Basil - Scripting Flash from Smalltalk

Lucas Streit
Supervisor: Tudor Girba
Software Composition Group
University of Bern, Switzerland

October 22, 2007

Abstract

Several graphics file formats exist today, each with its own strengths and weaknesses. However, most of them are static formats. Adobe's Flash File Format (SWF) takes another approach. It is a vector graphics format which comes with a built-in script language, allowing the user to directly interact with the graphics. Furthermore, SWF has other interesting aspects like its popularity and availability, support for alpha blending or the possibility to include video and sound. The goal of this thesis is to provide a solution to create SWF files directly from Smalltalk. We present our implementation named *Basil* and as a validation we show how to use it for visualization purposes.

Contents

1 Introduction

Adobe Flash File Format (SWF) is a vector graphics format designed for publication on the web. At the moment it is the most popular file format to display animated vector graphics, far exceeding *SVG* which has not yet prevailed. To view an SWF file the user needs to install the *Adobe Flash Player* which is also available as a browser plugin for all major platforms in several languages [1]. According to a census of Adobe, 99% of web users have a *Flash Player* installed [2].

SWF has a built-in script language called *ActionScript* which allows it to react to user events. *ActionScript* is a *JavaScript*-style script language with an object model, data types and functions. Thanks to *ActionScript*, SWF can be used to create interactive graphics or even rich internet applications.

But SWF provides more. It has an advanced compression strategy that allows us to create files at a low file size. SWF may also embed bitmap data, joining both the strengths of vector and raster graphics. Even sounds and videos in different formats can be included in SWF.

With *Basil* we introduce a flash output framework for Smalltalk. Contrary to the common way of creating SWF files, *Basil* does not require us to own a proprietary development environment, e.g. the *Flash Creative Suite*. Instead, *Basil* lets us script SWF files directly from Smalltalk, allowing us to create an interactive graphics export with any data.

We chose Smalltalk as programming language because there are a number of visualization engines available which needed a strong graphics export. As an example, we used *Basil* to create a sophisticated graphics export from *Mondrian* [3], a scriptable visualization engine written in Smalltalk.

Basil also provides a subclass of the standard VisualWorks *GraphicsContext*. This allows us to fall back on existing graphics environments and to render them in SWF with little effort.

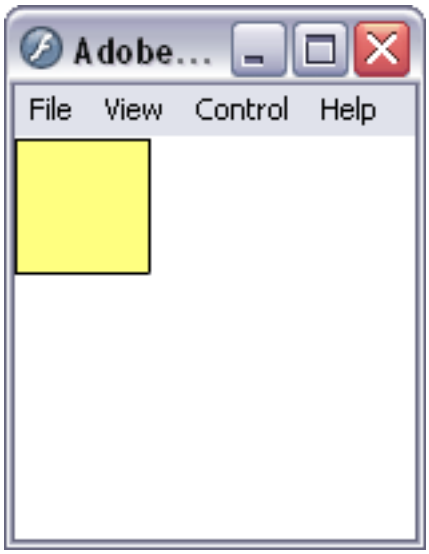
SWF is a proprietary format, owned by Adobe (formerly Macromedia). Nevertheless, the SWF and FLV File Format Specification License [4] explicitly grants the right to implement software which creates SWF files.

2 Basil by Example

In this section we show the basic facilities of *Basil* by using hands-on examples. All code is written in Smalltalk and uses *Basil* for scripting.

Creating a SWF file. In our first example we create a simple SWF file which contains a single colored square.

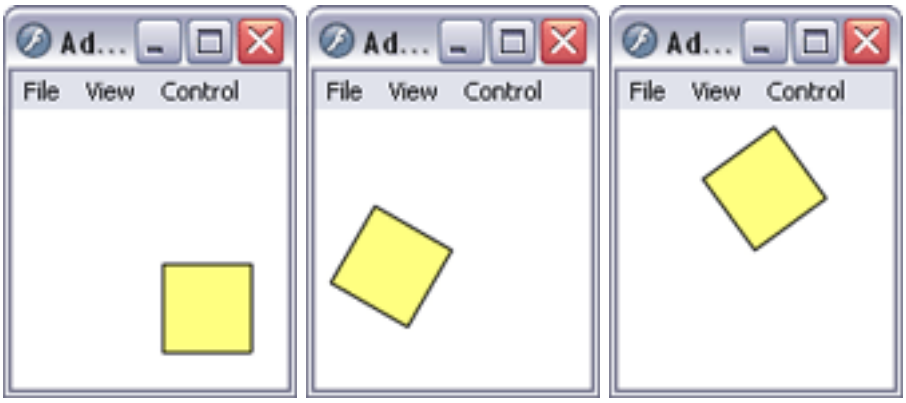
```
movie := (SWFFile new)
        frameWidth: 150;
        frameHeight: 150.
square := (SWFSquare new)
        side: 50;
        fillColor: (ColorValue yellow alpha: 0.5).
movie place: square
```



Creating an animation. Each SWF file consists of a number of frames which are displayed one after another with a specific number of frames per second. In this example we add several of these frames to our movie to create a simple animation. At each frame we increase the rotation of our shape to make it rotate around its origin. The rotation and position (among other things) of the shape are managed by a *SWFInstance* object.

```
movie := (SWFFile new)
        frameWidth: 150;
        frameHeight: 150;
        frameRate: 22.
```

