript code to those keyframes or to modify the timeline from JavaScript, which is a basic feature required for SWF compatibility. Another limiting factor is that animation in the SWF file itself is not defined in terms of high-level transitions, but defines the position of every object at specific keyframes. Mapping these definitions back onto CSS or SMIL transitions is not always possible. Finally, these standards are not widely available: The CSS animation specification is still in working draft state, while SMIL is not implemented in the Internet Explorer browser.

We have therefore chosen to use JavaScript to animate SVG on the client. Although this imperative approach might be less efficient, the level of control it provides is required to match all SWF functionality.

4.1.3 Programming paradigm

To characterise the program transformation that takes place in the Swiffy compiler, one can describe the *scope*, *direction* and *staging* of the transformation [29]. This characterisation may influence the programming paradigm used and indicates where common transformation design patterns can be applied.

Scope The scope of a transformation indicates the effect of a single transformation step on the *source* and *target* of the transformation. Since the structure of SWF files is much simpler than a typical abstract syntax tree, the concept of scope is simpler as well. We define a *local*

scope to affect a single tag: a SWF tag in the source and a Swiffy bytecode instruction in the target. A transformation step has *global* scope when multiple tags are involved.

Most transformation steps of the Swiffy compiler have an even simpler special case of local scope: *1-to-1*. This transformation type transforms a single SWF tag into a single Swiffy bytecode instruction and preserves the order of that tag in the file. Transformation steps of this type include the transformation of the following SWF tags:

- PlaceObject, PlaceObject2 and PlaceObject3
- DefineShape, DefineShape2, DefineShape3 and DefineShape4
- DefineBitsJPEG2, DefineBitsJPEG3, DefineBitsLossless and DefineBitsLossless2
- DefineButton and DefineButton2
- DefineFont and DefineFont2

Most of these transformations transform several versions of the same tag in the SWF specification to one type of instruction in the Swiffy bytecode. An example of a new version of a tag is the PlaceObject2 tag, which extends the PlaceObject tag by introducing several new data fields. The PlaceObject instruction in Swiffy bytecode supports the superset of the functionality available in the various versions of the SWF PlaceObject tag. These transformation steps can be regarded as a normalisation of the source.

Sometimes, multiple SWF tags in the source can be combined to a single bytecode instruction, a *global-to-local* transformation step. Transformation steps of this type include the merging of:

- JPEGTables and DefineBits
- DefineButtonCxform and DefineButton

The DefineButtonCxform SWF tag applies a colour transformation to a button that is defined elsewhere in the SWF file by a DefineButton tag. Swiffy combines these tags and generates a single button definition instruction that incorporates the colour transformation. This makes the Swiffy bytecode simpler and simplifies the client-side interpreter.

Direction and staging The direction of a transformation can be either *source-driven* or *target-driven*. Most template languages are examples of target-driven transformations: The target is defined in terms of units that can be queried from the source. Most compilers on the other hand are source-driven: The output is generated using a traversal over the source.

Since Swiffy bytecode is generated by a traversal of the SWF tags in the source file, the Swiffy compiler is source-driven. Only a single-stage traversal of the input is required.

Given that only two simple types of transformations (1-to-1 and global-to-local) are performed, the direction is source-driven and the transformation is single-staged, Wijngaarden and Visser [29] indicate that an OO-language to execute the transformation can be suitable, without employing the more advanced program transformation tools which they recommend for more complex transformations.

4.2 Architectural overview

A SWF file is transformed into HTML5 using two main components: a server-side compiler and a client-side interpreter (Figure 4.1). 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` file.

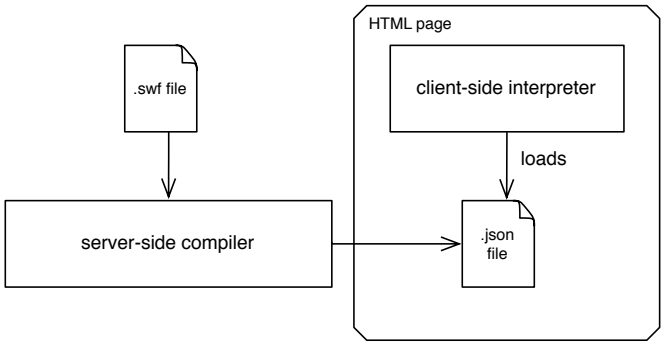


Figure 4.1: System architecture. A server-side compiler parses the SWF file and transforms it into Swiffy bytecode. This bytecode is interpreted on the client by an interpreter written in JavaScript.

When possible, computationally intensive transformations are executed in the server-side phase, allowing the client to do as little work as possible. In typical use, the server-side phase will be executed just once for a given file. The resulting JavaScript or HTML file can then be hosted in the same manner as the SWF file.

4.3 Server-side compiler

The server-side compiler uses various transformations and optimisations to convert the SWF file into a format that is convenient to handle on the client. The main phases within the server-side compiler are *parsing*, *tag transformation* and *serialisation* (Figure 4.2).

Traditionally, compilers consist of three main components: a front-end, middle-end and back-end [3]. The front-end is responsible for the creation of an intermediate representation (IR) from the source code, which usually involves lexical analysis, syntactic analysis and type checking. The IR can be optimised by the middle-end, resulting in optimised IR that is ready to be processed by the back-end. In the back-end, the IR is translated to the target language, which may be assembly language, a high-level language or any other type of language.

In our architecture, the parsing and tag transformation phase form the front-end of the compiler. The middle-end optimises the bytecode by merging duplicate elements and removing unused definitions. In Swiffy, the back-end is a serialisation of the bytecode as a JSON object. We will now discuss the three phases in more detail.

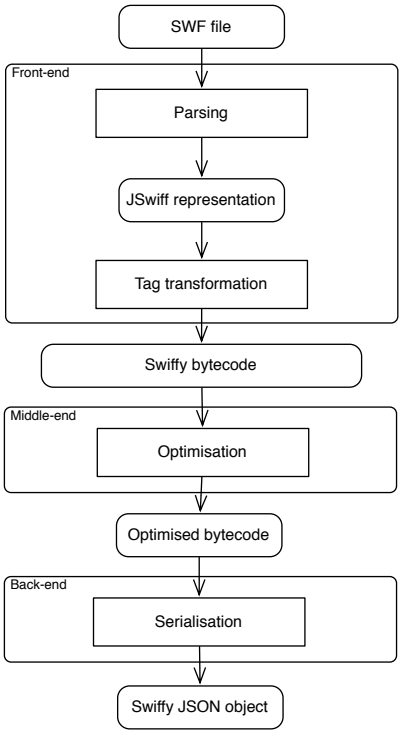


Figure 4.2: Overview of the server-side compiler.

Parsing In the parsing phase, an object representation is created from the binary SWF source file using the open-source Java library JSwiff [7]. JSwiff creates Java objects for each tag in the SWF file and creates hierarchies of tags where needed.

Figure 4.3 shows an overview of the top-level objects as created by JSwiff. A single SWF file is represented as a SWFDocument object, which is defined by the control and definition tags it contains.

Tag transformation The core work of the Swiffy compiler is performed in the tag transformation phase. In this phase, the SWF objects generated by JSwiff are converted into Swiffy objects. Although the SWF bytecode and Swiffy bytecode share many characteristics, the latter is simpler and shares more characteristics with the web standards that are targeted. The transformations in this phase are, for example:

- **Vector graphics** SWF files use a vector graphics format that is not compatible with SVG. This transformation is described in Section 5.1.
- **Font definitions** Embedded fonts are defined using an incompatible vector graphics format and need to be converted.
- **Images** Besides standard image formats such as JPEG, PNG and GIF, SWF files may contain several non-standard image formats.

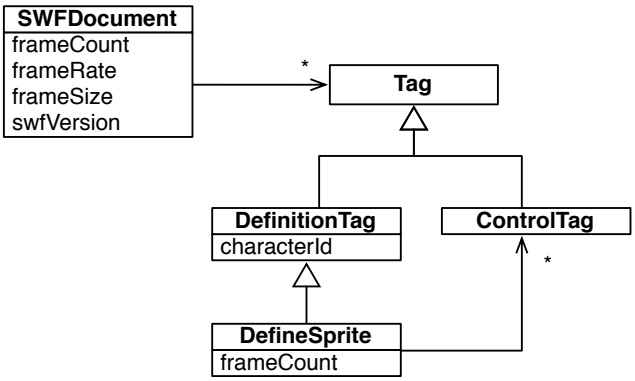


Figure 4.3: Overview of the top-level objects created by JSwiff.

- **Audio and video** SWF supports several audio and video codecs, whereas most browsers only have support for a subset of these codecs.

- **Historical SWF characteristics** The evolution of the SWF specification over the years is reflected in some of the tags in the specification. The Swiffy compiler can work around those characteristics to create a representation that is simpler than the original SWF file and easier to interpret.

- **Reordering of right-to-left text** Until version 10, SWF did not support right-to-left scripts such as Hebrew and Arabic. To work around this, Flash authors embedded text in such scripts using visual ordering of the characters. Web browsers and search engines expect logical ordering, so Swiffy attempts to reorder the characters to improve the accessibility and crawlability of the output.

Serialisation In the proposed architecture, Swiffy bytecode is interpreted in the browser. Since the bytecode must be stored in a format that can be parsed in a browser environment, Swiffy serialises it to JSON. JSON is a convenient way to distribute the bytecode, since modern browsers have native support for parsing JSON strings to JavaScript objects. Furthermore, using JSON allows Swiffy to support loading additional SWF files at run-time by fetching the JSON files using `XMLHttpRequest`.

Figure 4.4 shows the JSON representation of Swiffy bytecode that draws a red and green rectangle on the screen. Note that the comments have been added for clarity and are not generated by the compiler. The bytecode contains three instructions: a shape definition instruction defining a red and green rectangle, a place instruction that places the shape on the screen and a final instruction that marks the end of the frame. The final lines describe metadata such as the frame rate and dimensions of the animation.