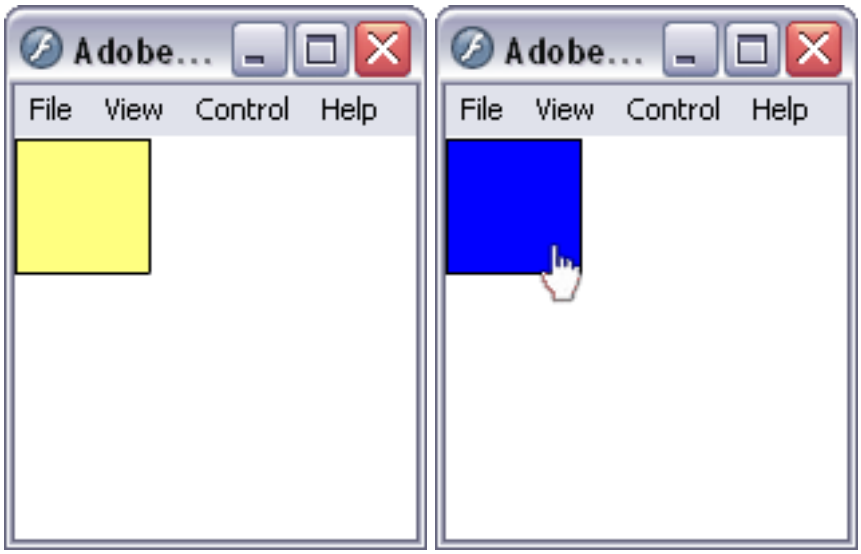
```
square := (SWFSquare new)
        side: 50;
        fillColor: (ColorValue yellow alpha: 0.5).
instance := movie place: square
0 to: 119 do:
        [:each |
        instance
                rotation: each * 3;
                position: 75 @ 75.
            movie nextFrame.
        instance := movie place: instance]
```



Creating a button. *Basil* also allows for interaction. In this example we add a button which will change its display state if the mouse moves over it. The *SWFButton* class has several extended *#place:* methods. These methods provide information about at which button state an object should be displayed. In this case we place two squares in the button. The first one is displayed at each button state, while the second one is only visible at the state *stateDown*.

```
movie := (SWFFile new)
        frameWidth: 150;
        frameHeight: 150.
square := (SWFSquare new)
        side: 50;
        fillColor: (ColorValue yellow alpha: 0.5).
highlight := (SWFSquare new)
        side: 50;
        fillColor: ColorValue blue.
```

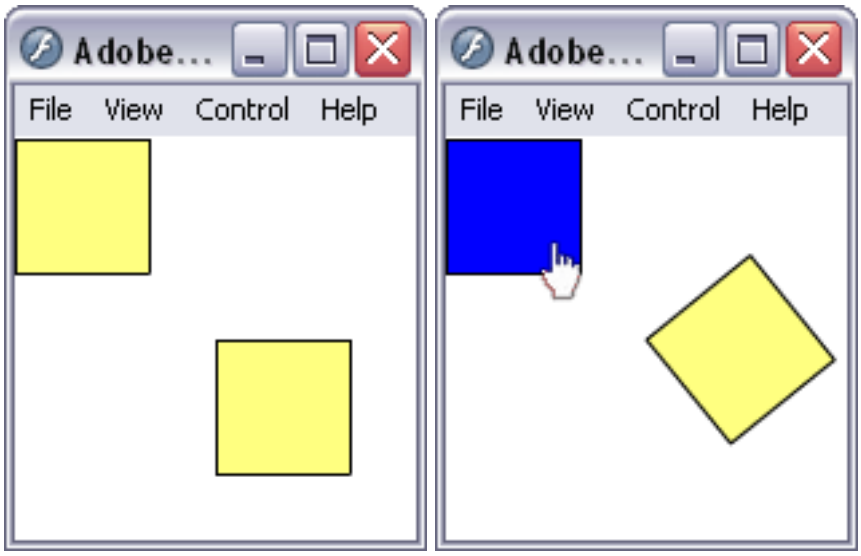
```
button := (SWFButton new)
        place: square;
        placeStateOver: highlight;
        yourself.
movie place: button
```



Adding actions. Next we join the previous two examples. We want the animation to start when the mouse is moved over the button and to stop when the mouse leaves it. Each time a user event occurs, SWF fires an action event. A button has several of these events. In this example we make use of the *rollOver* and *rollOut* events by adding start and stop commands to them. To prevent flash from starting playback immediately when the file is opened we also add a stop command to the first frame of the movie.