The object in Figure 4.4 is pretty-printed for readability. In typical operation no whitespace is added by the compiler to achieve smaller files.

```
{
  "instructions": [
    {
      "type": 1, // This instruction is a shape definition.
      "id": 1, // Unique identifier of this definition.
      "fillstyles": [ // This shape defines two fill styles.
        {
          "type": 1, // Solid fillstyle.
          "color": 16711680 // The colour red as an ARGB value.
        },
        {
          "type": 1, // Solid fillstyle.
          "color": 65280 // The colour green as an ARGB value.
        }
      ],
      "paths": [ // This shape contains two paths.
        {
          "fill": 0, // Index of the fill style to use.
          "data": // Encoded path data.
              [0,2479,2679,1,360,0,1,0,560,1,2479,0,1,0,2679]
        },
        {
          "fill": 1, // Index of the fill style to use.
          "data": // Encoded path data.
              [0,2479,560,1,4598,0,1,0,2679,1,2479,0,1,0,560]
        }
      ]
    },
    {
      "type": 3, // This object places an instance on the screen.
      "id": 1, // Identifier of the definition to use.
      "matrix": [1,0,0,1,0,0], // A transformation matrix.
      "depth": 1 // Z-order of the instance.
    },
    {
      "type": 2 // Instruction to render a frame.
    }
  ],
  "backgroundColor": 16777215, // The colour white as an ARGB value.
  "frameWidth": 11000,
  "frameHeight": 8000,
  "frameCount": 1,
  "frameRate": 24
}
```

Figure 4.4: Example of Swiffy bytecode serialised as a JSON object.

Using a plain-text format like JSON comes at the cost of larger files compared to the binary SWF format. The JSON representation of the bytecode is essentially a key/value map where the keys (which are strings) are typically repeated many times in the file. Sending the data over HTTP using Gzip compression [8] is therefore essential. Gzip is one of the compression methods specified in the HTTP/1.1 specification [2] and is supported by most clients and servers, and yields a significant reduction in file size for the type of files the Swiffy compiler generates.

In the example shown in Figure 4.4, Gzip-compression of the file results in a 33% reduction in file size. This reduction is even more significant for actual files in which most JSON keys occur many times. An evaluation of the output file size and the effect of compression will be presented in Chapter 8.

4.3.1 Error reporting

Swiffy will not support every feature in the SWF specification or every API in the ActionScript runtime libraries. The requirements in Section 2.2 state that Swiffy should be able to give accurate feedback about the conversion process. That feedback will be given in the form of compile errors and warnings, which may arise during both the parsing and the tag transformation phase.

JSwiff is responsible for checking the validity of the input and will raise Java exceptions in case unexpected input is encountered. These exceptions should never occur in files created with Adobe Flash Professional, but may sometimes occur with files created with other editors.

In the tag transformation phase, warnings and errors may occur regarding unsupported SWF features and ActionScript runtime libraries. The distinction between an error and warning is made by estimating the impact of the message. When unsupported runtime libraries are used, the output is very likely to be functionally incorrect. We therefore classify those messages as errors. The mechanism to detect runtime library usage will be discussed in Section 6.4.1.

Other unsupported features will only result in a visual defect in the output. For example, blend modes (that define how two layers are blended) is a commonly used feature in SWF files, but is not yet available in SVG or CSS. This limitation is exposed to the user using a warning. Since the functionality is not impacted, the conversion might still be acceptable to the user.

In compilers that operate on source code, the context of an error or warning is typically the line number. Since such a context does not exist in a SWF file, we will discuss an alternative mechanism to provide context to compile messages in Section 5.3.

4.4 Client-side interpreter

The client-side interpreter is a JavaScript library that offers developers an API to place Swiffy animations in a web page. Figure 4.5 illustrates the four main components of the interpreter. The model-view-controller (MVC) pattern is used to achieve modularity between the components. The interpreter is the controller that loads the JSON object and builds and

controls a scene graph (the model). Furthermore, it passes blocks of ActionScript instructions to the ActionScript VM to be executed. The rendering engine builds and modifies an SVG DOM from the scene graph maintained by the interpreter. This modularity makes it possible to use a different rendering engine or different VM (ActionScript 2.0 versus ActionScript 3.0, for example) without affecting the other components.

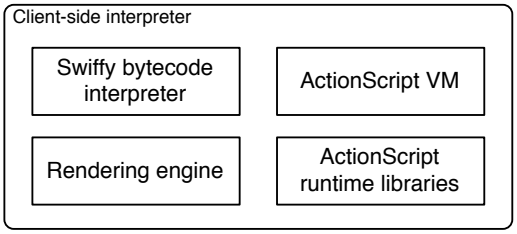


Figure 4.5: Overview of the client-side interpreter.

In typical operation the JavaScript file that contains the interpreter is hosted separately and linked from web pages using it, rather than being embedded in the page. This makes it possible to host the interpreter on a content delivery network (CDN) and allows it to be cached by clients.

The remainder of this section describes the bytecode interpreter and rendering engine in more detail. The ActionScript VM and ActionScript runtime libraries will be discussed in Section 6.

Bytecode interpreter The interpreter loads the JSON object and translates it into a sequence of actions to execute on every frame. Typical actions are the instantiation of a display object (a node in the scene graph), the removal of a display object or the execution of ActionScript code. Most of the knowledge about the semantics of the SWF file format are encapsulated in this component. For example, it implements the complex (undocumented) rules that determine how the contents of two frames should be merged when jumping backwards on the timeline.

Rendering engine The rendering engine renders the scene graph using SVG. It is completely decoupled from the other components of the system, and can therefore easily be replaced by a different renderer (such as Canvas or WebGL) or be reused in a different environment.

Several open-source web rendering engines exists, such as Raphaël for rendering using SVG and VML [23]. However, none of the rendering engines have enough functionality for Swiffy to support all SWF features. We therefore decided to implement a custom rendering engine.

A renderer object is created for every object in the scene graph. This way a hierarchy of renderers is created that has a one-to-one correspondence with the scene graph. Since SVG has a hierarchical DOM, there is a convenient correspondence between DisplayObjects,

their renderers and the elements in the DOM that represent them. Figure 4.6 is an example of a the typical correspondence for a small scene graph with three elements. This example illustrates a MovieClip object with two children: a Shape and a Text object. Specialised renderers are created for each of those objects using the same hierarchy. Subsequently, those renderers create a DOM hierarchy that preserves the same relationships using SVG groups (the `<g>` nodes).

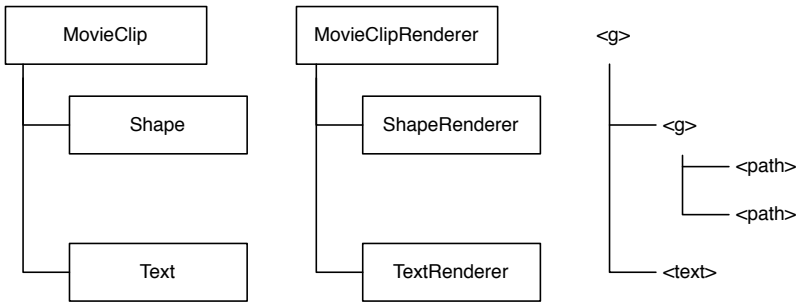


Figure 4.6: Typical correspondence between display objects in the scene graph, their renderers and the SVG DOM elements they own.

Chapter 5

Implementation details

In this chapter we describe the implementation of two transformations from Flash to HTML5. We explain the representation of vector graphics in the SWF format and provide the algorithms that are required to make it compatible with SVG. In Section 5.2, we show how SVG and HTML are combined to transform SWF fonts and text. Finally, we describe the construction of the reference graph: a graph used to provide the user context to compile errors and warnings.

5.1 Transformation of vector graphics

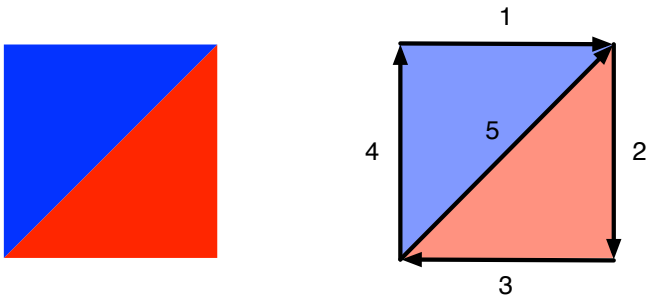
Vector data is used in the SWF file format to describe shapes (in the DefineShape, DefineShape2, DefineShape3 and DefineShape4 tags) and glyph outlines of fonts (in the DefineFont, DefineFont2 and DefineFont3 tags). Both classes of tags use the same vector format, which we will analyse in this section.

A shape is defined as a list of edges in the SWF vector format. Such an edge is either a straight line, a quadratic Bezier curve or an instruction to move the drawing position to a new location. In between the edge definitions, the fill style and line style properties may be changed, affecting all subsequently defined edges. In SVG a `<path>` node is used to draw a shape, which can only have a single fill style and line style. Therefore, a SWF shape may have to be split up into multiple SVG paths, where each path contains edges sharing the same fill style and line style.

A sequence of edges may be filled using a solid colour, a linear or radial gradient or a bitmap. To optimise the number of edges that are needed to define a path, SWF allows two types of fill styles for each edge: one for the left-hand side of the edge (when the edge is viewed as a vector) and one for the right-hand side, called *fillstyle0* and *fillstyle1* respectively. This way a single edge can be part of two adjacent subshapes, where other vector formats would need two edges. SVG supports only one fill style per path, so a transformation of the vector data is necessary.

Example Figure 5.1(a) is the example shape that we will use to discuss the transformation. Since SWF allows a single edge to define fill styles for the left and right hand side of the

vector, this shape can be defined using five edges, as illustrated in Figure 5.1(b). Edge 5 is shared between the two triangles and uses a fill style on both the left and right hand side.



(a) Visual appearance of the shape.

(b) One way the shape can be described in a SWF file by a series of edges. The labels indicate the order in which the edges are defined.

Figure 5.1: Example shape to illustrate path splitting.

Figure 5.2 shows how this shape is encoded in a SWF file. Each line describes what is called an *edge record* in SWF: either an actual line fragment or a *style change record*. Such a record is able to change the fill style and line style of subsequent edges, and can move the 'pen' without drawing a line. Note that all coordinates in the SWF file are relative.

```
fillstyle1: blue
line x: 10 (edge 1)
fillstyle1: red
line y: 10 (edge 2)
line x: -10 (edge 3)
fillstyle1: blue
line y: -10  (edge 4)
fillstyle0: blue, fillstyle1: red
move y: 10, line x: 10 y: -10 (edge 5)
```

Figure 5.2: Representation of Figure 5.1(a) in a SWF file. The edge numbers are not present in the SWF file, but have been added for clarity.

For this example, our transformation should generate two SVG paths: one describing the red triangle, and one describing the blue triangle.

5.1.1 Transformation to Swiffy bytecode

We propose an algorithm that maps SWF vector data onto path definitions that are compatible with SVG. This algorithm resolves the two main differences between SWF and SVG vectors graphics, as outlined in the previous paragraphs:

1. In SVG, all edges in a path share the same fill and line style, whereas in SWF the styles can be changed in between edges.

2. In SVG, a path has only one fill style, whereas in SWF each edge may have two fill styles.

The algorithm proceeds in two phases. In the first phase, the edges are placed in buckets of edges having the same fill style and buckets for edges having the same line style. A conversion to absolute coordinates is performed to make it easier to change the order the edges are in.

The buckets cannot be used directly to create SVG-compatible paths. In our example from Figure 5.1, edges 1, 4 and 5 would be in the 'blue' bucket, resulting in the path illustrated in Figure 5.3(b). Two of the edges use a right-hand side fill style, and one uses a left-hand side fill style, which causes the edges to have different directions.

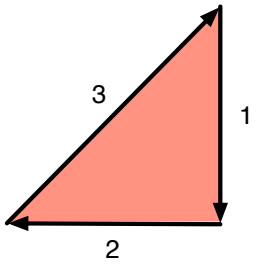
To make sure all edges are either clockwise or anticlockwise, we flip the endpoints of edges that define a left-hand side fill (*fillStyle0* in the SWF format). Edges that have both a left-hand and a right-hand fill style are duplicated: one flipped and one not. These edges might end up in different fill style buckets if the two fill styles were different. This phase is shown in Figure 5.4.

Now that we have a list of edges for each fill style, we are nearly ready to construct the corresponding paths. Because we might have flipped some edges, the endpoints of the edges might no longer line up for all edges belonging to a certain fill style or line style. In our example, this is illustrated in Figure 5.3(c). Therefore, for each fill style, we reorder the edges such that the endpoint of each edge is the start point of the next edge. Fill styles are traversed in the order they were defined in the SWF file, since the Flash player renders paths in that order. This phase is shown in Figure 5.5. The end result for the blue triangle in the example shape is displayed in Figure 5.3(d).

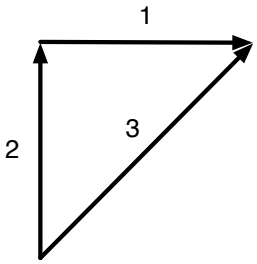
After this phase, the transformation completes by constructing paths from the edges in the line styles buckets. These do not need to be reordered since they were added in the original, connected, ordering specified in the SWF file. The combined set of paths for a shape now starts with paths that use a fill style, followed by paths that use a line style. This corresponds to the way shapes are rendered by the Flash player, in which line styles are always drawn on top of fills.

This procedure may result in more paths than necessary. A path using both a fill style and a line style in the SWF file will result in two paths after the transformation, one with only a fill style and the other with only a line style. This can be optimised by combining all such paths if they describe the same area.

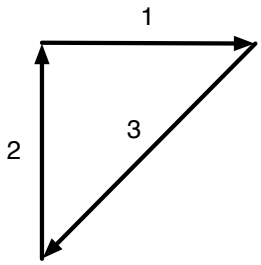
Splitting of the paths as outlined by our algorithm can be executed server-side by the compiler, reducing the work that needs to be done at run-time. However, this may also



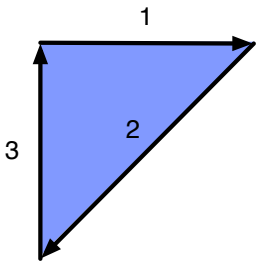
(a) The red triangle as it is constructed by ordering edges in buckets with the same fill style. This is a valid SVG path.



(b) The shape that would be constructed from the edges with a blue fill style. This path does not describe a shape that can be filled in SVG.



(c) The result of phase one of the path splitting algorithm. Edge 3 is flipped because it defined the blue fill style on the left hand side of the vector. This path still does not describe a shape that can be filled by SVG because the edges are out of order.



(d) The shape that result from phase two of the algorithm. The edges have been reordered, and now describe a valid SVG path.

Figure 5.3: Illustrated example of splitting a SWF shape in SVG paths.

```
1: Convert all edges to absolute coordinates
2: fillStyle0 ← none
3: fillStyle1 ← none
4: lineStyle ← none
5: for record in shape do
6:     if record is a style change then
7:         Update fillStyle0, fillStyle1 and lineStyle
8:     else
9:         if fillStyle0 is set then
10:            Flip the endpoints of the edge
11:            Add flipped edge to fillEdges[fillStyle0]
12:        if fillStyle1 is set then
13:            Add edge to fillEdges[fillStyle1]
14:        if lineStyle is set then
15:            Add edge to lineEdges[lineStyle]
```

Figure 5.4: Phase one of the path splitting algorithm. Edges are placed in buckets of edges that share the same fill style or line style.

```
1: Sort fill style buckets by the order the fill styles are defined in the SWF file
2: for each fillStyle do
3:     path ← empty path
4:     remaining ← all edges with the current fillStyle
5:     current ← first edge in remaining
6:     while remaining is not empty do
7:         Add current to path
8:         Remove current from remaining
9:         if remaining is not empty then
10:            if an edge in remaining connects to current then
11:                current ← first such connecting edge
12:            else
13:                current ← first edge in remaining
```

Figure 5.5: Phase two of the path splitting algorithm. Edges with the same fill style are ordered such that connecting edges are adjacent.

result in increased file size of the Swiffy bytecode because our representation may contain more edges.

5.1.2 Representation in the JSON object

The path data is stored in the JSON object as an array of numbers. Each segment of the array starts with an integer specifying the type of the segment, either a move instruction (0), a line to a point (1) or a quadratic Bezier curve (2). The type of the segment determines how many coordinate pairs follow in the array. For move and line commands one coordinate pair follows. A quadratic Bezier curve needs two pairs: the control and anchor points of the curve. This representation of path data is straightforward to convert to SVG and can be conveniently handled when transforming paths at run-time.

```
{
  "type": 1,
  "id": 1,
  "fillstyles": [
    {
      "type": 1,
      "color": 255
    },
    {
      "type": 1,
      "color": 16711680
    }
  ],
  "paths": [
    {
      "fill": 0,
      "data": [1,10,0,1,0,10,1,0,0]
    },
    {
      "fill": 1,
      "data": [0,10,0,1,10,10,1,0,10,1,10,0]
    }
  ]
}
```

Figure 5.6: Representation of a shape in the JSON object.

The JSON representation of the shape displayed in Figure 5.1(a) is given in Figure 5.6. It starts with a definition of the type and unique identifier of the tag, followed by the fill style definitions for the colours blue and red. The colour values are represented using an ARGB integer representation.

5.1.3 Transformation to SVG

Since shapes are stored in an SVG-compatible format in the Swiffy bytecode, the final transformation to SVG `<path>` elements is straightforward. SVG uses strings to define

path data, in which the character `M` denotes a move instruction, `L` a line to a point and `Q` a quadratic Bezier curve. Converting the array of numbers in the bytecode to SVG is a simple concatenation of the numbers, replacing the numeric segment types by the character used to describe the type in SVG. The SVG resulting from the bytecode in Figure 5.6 is shown in 5.7. An SVG group node, denoted by a `<g>` tag, is used to group the two paths to allow them to be handled as one object.

```
<g>
  <path d="L 10 0 L 0 10 L 0 0" fill="rgb(0, 0, 255)"></path>
  <path d="M 10 0 L 10 10 L 0 10 L 10 0" fill="rgb(255, 0, 0)"></path>
</g>
```

Figure 5.7: SVG generated to display the bytecode from Figure 5.6.

5.2 Transformation of fonts and text

Text in SWF files is classified as either *static text* or *dynamic text*. A static text definition contains the exact positioning of each glyph and cannot be modified using ActionScript. In dynamic text, the contents can be changed using ActionScript. No layout information is embedded in the SWF file for such definitions.

The glyph layout information embedded in the SWF file for static text makes it straightforward to render the text using SVG. Figure 5.8 shows an example of a static text field displaying the word Swiffy using an embedded font definition with the name Verdana.

```
<font>
  <font-face font-family="Verdana" units-per-em="20480"></font-face>
  <glyph unicode="S" d="M 1040 320..."></glyph>
  <glyph unicode="w" d="M 7730 1280..."></glyph>
  ...
</font>

<text font-family="Verdana" x="0 317 580 739 1000 1417">Swiffy</text>
```

Figure 5.8: SVG generated to display static text using an embedded font.