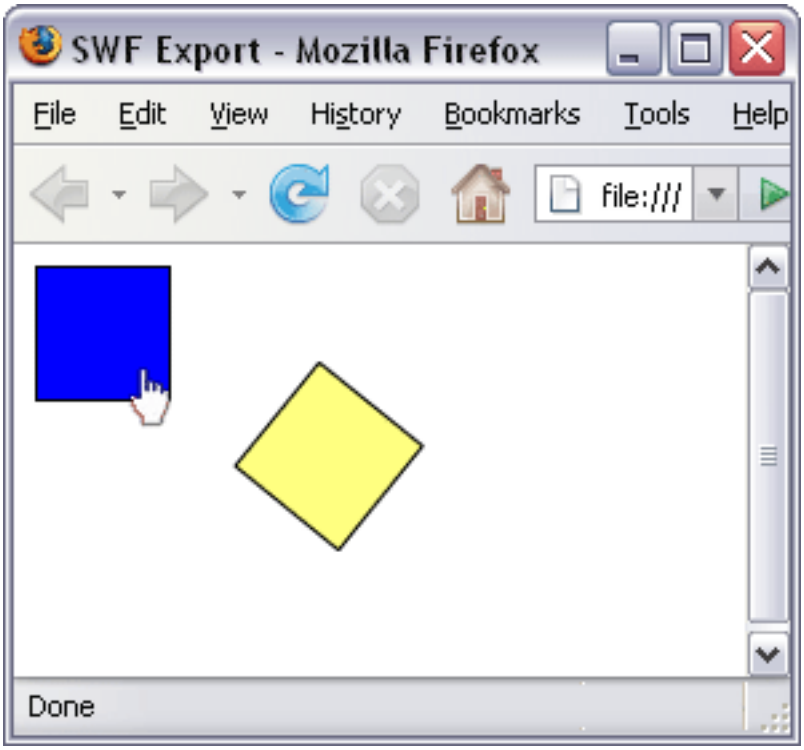
```
movie := (SWFFile new)
        frameWidth: 150;
        frameHeight: 150;
        frameRate: 22.
square := (SWFSquare new)
        side: 50;
        fillColor: (ColorValue yellow alpha: 0.5).
highlight := (SWFSquare new)
        side: 50;
        fillColor: ColorValue blue.
```

```
button := (SWFButton new)
        place: square;
        placeStateOver: highlight;
        yourself.
button on: 'rollOver' do: ActionStart new.
button on: 'rollOut' do: ActionStop new.
movie place: button.
movie doAction: ActionStop new.
instance := movie place: square.
0 to: 119 do:
        [:each |
        instance
                rotation: each * 3;
                position: 75 @ 75.
            movie nextFrame.
        instance := movie place: instance]
```



Storing the file. Finally, we want to store our newly created movie to a file. To do so we can either write the movie on an existing write stream or we can let Basil do the work by just sending *#save* to a *SWFFile* object. Additionally we can tell Basil to write an HTML page so we can launch the movie directly from a web browser.

```
movie save.
movie writeHTML
```



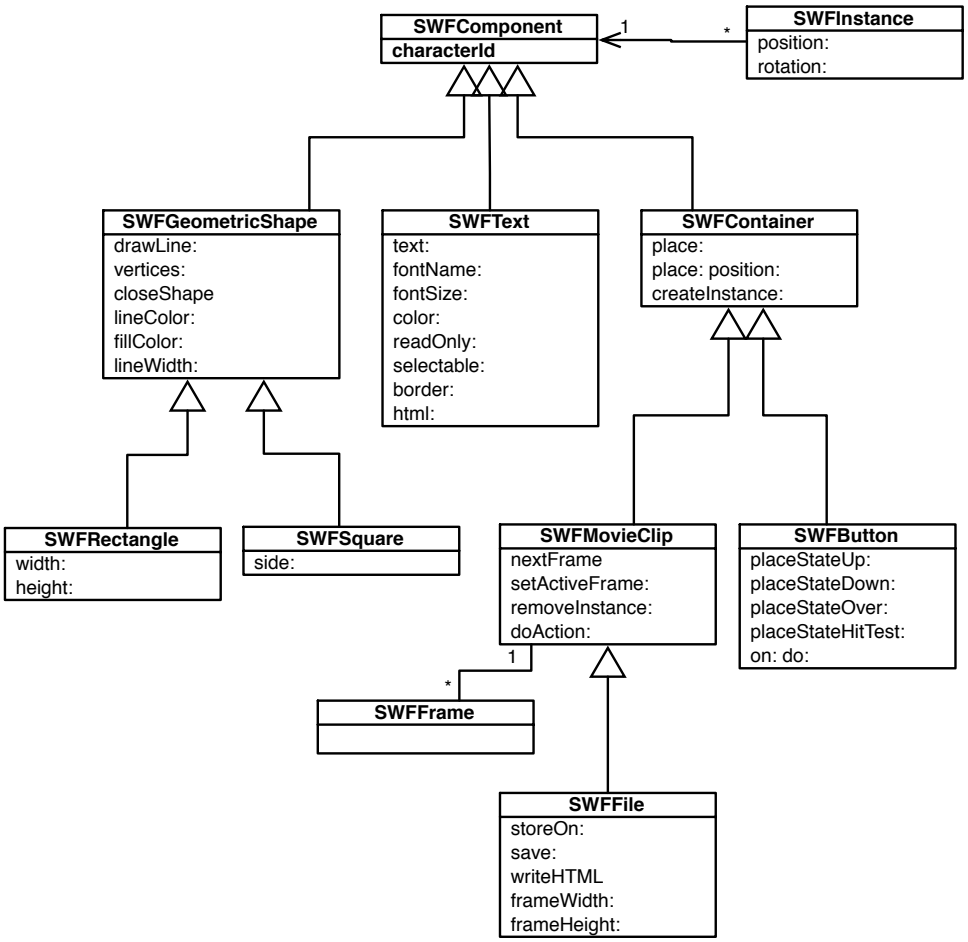
Summary. As shown above, *Basil* lets us create SWF files in a simple way. All the examples in this chapter are done with one single script. The script is completely written in Smalltalk and uses simple messages.