Server-side, the vector data that defines each glyph in the font is transformed using the procedure described in Section 5.1. At run-time the font definition is rendered using an SVG `<font>` tag, which contains a mapping from Unicode characters to vector data using `<glyph>` tags. These characters can then be used within an SVG text node. Using the `font-family` attribute, the browser will render the appropriate vector data. The layout information that was defined in the SWF file is used by the `x` attribute, which defines the horizontal offset of each glyph from the origin of the node.

Since no layout information is present in dynamic text, such text fields support line wrapping and word wrapping. This poses a problem for Swiffy since SVG does not provide

support for either type of wrapping; text needs to be laid out manually. We therefore use the *foreign object* feature of SVG to embed HTML within the SVG DOM (HTML does support wrapping). Figure 5.9 illustrates the code that is created at run-time.

```
<foreignObject width="1841">
  <body>
    <div style="font-family: Verdana; font-size: 560px;">
      <p>Text with line wrapping</p>
    </div>
  </body>
</foreignObject>
```

Figure 5.9: SVG generated to display dynamic text using an embedded font.

Although SVG fonts and foreign objects are part of the SVG specification, only Webkit-based browsers (such as Chrome and Safari) currently support them. For other browsers, Swiffy uses SVG path elements to display static text instead. The visual result is identical, but the semantics and the ability to select text are lost. This is another use of the ability to make run-time rendering decisions discussed in Section 4.1.1. Dynamic text is currently not supported in non-Webkit browsers, as rendering it using a path would require Swiffy to perform word and line wrapping itself.

5.3 Context for errors and warnings

One of the requirements for Swiffy is that the user should receive accurate feedback when a file can only be partially converted. We achieve this by providing the user with errors and warnings when unsupported features are encountered during the compilation process. In compilers that operate on source code, the context of such an error or warning is typically the line number. Since such a context does not exist in a SWF file, we propose an alternative mechanism to provide context to compile messages in this section.

The SWF file format uses a combination of definition and control tags. Typically, definition tags define graphical elements which can be placed on the screen by control tags. The concept of tags is not exposed to the user creating the SWF file in Adobe Flash Professional. Therefore, the name or position of a tag within the SWF file will not help the user in finding the source that triggered a compile message. For a definition tag, the frame it is encountered in is not helpful either: The placement and ordering of definition tags within the SWF file is transparent to the user, and a definition may be placed in a different frame than the frame it is used in. The only SWF properties directly controlled by the user are instance names and the frames objects are used in. These might therefore be helpful context for an error or warning.

5.3.1 The reference graph

A useful context should allow the user to navigate a file to find the cause of an error. To this end, the compiler will create a directed acyclic reference graph of SWF tags that links

definitions such as shapes, text and MovieClips to uses.

This graph will contain three types of nodes: a special root node, frame nodes and tag nodes. Figure 5.10 shows this graph for the simplest case where there is a single frame with one shape.

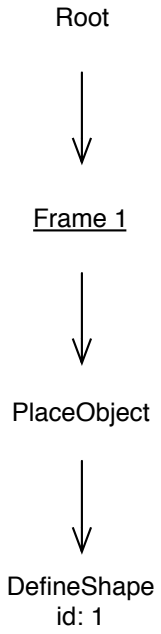


Figure 5.10: Reference graph for a simple SWF file. Frame nodes are underlined.

The root node of the graph represents the main timeline, serving as the main reference point for the user (the timeline is what the user sees when opening the file in Adobe Flash Professional). Every frame on the main timeline is represented by a frame node, all of which are referenced by the root node. A frame node can only reference a certain class of SWF tags: those that are placed in that frame *from the user's point of view*. This rules out all definition tags, since the user does not know in which frame a definition is placed.

Most nodes referenced by a frame node are nodes representing PlaceObject tags, which place an instance of a definition on the screen. In a SWF file, the PlaceObject tag is always placed in the frame in which the user placed the object on the stage. Since PlaceObject tags refer to definition tags, we will add edges in the reference graph between PlaceObject tags and the definition tags they refer to. Note that a single definition may have multiple incoming edges when it is placed on the stage multiple times.

We can now use this reference graph to provide context for compile messages. When a message is raised in the compiler, it will have a reference to the tag that raised it in the reference graph. Now, having built the complete graph, we can find the context of a message by calculating the paths from the root node to the tag node attached to the warning. Using

one such path, a message generated for the DefineShape tag in the example above might be displayed as:

```
Main timeline > Frame 1 > [message]
```

For any message that is raised by a definition tag, many different paths may exist. It is generally sufficient to output the shortest path to the tag that raised the warning. The shortest path in our reference graph corresponds to the quickest way in which a user can reach the part of the file the compiler refers to.

Sprites and buttons The graph structure for sprites and buttons, two types of definitions that can contain other objects, is similar. Like the root node, a DefineSprite tag directly references frame nodes. Figure 5.11 shows the reference graph for a SWF file that contains a sprite definition. In this example, the sprite defines a timeline of two frames. In the first frame, a DoAction tag is used that embeds ActionScript. Since this code is placed within the first frame from the user's point of view as well, an edge is added from frame node 1 to the DoAction tag.

When a warning is raised for this DoAction tag, the following context can be generated by finding the shortest path from the root to the DoAction node:

```
Main timeline > Frame 1 > "RedSpinner" > Frame 1: [warning message]
```

This context can be used by the user to easily navigate to the affected frame in Adobe Flash Professional.

Unreachable nodes There might be nodes in the graph that are not reachable from the root node, resulting from unused definitions in the SWF file. Such definitions are removed during the optimisation phase in our compiler using the reference graph. Warnings and errors for such definitions can therefore be suppressed, as they do not affect the output.

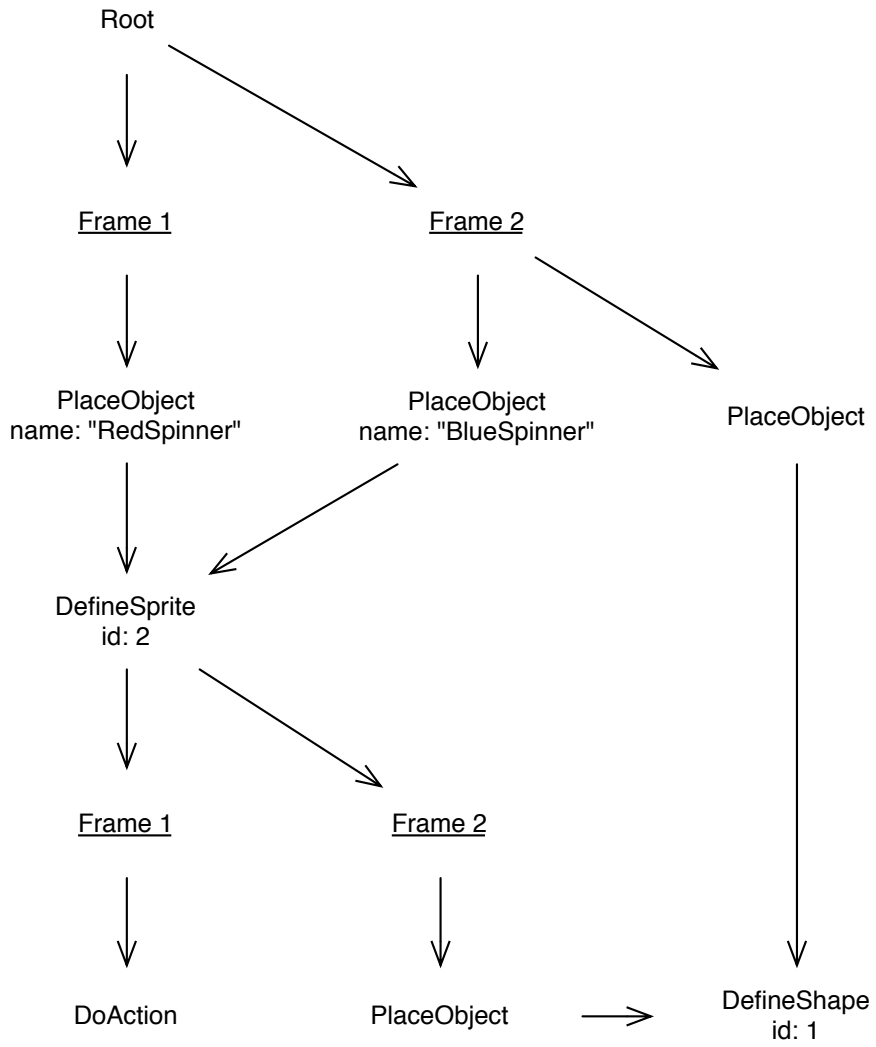


Figure 5.11: Reference graph for a SWF file containing a sprite. Frame nodes are underlined.

Chapter 6

ActionScript

In this chapter we take a closer look at the ActionScript language and how it is handled by Swiffy. First, we give an overview of the language and its relationship to JavaScript. We then describe the design of the ActionScript interpreter that is part of the Swiffy interpreter, followed by a discussion of compilation as an alternative to interpretation. In the final section we elaborate on our implementation of the ActionScript runtime libraries and how unsupported API calls are detected.

6.1 History

Scripting support was added to the SWF specification in 1997 with SWF version 2. At the time, scripting was limited to a predefined set of actions to control playback of the animation. Support for actual scripting using expressions and conditionals was added two years later with SWF version 4. The instructions in this version were added on top of the existing instruction set. Therefore, current ActionScript 2.0 has a mix of stack-based instructions such as `ActionPush` and instructions that semantically act as function calls such as `ActionNextFrame`.

SWF version 5 was the first version to be loosely based on ECMAScript [9], the standard derived from an early version of JavaScript. This was also the first version to be called ActionScript and was labelled as version 1.0. SWF version 7 introduced ActionScript 2.0 which added class-based inheritance syntax to the language. Although it adds keywords such as `class` and `implements` to the language, these serve only as syntactic sugar for prototype-based inheritance. Therefore, ActionScript 2.0 can be compiled into ActionScript 1.0 by Adobe Flash Professional.

Between SWF version 4 and 7, the semantics of parts of the language changed several times, ranging from the conversion of values to primitives to changes in the case sensitivity of the language. This complicates the development of a compatible interpreter, as it needs to switch semantics based on the version the SWF file is compiled for.

Adobe completely redesigned the ActionScript language with the release of ActionScript 3.0 in SWF 9. It uses a new virtual machine with just-in-time (JIT) compilation,

a new syntax and new runtime libraries. Current versions of the Flash player embed two virtual machines, as the new virtual machine is incompatible with ActionScript 2.0.

The semantics of the ActionScript 2.0 instruction language are defined in the SWF specification [19]. Although all instructions are documented, much of the semantics is unspecified, such as how invalid input should be handled. The ActionScript 3.0 bytecode language is described in greater detail in an overview of the ActionScript Virtual Machine 2 published by Adobe [18].

6.2 Interpreting ActionScript in JavaScript

Since the SWF file on which Swiffy operates contains only the ActionScript bytecode, source-to-source translation to JavaScript is only possible when using decompilation. We therefore considered both interpretation and compilation of ActionScript bytecode. In the former case, the Swiffy bytecode embeds ActionScript bytecode whereas in the latter case it embeds compiled JavaScript.

We have chosen to start with ActionScript 2.0 interpretation in Swiffy. It generally takes less effort to write an interpreter than a compiler [16, 21] and much of the experience and code obtained when writing the interpreter can be reused once a compiler is deemed necessary.

Figure 6.1 shows ActionScript bytecode, as represented in the JSON object, for an example program which performs addition of two numbers. Every instruction is represented by an object containing the type of instruction and optional arguments to the instruction. Type 305 represents a push instruction, type 10 addition and type 38 pops a value from the stack and prints it to the console. Within the VM, these instruction types are referenced by name using the mapping in Figure 6.2.

```
[
  { "type": 305, "value": 4 },
  { "type": 305, "value": 3 },
  { "type": 10 },
  { "type": 38 }
]
```

Figure 6.1: Example of ActionScript bytecode as represented in the JSON object.

```
vm.ActionType = {
  ADD: 10,
  TRACE: 38,
  PUSH_VALUE: 305,
  // ...
};
```

Figure 6.2: Mapping from instruction names to the numbers used in the JSON object.

A basic design of an interpreter for this bytecode would be instruction dispatch using one large switch statement. However, such an implementation can result in inefficient machine code due to the associated range checks and table lookups [10]. We therefore use the threaded code method [1] in which every bytecode instruction is translated into a function reference.

The JavaScript function `bindActions` translates the JSON representation of a list of bytecode instructions into function calls (Figure 6.3). This function is executed just once before execution of the ActionScript program. For every instruction, the JavaScript function `instruction` that implements it is fetched from the `vm.Instructions` array, which maps numeric instruction types to functions. Instructions that have arguments, such as the push instruction, need to have access to those arguments. We therefore use the JavaScript `bind` function to create a new function that once called, calls the instruction function with the JSON action as an argument.

```
1 vm.prototype.bindActions = function(jsonActions) {
2   var boundActions = [];
3   for (var i = 0; i < jsonActions.length; i++) {
4     var instruction = vm.Instructions[action.type];
5     var boundAction = instruction.bind(this, jsonActions[i]);
6     boundActions.push(boundAction);
7   }
8   return boundActions;
9 };
```

Figure 6.3: Translation of the JSON representation into function calls.

The main loop of the interpreter (Figure 6.4) now simply iterates and calls the functions in the `boundActions` array. An instruction pointer `ip` indicates the next function to execute in the array.

```
1 vm.prototype.execute = function(boundActions) {
2   this.ip = 0;
3   while (this.ip < boundActions.length) {
4     boundActions[this.ip++]();
5   }
6 };
```

Figure 6.4: Main loop of the interpreter.

Instructions that involve jumps such as the `jump` and `if` instructions are able to change the control flow by changing the instruction pointer `this.ip` to the desired jump target. The implementation of the `if` instruction is shown in Figure 6.6. This function uses the JSON object `data` to access the jump target `target` of the instruction.

Swiffy relies on the similarities of JavaScript and ActionScript such as its prototypical inheritance to construct an interpreter that is relatively light-weight and delegates as much

```
vm.Instructions[vm.ActionType.ADD] = function() {
  var argA = this.popNumber();
  var argB = this.popNumber();
  this.push(argB + argA);
};

vm.Instructions[vm.ActionType.TRACE] = function() {
  console.log(this.pop());
};

vm.Instructions[vm.ActionType.PUSH_VALUE] = function(data) {
  this.push(data.value);
};
```

Figure 6.5: Implementation of the `add`, `trace` and `push_value` instructions.

```
vm.Instructions[vm.ActionType.IF] = function(data) {
  if (this.pop()) {
    this.ip = data.target;
  }
};
```

Figure 6.6: Implementation of the `if` instruction.

work to JavaScript as possible. By doing so, the interpreted code benefits from the heavily optimised JavaScript JIT compilers found in modern browsers.

An example of this delegation is how JavaScript objects are used to represent ActionScript objects. There is no abstraction layer on top of these objects: members set on the ActionScript object are set directly on the JavaScript object. This enables the JavaScript JIT compiler to perform an optimisation that is known as *hidden classes* in the V8 engine that is part of the Google Chrome browser [14]. With hidden classes, the JIT compiler creates optimised code for all objects that have the same type of members.

Although this straightforward mapping works well for simple programs, it can result in problems when the semantics of JavaScript and ActionScript differ. For example, in JavaScript array indexing can be used to get a single character from a string:

```
return "foo"[0] // returns "f"
```

However, the same code should return `undefined` in ActionScript. Directly mapping ActionScript strings to JavaScript would therefore fail in this example, so care must be taken in the mapping to preserve the semantics. In this example, the problem is solved by adding a special case to the interpreter for the `get_member` instruction. In general, the behaviour of the ActionScript class has to be studied and tested in detail to find all subtle differences in the JavaScript mapping.

6.3 Compiling ActionScript bytecode to JavaScript

Swiffy currently uses an interpeter written in JavaScript to execute ActionScript bytecode. In this section, we consider the benefits and challenges of directly compiling ActionScript bytecode to JavaScript

While an interpreter is a good starting point to support ActionScript, there are two scenarios in which a compiler is more suitable. The most important reason is to eliminate the interpretation overhead. ActionScript performance has not been a concern for Swiffy since ads typically use little ActionScript. However, when one wants to convert Flash games or applications to HTML5, the interpretation overhead might become a bottleneck.

Secondly, if a compiler is able to generate human-readable JavaScript code, the developer will be able to edit and extend the file after conversion. This can for example be useful when integrating the conversion output with other JavaScript code.

To translate ActionScript bytecode to high-level JavaScript, decompilation techniques need to be used. This can be accomplished by either developing a decompiler that generates JavaScript directly, or by chaining an ActionScript decompiler with a source-to-source translator to JavaScript. Several ActionScript decompilers (such as Flare [22]) and translators to JavaScript (such as Jangaroo [6]) are freely available. ActionScript decompilation has mainly been studied in the context of malware detection for SWF files [11, 12].

Since the majority of Flash content is built using either Adobe Flash Professional or Adobe Flex, one can assume the compilation strategy of a single ActionScript compiler. This simplifies the design of a decompiler since various assumptions about the compilation strategy can be used [24]. For example, the compiler always leaves the stack empty after an expression. This simplifies assignment of variables to stack locations and reconstruction of expressions.

6.4 ActionScript runtime libraries

Nearly all ActionScript code uses the ActionScript runtime libraries provided by the Flash player. For example, every Flash ad needs a call to `MovieClip.stop()` to prevent the animation from looping and a call to `MovieClip.getURL()` to open a web page when the ad is clicked. Swiffy needs to provide its own implementation of the runtime libraries because these libraries are provided by the Flash player and are therefore not compiled into the SWF file.

The ActionScript runtime libraries are implemented in Swiffy using regular JavaScript objects and methods. Some libraries can be directly mapped to their JavaScript equivalent, such as the `Math`, `String` and `Date` classes. Interpreted ActionScript code does not have access to the Window context in the browser to get to these objects. Instead, we define a JavaScript object named `vm.Globals` that defines the topmost scope. ActionScript classes can then be mapped onto JavaScript classes by defining the mapping on the prototype of `vm.Globals`:

```
1 vm.Globals.prototype.Math = Math;
```

Classes that do not have a JavaScript equivalent are implemented in a similar way. The most commonly used class in ActionScript is the `MovieClip` class, which defines methods for controlling playback, creating graphics and various other utilities. As described in Section 4.4, the client-side interpreter in Swiffy uses a MovieClip class to represent the state of MovieClips. We do not want to expose this object to the ActionScript code, as it would expose the implementation of the Swiffy interpreter to the ActionScript code. Therefore we define an ActionScript-specific MovieClip class that serves as the interface between the ActionScript code and Swiffy's internal representation. The ActionScript-specific class delegates most of the work to the Swiffy internal class that represents the model. Other ActionScript classes such as `Stage` and `TextField` are implemented using the same design.

6.4.1 Detecting unsupported features

Swiffy does not support every API available in the ActionScript runtime libraries. An important design goal of Swiffy is to inform the user of the accuracy of the output during the compilation phase, but this is difficult to achieve considering the dynamic nature of ActionScript.

ActionScript 2.0 does not use *import* statements. Therefore, calls to the libraries have to be detected from the bytecode at compile time to be able to warn for unsupported features. It is impossible to solve this with 100% accuracy, but heuristics prove to be powerful in this scenario.