3 Basil Internals

Basil is designed as a two layers architecture. On top is the *Components Layer* which acts as interface between the Smalltalk program and the *Storage Layer*. At the bottom is the *Storage Layer* which assembles the byte code and stores it on a file.

3.1 Components Layer

As shown in the tutorial, a SWF file may consists of many different components like shapes, buttons, etc. The *Components Layer* provides a class for each of these components. The following figure gives an overview of the available classes in the *Components Layer*.



SWFComponent. The root of the *Components Layer* class hierarchy is the *SWFComponent* class. This class manages the *character id*, a unique identification number each entity in a SWF file must have.

SWFGeometricShape. The *SWFGeometricShape* class is used to create all kinds of geometric shapes. It defines the drawing interface for specialized subclasses like *SWFSquare* or *SWFRectangle*. Moreover, it may also be accessed directly to create general shapes not covered by one of the subclasses. A shape consists of a list of straight edges called a path. A path may be open or closed (which means that the start point of the path is the same as its end point). A new edge can be added to the path with the *#drawLine:* method. Notice that a new edge always starts at the end point of the last edge. As an argument the *#drawLine:* expects a point representing the relative movement in pixels. `shape drawLine: 3@5` for example draws a new edge which goes 3 pixels in the x-direction and 5 pixels in the y-direction. The *#vertices:* method in contrary expects an array of absolute vertices. Finally, the *#closeShape* method may be used to ensure that the path is closed.
The appearance of the shape outline can be defined using the *#lineWidth:* and *#lineColor:* methods. The line width is specified in pixels. Notice that SWF always draws a shape outline, even if the line width is set to 0. To draw a shape without a visible outline, the line color has to be specified as transparent.
The fill color of a shape is set with the *#fillColor:* method. It expects either a standard *VisualWorks ColorValue* or a *ColorWithAlpha*, which can be easily created by sending *#alpha:* to an existing *ColorValue*.

SWFText. To display a text, Basil offers the *SWFText* class. The text that should be displayed is specified with the *#text:* method.
There are also messages for the font size, the font name and the font color. Consider that fonts are rendered by the playback platform which means that the specified font should be available there during playback. To reduce this problem SWF offers a number of indirect font names. They are mapped to actual font names by SWF depending on the playback platform. The most important ones are '*_sans*', '*_serif*' and '*_typewriter*'. Furthermore *SWFText* has several other messages which affect its appearance, for example *#bold:*, *#italic:* or *#align:*.
The *SWFText* class is also used to create text input fields. By sending *true* to the *#readOnly:* method, the text is made editable. Other interesting methods in this context are *#border:* to frame the text with a border or *#selectable:* to make it selectable.
SWFText also understands a small subset of HTML tags, for example *a, br, p, font* or *ul*. `text html: true` tells Basil to interpret the specified text as HTML.

SWFContainer. A *SWFContainer*, in contrast to the classes described above, has no appearance on its own. Its appearance is defined by a list of children components that are added to the container. The children components may be shapes or texts as well as other containers.
The *#place:* method adds a new component to a container. Additionally, with *#place: position:* the position of the component inside the container can be specified. Notice that each container has its own coordinate system. Moreover, all transformations, e.g. translations or rotations, performed on the container are automatically performed on all its children components.

SWFInstance. Each time a component is placed inside a container with a *#place:* method we get as a return value a *SWFInstance* object. The reason for this is that SWF distinguishes between the definition of a component and its placement inside a container. The definition side is done by the subclasses of *SWFComponent* while a *SWFInstance* represents the component in a specific context, e.g. inside a button.
The goal of this concept is to allow components to be reused. The same component is only defined once in a SWF file but may be used multiple times.
The *SWFInstance* object holds information about the position and the rotation of a component inside the container. Instances are also used to change the rotation or the position of a component in a subsequent frame. This is done by placing an instance of the component in a container rather than the component itself.

SWFButton. Buttons make a substantial contribution to the interactivity of SWF. Due to them, SWF is able to react to various user events (e.g. mouse clicks) and to perform corresponding responses.
A button has three display states. *stateUp* is the default state. It is displayed whenever the mouse is outside the button. *stateOver* is displayed when the user moves the mouse over the button and *stateDown* is displayed when the user clicks on it.
A fourth state, *stateHitTest*, is never displayed. Instead, it defines the area of the button which reacts to user events. This hit area does not necessarily match the visible shape of the button.
At each of these display states, children components can be added individually. To do so, the *SWFButton* class has multiple extended place methods, e.g. *#place: stateUp: stateDown: stateOver: stateHitTest:*. A boolean determines each time whether the component is displayed in the corresponding state or not. If a component is added using the standard *#place:* method it is added to all four states.
As mentioned above, buttons may respond to user events. The following list shows the events that can be trapped by SWF.

- *press*: A mouse button is pressed down while the mouse is inside the

hit area.