The Swiffy compiler uses a form of abstract interpretation of the bytecode at compile time to determine method calls and variable references. Stack operations are executed, but control flow is typically ignored. For example, function calls are not executed, only the name of the function being executed is registered (if available). The recorded named references are then matched with a known list of all members of the ActionScript runtime libraries.

This procedure results in both false negatives and false positives. For example, when the name of a function A to be called is the return value of another function call B, the name of function A will not be detected, since functions are not evaluated by the abstract interpreter. False positives may arise when a file defines a function with a name identical to one in the runtime libraries and calls this function.

The described detection procedure is an approximation: it cannot detect on which type of object a function is called or which arguments are passed to a function. When functionality is partially supported, this level of accuracy might not be enough. Type analysis [20] or run-time evaluation of the code may be used to improve the accuracy of the detection.

Chapter 7

Evaluation methodology

In this chapter we describe the methodology used to validate Swiffy against the requirements. First, we present the most important evaluation metrics: coverage, accuracy, overhead and performance. We then describe the dataset used to extract these metrics. Finally, we look into the framework we developed to evaluate performance.

7.1 Metrics

The main purpose of Swiffy is to convert SWF ads to HTML5. To evaluate this we need to know what percentage of SWF ads can be converted using Swiffy. We will use the term *coverage* to describe the percentage of files for which the Swiffy compiler does not give any errors or warnings. These files will be referred to as *fully supported* files.

Coverage: The percentage of files that can be converted without errors or warnings.

Coverage would be 100% if Swiffy did not give errors and warnings at all, even when a feature is used that is not supported. That would however violate the requirement that at least 95% of files without errors or warnings should have no defects. Coverage is only useful when combined with *accuracy*: the percentage of fully supported files that have no defects after conversion.

We would like accuracy to be close to 100%, but this is hard to achieve in practice due to undefined SWF and ActionScript behaviour or SWF files that rely on bugs in the Flash player. In typical usage of Swiffy, the user will manually evaluate the output created by Swiffy. If the output is not accurate, the developer can try to change the input file to work around the problem.

Accuracy: The percentage of fully supported files that have no defects.

Since mobile devices are the main target for Swiffy, we are interested in the ratio between the file size of Swiffy-generated HTML5 files and the file size of the original SWF files.

File size overhead: The ratio of the file size of the Swiffy output and the original SWF file.

Running on mobile devices restricts the computational power available. This can be a problem with SWF files that have not been created with the limitations of mobile devices in mind. We therefore measure the performance of Swiffy output on various platforms.

Performance: The rendering performance on desktop and mobile devices.

In Section 7.3, we will present our definition of acceptable rendering performance.

7.2 Dataset

To evaluate the metrics described above in the context of Flash display ads, we need to assemble a dataset of SWF files that accurately reflects display advertising.

Our dataset consists of one thousand SWF files randomly sampled from the Google Display Network, a network of thousands of websites that show ads uploaded to AdWords. Only ads created in 2012 have been used in order to guarantee a representative sampling of recent ads.

Characteristics of the dataset The selection of the dataset influences the type of SWF files used for the evaluation. One of the most important characteristics of the dataset is the distribution of ActionScript 2.0 versus ActionScript 3.0, since our implementation currently only supports the former. ActionScript 3.0 may have become the dominant language for application development using Flash, but in the development of display ads ActionScript 2.0 is by far the dominant scripting language. Although the dataset consists only of files uploaded in 2012, 93% use ActionScript 2.0 (Table 7.1). Only 7% of files use ActionScript 3.0, which was released in 2006.

ActionScript version	Percentage of dataset
2.0	93%
3.0	7%

Table 7.1: ActionScript versions used in the ads dataset.

SWF version	Percentage of dataset
6	3%
7	3%
8	42%
9	24%
10	28%

Table 7.2: SWF versions used in the ads dataset.

The major share of ActionScript 2.0 is not due to old files that are still in use, since the results in Table 7.2 show that nearly 53% of files have been exported for Flash player 9 or

later, the player that introduced support for ActionScript 3.0. The popularity of ActionScript 2.0 is more likely due to the simplicity it offers ad designers. Many display ads only use ActionScript to prevent an animation from looping and to attach an event listener to a button. ActionScript 2.0 is well suited for this task.

7.3 Evaluating performance

Ads converted by Swiffy should be perceived to run smoothly on recent mobile devices. We evaluate the rendering performance on three popular mobile devices released in 2011: the iPhone 4S, the iPad 2 and the Samsung Galaxy Nexus.

Rendering performance depends on various factors such as the complexity of the SWF file, the performance of Swiffy itself and the performance of JavaScript and SVG implementations in the browser. We will evaluate the combined performance: how well typical SWF ads that are not specifically built for mobile devices run after conversion with Swiffy.

To automatically measure whether an animation runs smoothly, we run the Swiffy output using an instrumented version of the interpreter that reports the time it takes to render each frame.

Based on empirical testing, we define acceptable performance to be a run where no frame is delayed by more than 100 milliseconds with respect to the frame rate specified in the SWF file. Furthermore, the sum of delays of the first 1000 frames has to be less than 1000 milliseconds.

Every benchmark is executed on a blank page containing only the conversion output at its actual size. This generally means the animation does not fill the whole screen, but is displayed at the size it would have when embedded in a website.

Chapter 8

Experimental results

In this chapter we apply the evaluation framework to the ads dataset to evaluate Swiffy on the most relevant metrics: coverage, accuracy, file size overhead and performance.

8.1 Coverage

In the ads dataset, 58% of all files are fully supported by Swiffy, meaning no errors or warnings are raised by the Swiffy compiler during the conversion process (Table 8.1). Warnings indicate a visual limitation such as a missing filter effect or blend mode that might be acceptable to some users. One or more warnings are presented to the user for 23% of the dataset.

The remaining 19% of files in the dataset raise one or more errors. This indicates a functional problem with the conversion such as lack support for ActionScript 3.0. Conversions in this category are likely to be unusable and the input file needs to be changed to make it compatible with Swiffy. The most common sources of compile errors or warnings are the lack of support for ActionScript 3.0 and blend modes.

Coverage category	Percentage of dataset
Fully supported	58%
Warnings, no errors	23%
Errors	19%

Table 8.1: Coverage of Swiffy on the ads dataset.

8.1.1 Browser coverage

The Swiffy interpreter uses two SVG features that are not (yet) supported by all major browsers: SVG fonts and SVG filters. Table 8.2 indicates the percentage of ads that are impacted by these limitations.

The most severe limitation is the lack of SVG filters in Internet Explorer 9. SVG filters are primarily used by Swiffy to implement Flash's filter effects, a feature used by 49% of successful Swiffy conversions. Combined with Swiffy's total coverage of 58%, only 28% of all files in the dataset can be converted to run successfully on Internet Explorer 9. We refer to the combination of Swiffy's coverage and browser coverage as *combined coverage* in Table 8.2.

Output using SVG fonts or SVG filters degrades gracefully in the affected browsers: text will be displayed using a default browser font in Firefox and filters effects will not be applied in Internet Explorer 9.

Browser	Missing standards	Output supported	Combined coverage
Google Chrome 22	None	100%	58%
Safari 6	None	100%	58%
Mobile Safari (iOS 6)	None	100%	58%
Mozilla Firefox 14	SVG fonts	95%	55%
Internet Explorer 9	SVG fonts & filters	48%	28%
Internet Explorer 10	SVG fonts	95%	55%

Table 8.2: Coverage of Swiffy on the ads dataset per major browser. Combined coverage represents the percentage of files in the dataset that are supported both by Swiffy and the specified browser.

With the addition of SVG filters support to iOS 6 and Internet Explorer 10 the only standard used by Swiffy that is not available in every browser is SVG fonts, used in 5% of the successful conversions. To get full cross-browser support, a future version of Swiffy could use the Web Open Font Format (WOFF) instead of SVG fonts. WOFF is currently supported by all major browsers.

8.2 Accuracy

Accuracy of the errors and warnings given by the Swiffy compiler is measured using manual visual inspection. A human evaluator views the original SWF and conversion result side-by-side in the latest version of Google Chrome to inspect it for both functional and visual defects.

Defects	Percentage of dataset
None	98%
Functional	1%
Visual	1%

Table 8.3: Accuracy of Swiffy on the ads dataset.

Of the successfully converted files (files without errors and warnings), 98% passed manual evaluation (Table 8.3). The 2% of defects is evenly split between various functional problems (resulting from bugs in Swiffy) and visual defects which arise both from bugs in Swiffy and from bugs in the browser implementation.

8.3 File size overhead

The requirements in Section 2.2 state that the Gzip-compressed file size overhead of the conversion should be less than 50%. A large overhead slows down the initial download, especially on mobile devices. Most ad networks and publishers therefore enforce a maximum size on SWF and HTML files.

Figure 8.1 plots the size of each SWF file in the dataset against the Gzip-compressed JSON object that results from the conversion. The average SWF file size is 34 kilobytes versus 37 kilobytes for the JSON file, a 9% overhead on average. Therefore on average the resulting files obey the 50 kilobytes maximum file size enforced by various ad networks, although 22% of conversions exceed this limit.