- *release*: A mouse button is released while the mouse is inside the hit area.
- *releaseOutside*: A mouse button is released while the mouse is outside the hit area. This event only occurs if the mouse was initially pressed down inside the hit area.
- *rollOver*: The mouse enters the hit area while no mouse button is pressed.
- *rollOut*: The mouse leaves the hit area while no mouse button is pressed.
- *dragOver*: The mouse is dragged inside the hit area while a mouse button is pressed.
- *dragOut*: The mouse is dragged outside the hit area while a mouse button is pressed.

An *ActionScript* statement is added to a *SWFButton* using the *#on: do:* method. For example, `button on: 'release' do: ActionStop new` executes a stop command each time a mouse button is released inside the hit area of the button. Different events may be concatenated using a comma, e.g. `'release, releaseOutside'`. Details on *ActionScript* will follow in chapter 3.3.

SWFMovieClip. A *SWFMovieClip* consists of one ore more frames which are displayed one after another with a specific number of frames per second. One of these frames is always the active frame. The *#nextFrame* method marks the next frame as active and adds a new frame if necessary. Besides this the *#setActiveFrame:* method can be used to mark a specific frame index as active.
The *#place:* method always adds the specified component to the active frame. However, the component is not displayed only at the active frame. It remains visible in the following frames until it is explicitly removed with the *#removeInstance:* method. Notice that this method expects an object of the type *SWFInstance* rather than a *SWFComponent* object.
A *SWFSprite* object too may have *ActionScript* statements appended. Actions added through the *#doAction:* method are performed each time the playback reaches the active frame.

SWFFile. As shown in the tutorial, the *SWFFile* class represents the entry point of *Basil*. On the one side, as a direct subclass of *SWFMovieClip*, it inherits all the functionality to add and remove components, etc. On the other side it provides additional methods to store the movie on a file, e.g. *#storeOn:*. This method stores the *SWFFile* on an existing *WriteStream*.
Before storing the file there are some general attributes which should be

specified. *#frameWidth:* and *#frameHeight:* determine the dimensions of the file while *#frameRate:* sets the number of frames displayed per second. The background color can be changed with the *#background:* method.
To facilitate storing, the *SWFFile* class also provides a *#save:* and a *#writeHTML* method. The first one prompts for a filename and stores the movie to the specified file. The second one writes a HTML page that embeds the movie so we can launch it directly from a web browser.

3.2 SWFGraphicsContext

The *SWFGraphicsContext* class provides a simple way to render graphics from an existing graphics environment in SWF. It implements the primitives given by the *VisualWorks GraphicsContext*. For example, *#primDisplayPolyline: at:* draws a shape outline given by an array of vertices.

```
primDisplayPoliline:  pointCollection at:  aPoint

   |aShape |
   aShape := (Basil.SWFGeometricShape new)
          lineWidth:  self lineWidth;
          lineColor:  self lineColor;
          vertices:  pointCollection.
   movie place:  aShape position:  aPoint + pointCollection
      first.
```

This allows us to create an SWF export by just replacing the graphics context used so far with a *SWFGraphicsContext*. In chapter 4.1 we show how we used the *SWFGraphicsContext* to export graphics from *EyeSee*.

3.3 Actions

Probably the most interesting feature of SWF is its interactivity. A built-in script language called *ActionScript* allows it to react to user events and to generate suitable responses.
ActionScript is a script language based on *ECMAScript*. Thus its syntax is quite similar to the syntax of *JavaScript*. A valid *ActionScript* statement for example would be:

```
aShape._x = 10;
```

Unfortunately, *ActionScript* must be translated into an assembly language before storage. The assembly code will then be interpreted at Runtime by a stack machine, which is included in the *Flash Player*.
The present version of Basil can not compile *ActionScript*. Actions are implemented at a low-level instead. However, it is still possible to add all kinds of action statements, e.g. variable assignments, conditional statements or loops. Even functions can be definend (and then be called during playback).

As we saw in chapter 3.1, there are two possibilities to append actions: `button on: do:` and `movie doAction:` methods. Both expect an object of the type *ActionRecord*. The *ActionRecord* class has various subclasses, each representing an assembly instruction understood by the *Flash Player's* stack machine. For example there are classes for pushing values on the stack, for summing up two values, etc.

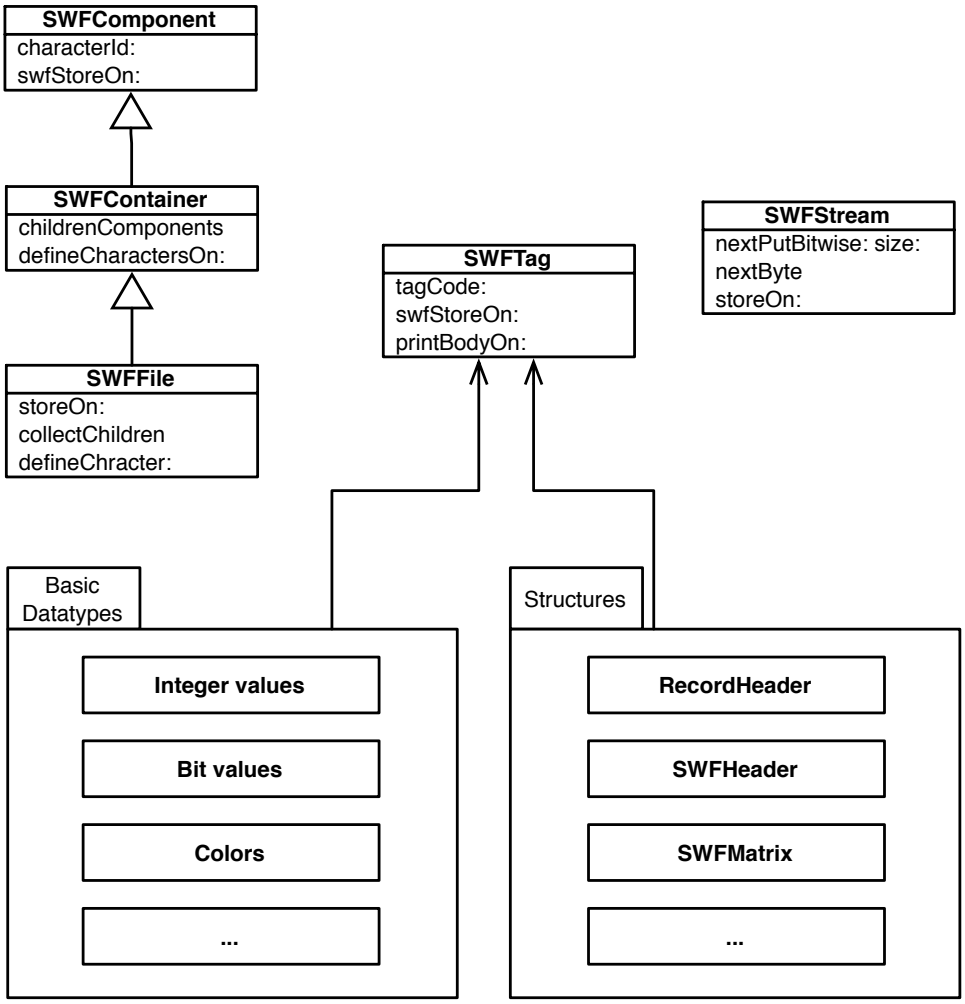
Most of the action record classes provide methods which facilitate the creation of action statements. The following example shows how the *ActionScript* statement from above is implemented in Basil.

```
ActionSetMember object: 'aShape' member: '_x' value: 10.
```

The *ActionSetMember* class in this case assembles a list of action records needed to perform the desired variable assignment. Other classes have similar methods.

3.4 Storage Layer

Once a movie is defined, we need to store it to a file. As shown, the *SWFFile* class provides several methods do so. As soon as one of these methods is called, the *Storage Layer* gets active. The following figure gives an overview of the most important classes of the *Storage Layer*.



SWF file structure. A SWF file is built up with tags. Because SWF was designed to create files with a low file size, the tags are stored as byte code rather than as plain text. Each file begins with a header tag and ends with an end tag. Between these two tags there may be various other tags, depending on the actual movie.
The available tags can be divided into two categories. The *definition tags* on one side define the content of the SWF file - e.g. shapes, texts etc. They do not cause anything to be rendered. The *Control tags* on the other side create

and manipulate instances of the previously defined objects and cause them to be rendered on the screen. Furthermore, *control tags* are used to control the flow of the movie.

Tag Format. Each tag begins with a record header. This header specifies both the tag type and the tag length in bytes. After the header follows a list of fields, depending on the actual tag type. Each of these fields has a specific type, either a *structure* or a *basic datatype*. The following table shows the structure of an example tag:

SetBackgroundColor

Field	**Type**	**Comment**
Header	Recordheader	Tag type = 9
BackgroundColor	RGB	Color of the display background

Basic datatypes. SWF supports a range of *basic datatypes*:

- *Integer data types*, i.e. 8-bit, 16-bit, 32-bit and 64-bit signed and unsigned integer types. All integer types are byte aligned and stored in *little-endian* byte order. *Basil* implements these data types by extending the Smalltalk *Integer* class with methods like *#asUnsignedInteger16* or *#asSignedInteger16*.
- *Bit values* in contrary are not byte aligned. They are of variable bit length, possibly spreading over multiple bytes. Negative bit values are stored using the *two's complement representation*. If a bit value is followed by a byte aligned data type, the unused bits are padded with zeros. In *Basil*, unsigned bit values can be treated like standard integers. Signed bit values in contrary are created by sending *#asSignedBitPaddedTo:* to an *Integer* object.
- All *x-y coordinates* are stored as integer values in a unit called *twips*. A *twip* is 1/20th of a logical pixel. In *Basil*, all coordinates are specified in pixels. Nevertheless, it is still possible to benefit from the accuracy of *twips*. By specifying a coordiate as a *float* instead of an *integer*, the coordinate is rounded to the next twip.
- *Strings* are stored as a sequential list of characters, where each string has to be terminated by a zero byte.
- *Colors* are stored either as a 24-bit red, green and blue value or as 32-bit red, green, blue and alpha value. Each color or alpha channel thereby is stored as 8-bit unsigned integer.