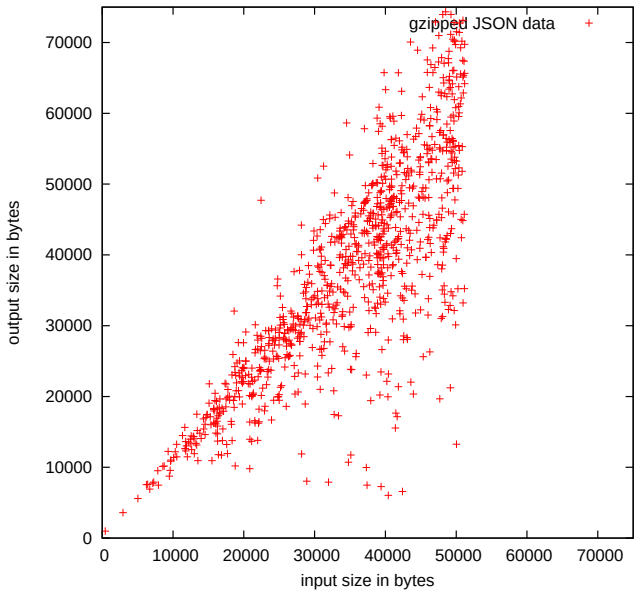


Figure 8.1: Comparison of input and output file size for the ads dataset. The x-axis represents the size in bytes of each SWF file plotted against the size in bytes after gzipping the JSON conversion result. Only files that could be converted without warnings or errors have been included.

We assume that it is the Gzip-compressed file size instead of the uncompressed size that is of most relevance. All modern browsers support Gzip compression. However, many ad networks use the uncompressed size to determine whether the file is no larger than 50 kilobytes. This poses a problem for Swiffy output as the uncompressed file size is 118 kilobytes on average. If that requirement does not change in the future, the Swiffy compiler could be changed to provide an option to output in a format optimised for uncompressed file size instead.

The output file size analysis only includes the JSON object that is generated by the Swiffy compiler, it does not include the size of the client-side JavaScript interpreter. By using a content delivery network, caching ensures that a client only needs to download this file once.

8.4 Performance

On modern desktop computers, 96% of ads achieve acceptable performance in the slowest browser tested (Internet Explorer 10) and over 99% in the fastest browser tested (Google Chrome 22). Files with performance problems on desktop typically make extensive use of animated filter effects, which are not yet hardware accelerated in any of the browsers tested. Some of these files appear to have performance problems when run with the Flash player as well. Unlike the Swiffy interpreter, we cannot instrument the Flash player to run the same benchmark. Table 8.4 summarises the results [1].

Desktop browser	Acceptable performance
Google Chrome 22	99%
Safari 6	98%
Mozilla Firefox 14	98%
Internet Explorer 10	96%

Table 8.4: Performance of Swiffy output on desktop browsers (higher numbers are better). The notion of acceptable performance is defined in Section 7.3.

On mobile devices, the lack of hardware acceleration for SVG filters is the most significant performance problem. About a third of all successfully converted files are unable to achieve acceptable performance on the three devices included in our test.

To assess the impact of SVG filters, Table 8.5 lists both the performance on all successfully converted files and on the 51% of files that do not require SVG filters. In the latter set, around 95% of the files achieved acceptable performance on the three mobile devices tested.

[1] Chrome, Safari and Firefox are tested on a MacBook Air running OS X 10.7.4 with a 2.13 GHz Intel Core 2 Duo and 4 GB RAM. Internet Explorer is tested on a Lenovo ThinkPad X201 with a 2.53 GHz Intel Core i5 and 4 GB RAM.

Mobile device	Acceptable performance (including SVG filters)	Acceptable performance (excluding SVG filters)
Samsung Galaxy Nexus Android 4.2, Google Chrome 18	65%	94%
iPhone 4S iOS 6, Mobile Safari	63%	96%
iPad 2 iOS 6, Mobile Safari	63%	96%

Table 8.5: Performance of Swiffy output on various mobile devices (higher numbers are better).

Chapter 9

Applications

Since the first public release in July 2011 on Google Labs, millions of SWF files have been converted using Swiffy, generating HTML5 that is used across the Internet. In this chapter, we list three applications in which Swiffy is currently integrated, along with examples of large websites using Swiffy output.

9.1 Flash to HTML5 conversion in Google AdWords

AdWords is Google's online advertising program, which can be used to create a variety of online ads, including image ads such as SWF files. Since November 2012, users uploading a SWF file to AdWords are automatically offered an HTML5 equivalent of their ad, powered by Swiffy. By selecting *Save HTML5 ad in addition to Flash ad*, they can reach customers that use devices without support for Flash. This option is only offered to the user if the SWF file is completely supported by Swiffy.

Figure 9.1 shows the AdWords interface after a user has uploaded a SWF image ad. A toggle button can be used to switch between the Flash and HTML5 previews.

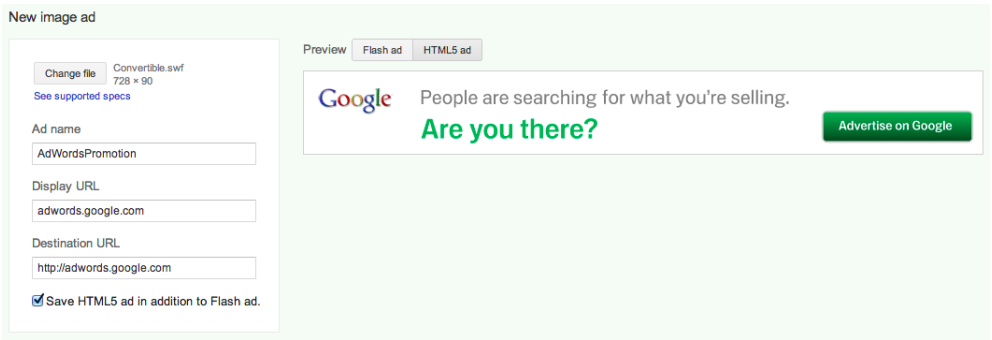


Figure 9.1: Flash to HTML5 conversion in Google AdWords, powered by Swiffy.

9.2 Extension for Adobe Flash Professional

In November 2011, the Swiffy Extension for Adobe Flash Professional was launched. It allows users to convert a SWF file to HTML5 with a single click from the editor. Once installed, a menu item *Export as HTML5* is added to Flash Professional (Figure 9.2). The extension invokes Swiffy as a web service to ensure it always uses the latest version of the compiler.

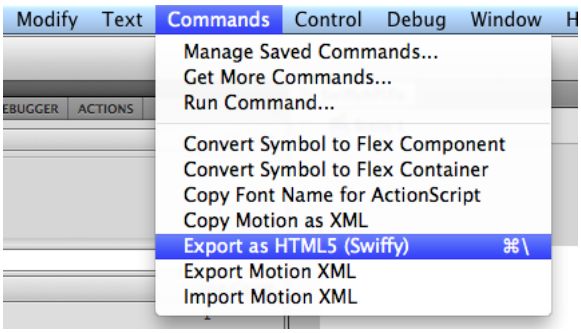


Figure 9.2: Swiffy Extension for Adobe Flash Professional.

Once the file is converted, the default browser is opened with the HTML5 result. Errors and warnings reported by the compiler are shown in the output panel to assist the user in making the file compatible with Swiffy (Figure 9.3).

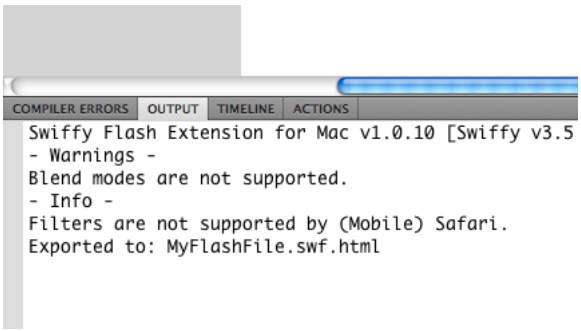


Figure 9.3: Conversion feedback for the Swiffy Extension for Flash Professional.

9.3 Online conversion service

Google Swiffy was first shown to the public in July 2011, when an online conversion service was launched on Google Labs [15]. It allows users to upload a SWF file and view the HTML5 conversion result and the original side-by-side (Figure 9.4).

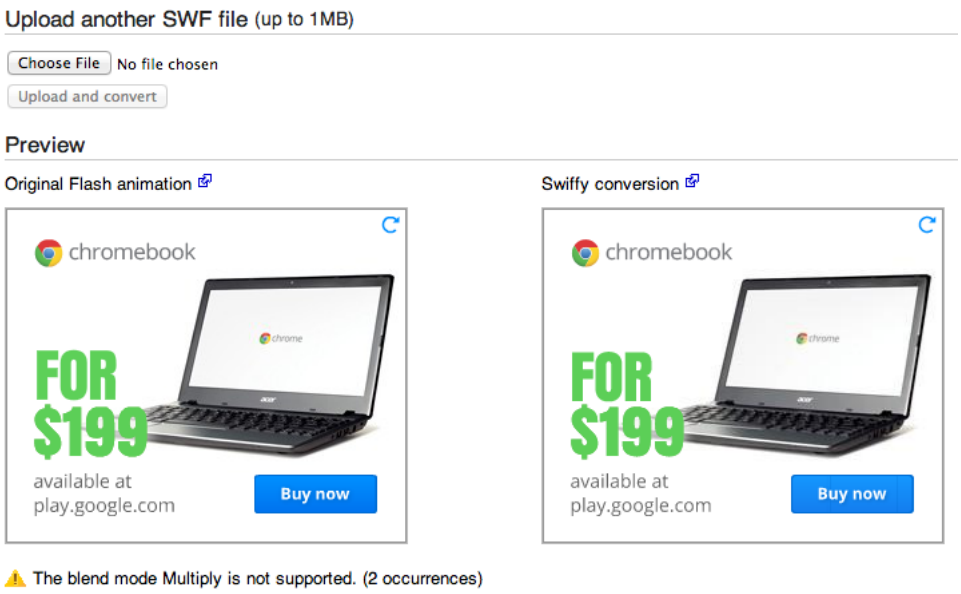


Figure 9.4: Swiffy online conversion service at g.co/swiffy.

Swiffy conversions are used on a large number of websites, both outside and inside Google. Examples of Google websites using Swiffy conversions are google.com/business, google.com/fiber and onehourpersecond.com (Figure 9.5).