Structures. Some of the tags in SWF make use of special *structures*. Structures are generally a collection of grouped basic datatypes. For example, the *SetBackgroundTag* shown above includes a *RecordHeader* structure. The *RecordHeader* on the other hand consists of an unsigned integer representing both the tag type and the tag length:

Recordheader (short)

Field	Type	Comment
TagCodeAndLength	UI16	Upper 10 bits: tag type Lower 6 bits: tag length

SWFStream. As shown above, some of the basic data types in SWF are not byte aligned. To be able to store data bit by bit, *Basil* extends the standard *WriteStream* class of Smalltalk. The *SWFStream* provides the *#nextPutBitwise: size:* method, which writes the next *size* bits on the stream. The *#nextByte* message ensures that successive data is written at the beginning of the next byte.

Basil Storing process. The *#storeOn:* method of the *SWFFile* class starts the storing process of *Basil*. First of all, the *SWFFile* class creates a new *SWFStream* object and stores the header tag on it. After that, the *SWFFile* collects all the components which were added to it. If one of these components is a container, the children of the component are collected too. The reason for this is, that all definition tags have to occur in the file before any control tag can refer to it.
Once this is done, the *#swfStoreOn:* method of each of these children is called and causes the components to store themselves on the *SWFStream*. Finally, the *SWFStream* is stored on the *WriteStream* initially passed to the *SWFFile* object.

4 Basil in Action

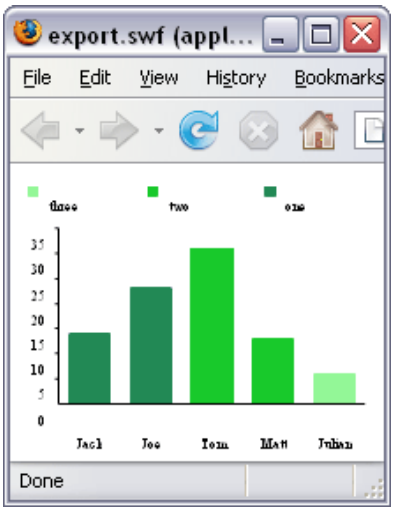
In this section we validate *Basil* by creating graphics exports both from *EyeSee* and *Mondrian*.

4.1 EyeSee Integration

EyeSee [5] is a diagram drawing engine written in Smalltalk. To render *EyeSee* diagrams in SWF, we simply add a *#swf* method to the *DiagramRenderer* class of *EyeSee*. This method creates a *SWFGraphicsContext* object and tells *EyeSee* to render itself on it. Finally we save the diagram to a file and create a HTML page which embeds it.

```
self diagram
        gcWrapper:  DiagramGraphicsContextWrapper new;
        setup.
swfGc := (SWFGraphicsContext new)
        font:  diagram getDefaultFont;
        frameWidth:  diagram width;
        frameHeight:  diagram height;
        yourself.
self diagram displayOn:  swfGc.
swfGc
        save;
        writeHTML
```

The following figure shows a vertical bar diagram created with *EyeSee* and rendered in SWF with *Basil*.



Using the *SWFGraphicsContext* is very simple. There is no need to implement a new rendering process. Instead, a few lines of code are enough to render *EyeSee* diagrams in SWF. This graphics export however does not support any interactivity.

4.2 Mondrian Integration

In this section we show how we use *Basil* to create a sophisticated graphics export for *Mondrian* visualizations. *Mondrian* [3] is a scriptable visualization engine written in Smalltalk. It already includes export options for png, svg and postscript. However, none of these exports covers the full power of exploring a *Mondrian* visualization directly in a dedicated environement. Hence our goal was to created an export which covers all *zooming*, *scrolling* and *dragging*.

Mondrian visualizations basically consist of nodes which are connected by several edges. Within the *Mondrian* environment, the user can interact with the visualization. It is especially possible to zoom and scroll the visualization and to drag around a nodes. Thereby, the connected edges are redrawn dynamically.

To make the *Mondrian* visualization zoomable and scrollable in Flash, we emulated the interface of *Mondrian* in SWF. The *MondrianInterface* class draws the buttons for zooming and the scrollbars for scrolling the visualization. It also adds a text field which displays additional information if the user moves the mouse over a figure in the visualization. To add the interactivity, the *MondrianInterface* class uses *ActionScript*.

The actual visualization is rendered by the *MondrianSWFContext* class. It renders the visualization on a *SWFMovieClip* and places that movie clip in the *MondrianInterface*.
The *MondrianSWFContext* creates a *SWFMovieClip* for each edge and sprite. The nodes are caused to render themselves on the movie clip. The movie clips of the edges in contrary stay empty since they have to be drawn dynamically at runtime. To do so, we need SWF to know which edge belongs to which node. Therefore, we store references of all incoming and outgoing edges of a node in the node movie clips. The incoming edges for example are stored in an array named *ic*.

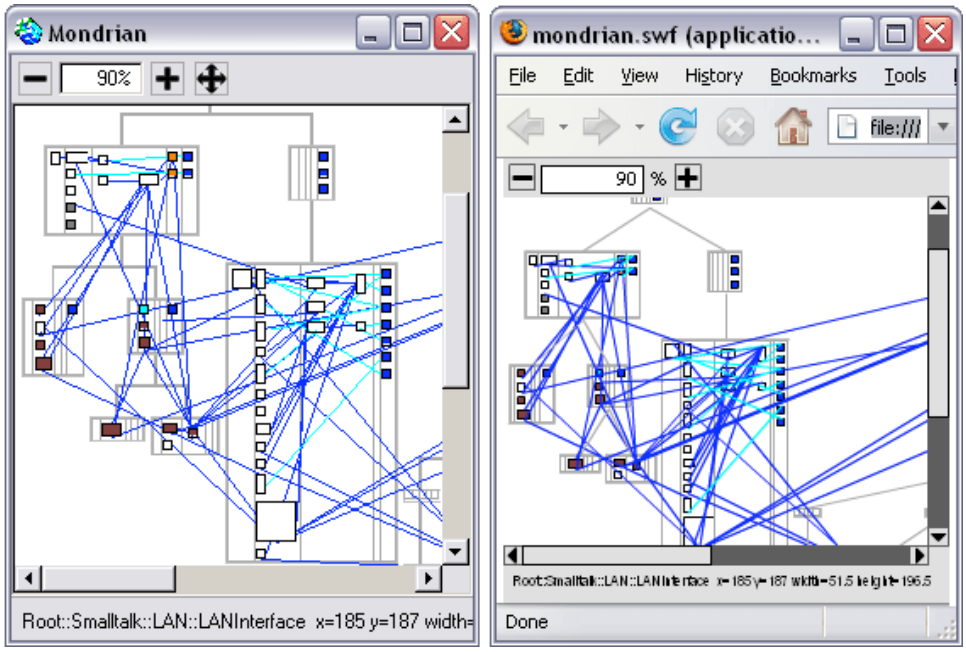
```
nodeClip doAction: (ActionSetVariable name: 'ic' value:
   (ActionInitArray elements: self incoming))
```

Furthermore, we store information about to which nodes an edge is connected. For example, the node which an edge comes from is stored in a variable called *from*.

```
edgeClip doAction: (ActionSetVariable name: 'from' value:
    self fromNode absoluteName)
```

With that information, we can define a function in Flash, that finally draws an edge between two nodes. This function will be called cyclically while a node is dragged and causes every connected edge to be redrawn.

The following figure shows the dedicated environement of *Mondrian* on the left, and a *Mondrian* visualization rendered with *Basil* on the right. Both figures show a *Blueprint complexity* visualization created with *Mondrian*.



The two graphics look quite similar. Moreover, the SWF exports provides the same interactivity as *Mondrian* does in its environment. The difference is that the SWF file may be easily shared with other people. However, this graphics export is more complex. We had to implement a number of additional classes and model some of the *Mondrian* logic in SWF.

5 Future Work

The present version *Basil* covers the core aspects of SWF. However, there is still a number of interesting features which we plan to add in the future. This section contains a list of the most important ones.

Action Script

As we saw in Section 3.3, *Basil* implements actions at a low level. A more abstract way to add *ActionScript* would definitely improve the comfort of use. As a first step we plan to include a parser for *ActionScript*. Valid *ActionScript* statements could then be added to a component as a String, e.g. `button do: 'on(click) {aShape._x = 10;}'`.

File Compression

As SWF is designed for publication on the web, its files should be as compact as possible. The compression strategy of SWF includes an option to compress the files using the *ZLIB open standard*. Until now, *Basil* stores the SWF files uncompressed. Information about SWF file compression can be found in the Flash File Format Specification [6], page 17.

Glyph Text

SWF supports two kinds of text. On the one side, *Device Text* is rendered by the playback platform. On the other side, *Glyph Text* embeds character shapes in the SWF file. While using *Device Text* produces slightly smaller files, it still has some disadvantages in comparison with *Glyph Text*. For example, *Device Text* requires the specified font to be available on the playback platform. Moreover, *Device Text* can not be rotated and does not support alpha blending. For now, *Basil* does not support *Glyph Text*. Information about *Glyph Text*, including an instruction of how to convert *TrueType* fonts to SWF glyphs can be found in the Flash File Format Specification [6], chapter 12.

Curved Edges

Straight edges can be drawn by sending a #drawLine: message to a *SWFGeometricShape* object. Anyway, there is no way to draw curved edges at the moment. SWF offers a way to do so using, quadratic bezier curves. A new kind of structure had to be implemented, the *CurvedEdgeRecord*. The Flash File Format Specification [6], chapter 8, section Edge Records gives details about this record.

Bitmaps

As SWF is a vector graphics format, it is not optimal for some types of graphics, for example photographs. Therefore, SWF offers a way to embed bitmap formats. The supported formats are *JPEG bitmaps* for lossy compression and *ZLIB bitmaps* for lossless compression. Both types of bitmaps may also contain an alpha channel. Chapter 10 of the Flash Files Format Specification [6] lists the tags needed to include bitmaps.

6 References

[1] *Adobe Flash Player* download
http://www.adobe.com/go/getflashplayer

[2] *Adobe Flash Player* Penetration statistic
http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

[3] Michael Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, November 2006.

[4] SWF and FLV File Format Specification License.
http://www.adobe.com/licensing/developer/fileformat/license/

[5] Matthias Junker, Markus Hofstetter. Scripting Diagrams with *EyeSee*.

[6] Macromedia Flash (SWF) and Flash Video (FLV) File Format Specification, Version 8
http://www.adobe.com/licensing/developer/