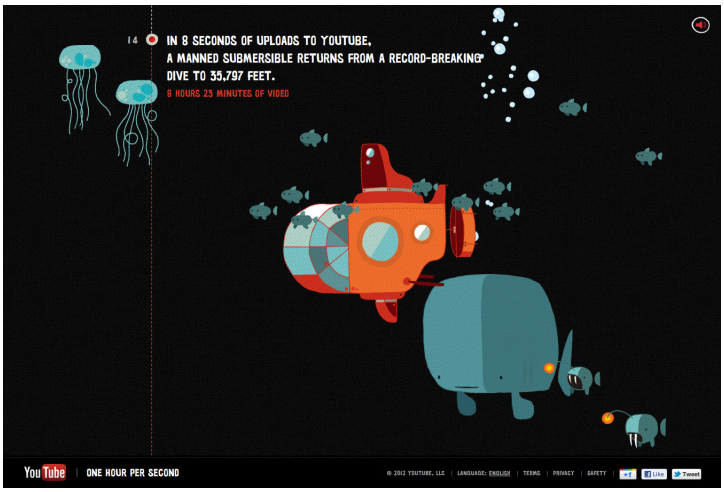


Figure 9.5: Swiffy combined with other HTML5 techniques at onehourpersecond.com.

Chapter 10

Conclusions and future work

In this chapter we summarise the results of this thesis and give pointers to future work.

10.1 Summary and conclusions

The goal of this thesis was to design and implement a Flash to HTML5 converter. In Section 4.1.1 we discussed the architecture of existing SWF converters, which are either pure server-side or pure client-side. We proposed a hybrid of these architectures with a server-side compiler and client-side interpreter based on a custom Swiffy bytecode. This has advantages compared to existing Flash converters in terms of performance, file size overhead and interoperability.

Implementation details of both the server-side compiler and client-side interpreter were discussed in Chapter 5, such as the path splitting algorithm used to make SWF vector graphics compatible with SVG. We gave details of both the intermediate representation of shapes and how shapes and text are transformed to SVG. A technique to provide context to the user for compile errors and warnings was presented in Section 5.3. This technique allows us to show the user in which part of the input file a warning or error occurred.

In Chapter 6 we discussed the ActionScript language and how it is handled by Swiffy. First, we described the implementation of the ActionScript interpreter in Swiffy, which uses the similarities of JavaScript and ActionScript to achieve efficient interpretation. We then discussed compilation of ActionScript bytecode to JavaScript, which may be desirable if either human-readable or more efficient code needs to be generated. Finally in Section 6.4 we described how the ActionScript runtime libraries are implemented in JavaScript and how the compiler detects unsupported API calls.

We presented our evaluation methodology in Chapter 7. It is focused around four metrics: coverage, accuracy, overhead and performance. In the following chapter, these metrics were used to evaluate Swiffy on a dataset of one thousand randomly selected Flash ads.

Of the files in the dataset, 58% were fully supported by Swiffy. The most common source of compile errors and warnings were lack of ActionScript 3.0 support and lack of support for blend modes. Use of SVG fonts and SVG filters was shown to limit coverage on some browsers, most notably Internet Explorer 9 which does not support SVG filters. If

Swiffy used the Web Open Font Format instead of SVG fonts, full cross-browser compatibility support could be achieved. Furthermore, by adding support for ActionScript 3.0 and blend modes, coverage can be increased to over 90%.

We then evaluated the accuracy of the conversion by measuring the percentage of fully supported files that have no defects. Using a human evaluator, 98% of conversions were determined to be free of defects, exceeding the requirement of 95% accuracy. We therefore deem the accuracy to be sufficient for our intended use case, in which the user will review the quality of the conversion.

In Section 8.3, we validated the file size overhead of the conversion against the requirements. With an average increase in file size of 9% this was well within the requirements.

Finally we measured the performance of the conversions on various desktop browsers and mobile devices. The lack of hardware acceleration for SVG filters was shown to cause performance problems on the mobile devices in our test.

We conclude that the rendering performance is sufficient for the application domain of display ads when files that require SVG filters are excluded, with over 99% of ads achieving acceptable performance on the desktop Chrome browser and around 95% on the Android and iOS devices under test. Files with SVG filters can only be used with care until the performance on mobile browsers improves.

In Chapter 9 we presented three real-world applications of Swiffy. The most significant application to date is the use of Swiffy for Flash to HTML5 conversion in Google AdWords, enabling advertisers to reach customers that use devices without support for Flash.

10.2 Future work

In this section we raise interesting areas for future work based on the results of this thesis. Apart from these areas, improvements to Swiffy can be considered such as extending its support for SWF features and the ActionScript runtime libraries, adding support for ActionScript 3.0 and further improving performance.

10.2.1 Partial evaluation of the Swiffy interpreter

Swiffy bytecode is interpreted by the Swiffy interpreter, resulting in an interpretation overhead. One way to improve the execution speed would be to partially evaluate the Swiffy interpreter for a given bytecode. Partial evaluation is a program specialisation technique that, when given a program and part of the input data, yields a new program in which all computations that depend on that part of the input data have been precomputed [21]. Since the majority of the input data to the Swiffy interpreter is known at compile time, the applicability of this optimisation may be widespread.

Program specialisation techniques can be applied to specialise both the Swiffy bytecode interpreter and the ActionScript interpreter. Thibault et al. have demonstrated that specialisation of bytecode interpreters can help bridge the gap between the performance of interpretation and compilation [28].

10.2.2 Browser benchmark based on Swiffy output

Browser vendors regularly develop benchmarks to showcase the performance of their browser or to improve the state of the art in browser development. Most benchmarks in recent years have focused on JavaScript performance and have helped in greatly improving JavaScript performance in all major browsers.

Currently no such benchmark exists for SVG performance. This makes the performance problems in SVG implementations invisible to browser developers.

The dataset and performance benchmark we created in Chapter 7 could be used as the inspiration for an SVG browser benchmark. It uses real-world animations with very complex SVG, which fits nicely with the trend of JavaScript benchmarks shifting from crafted test cases to real-world code.

For the benchmark to really focus on SVG performance as opposed to JavaScript execution, most of the Swiffy-specific JavaScript code would have to be eliminated from the test cases.

10.2.3 Detection of performance problems using static and dynamic analysis

Swiffy currently uses simple heuristics to raise a warning when a files uses so many objects or filter effects that it is unlikely to perform well in current browsers. As the performance results in Section 8.4 indicate, this might not be enough since about 5% of ads do not have acceptable performance on mobile devices. To improve upon this, a better model of the SVG performance of various browsers would need to be built. Static and dynamic analysis of the bytecode could then be used to determine whether the file is likely to trigger performance problems on any of those browsers.

Bibliography

[1] J.R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[2] T. Berners-Lee, J. Mogul, L. Masinter, P. Leach, R. Fielding, H. Frystyk, and J. Gettys. Hypertext Transfer Protocol - HTTP/1.1. 1999.

[3] C. Chambers. *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, Citeseer, 1992.

[4] C. Concolato, J. Le Feuvre, and J.C. Moissinac. Timed-fragmentation of SVG documents to control the playback memory usage. In *Proceedings of the 2007 ACM symposium on Document engineering*, pages 121–124. ACM, 2007.

[5] C. Concolato, JC Moissinac, and JC Dufourd. Representing 2D cartoons using SVG. *Proceedings of SMIL Europe*, 2003.

[6] Various contributors. Jangaroo project. `http://www.jangaroo.net`.

[7] Various contributors. JSwiff open-source project. `https://github.com/rsippl/jswiff`.

[8] P. Deutsch. GZIP file format specification version 4.3. 1996.

[9] E. Ecma. 262: ECMAScript Language Specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr, third edition, December*, 1999.

[10] M.A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[11] D.M.A.F. Files. SWF and the Malware Tragedy. In *OWASP Application Security Conference*, 2008.

[12] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious Flash advertisements. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 363–372. IEEE, 2010.

[13] GNU Project Free Software Foundation. Gnash open-source project. http://www.gnu.org/s/gnash.

[14] Google. V8 Design Elements. https://developers.google.com/v8/design.

[15] M. Gordon. Swiffy: convert SWF files to HTML5. http://googlecode.blogspot.co.uk/2011/06/swiffy-convert-swf-files-to-html5.html, 2011.

[16] D. Grune, H.E. Bal, C.J.H. Jacobs, and K. Langendoen. *Modern compiler design*, volume 3. Wiley, 2000.

[17] P. Hoschka. An introduction to the synchronized multimedia integration language. *Multimedia, IEEE*, 5(4):84–88, 1998.

[18] Adobe Systems Incorporated. ActionScript Virtual Machine 2 (AVM2) Overview. http://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf, 2007.

[19] Adobe Systems Incorporated. SWF File Format Specification Version 10. http://www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf_file_format_spec_v10.pdf, 2008.

[20] S. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. *Static Analysis*, pages 238–255, 2009.

[21] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[22] I. Kogan. Flare: ActionScript decompiler. http://www.nowrap.de/flare.html.

[23] Sencha Labs. Raphaël JavaScript library. http://raphaeljs.com.

[24] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Compiler Construction*, pages 153–184. Springer, 2002.

[25] Alessandro Pignotti. Lightspark. http://sourceforge.net/apps/trac/lightspark.

[26] 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.

[27] Tobey Tailor. Gordon open-source project. https://github.com/tobeytailor/gordon, 2010.

[28] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. *Rapport de Recherche, Institut National de Recherche en Informatique et en Automatique*, 1998.

[29] J. Van Wijngaarden and E. Visser. Program transformation mechanics. A classification of mechanisms for program transformation with a survey of existing transformation systems. *Technical report UU-CS*, (2003-048), 2003.