Web-based Sprite Sheet Animator for Sharing Programmatically Usable Animation Metadata

Xinze Lin
Abstract

**Web-based Sprite Sheet Animator for Sharing Programmatically Usable Animation Metadata**

*Xinze Lin*

In this project, we have developed a prototype application which is capable of creating and sharing programmatically usable sprite sheets via the web. At the same time, we also proposed a technique called Meta-pixel Enhanced Sprite Sheet which can enforce 2D game animation metadata to be always attached with its corresponding sprite sheet image. The project is dedicated to help 2D game programmers to quickly obtain programmatically usable raster graphics.
Contents

1 Introduction ........................................... 5
  1.1 Background ....................................... 5
    1.1.1 Status Quo of 2D Video Games ................. 5
    1.1.2 Problem with Obtaining 2D Game Animations .... 5
    1.1.3 Sprite Sheet Animator in A Nutshell ........... 6
    1.1.4 Meta-pixel Enhanced Sprite Sheet ............... 7
  1.2 Research Questions ................................ 7
  1.3 Goals ............................................... 7
  1.4 Delimitation ....................................... 7

2 2D Game Graphics and Animation Metadata .......... 9
  2.1 2D Game Graphics .................................. 9
    2.1.1 Game Graphics in General ....................... 9
    2.1.2 Vector Graphics vs Raster Graphics .............. 10
    2.1.3 Sprite Sheet .................................... 10
    2.1.4 How Sprite Sheet Works in Game ................ 12
    2.1.5 The Notion of Programmatically Usable ......... 14
    2.1.6 Sprite Sheet Communities ....................... 15
  2.2 Sprite Sheet Animation Metadata ................... 15
    2.2.1 Usage of Animation Metadata ..................... 15
    2.2.2 Positioning and Sequencing ...................... 16
    2.2.3 Metadata Representation ....................... 18

3 The Idea of Meta-pixel Enhanced Sprite Sheet .... 19
  3.1 Problem with The Conventional Way ................. 19
  3.2 MESS — Meta-pixel Enhanced Sprite Sheet .......... 20

4 A Brief Study on Sprite Sheet Animators ........... 22
  4.1 Usage of Sprite Sheet Animators .................... 22
  4.2 Related Work ....................................... 23
  4.3 Notable Features Of Existing Tools ................. 24
5 The Basics of Developing A Web Application

5.1 Overview ......................................................... 26
5.2 Client-side Technology ........................................ 26
  5.2.1 Web Browsers .............................................. 27
  5.2.2 RIA Frameworks .......................................... 28
  5.2.3 Client-side Scripting ...................................... 29
  5.2.4 Browser Supported Image Formats ....................... 30
5.3 Server-side Technology ....................................... 30
  5.3.1 Server-side Scripting ..................................... 30
  5.3.2 Web Servers for PHP ...................................... 31
  5.3.3 Data Store .................................................. 31

6 Design and Implementation of the Prototype ................. 33

6.1 Design Overview .............................................. 33
6.2 The Animator .................................................. 35
  6.2.1 Animator User Interface .................................. 35
  6.2.2 Involved Image Analysis Algorithms .................... 37
6.3 The Gallery ..................................................... 38
6.4 The Metadata Codec .......................................... 40

7 Experiments and Results ..................................... 43

7.1 Experiments ................................................... 43
  7.1.1 Animator Testing ......................................... 43
  7.1.2 Gallery Testing .......................................... 44
  7.1.3 Optimal Image Format for MESS ......................... 44
  7.1.4 File Size Impact on Applying MESS ..................... 44
  7.1.5 MESS Use Case Demo: Web-based Fighting Game .... 44
7.2 Results ......................................................... 45
  7.2.1 Animator Test Results ................................... 45
  7.2.2 Gallery Test Results ...................................... 45
  7.2.3 Optimal Image Format for MESS ......................... 45
  7.2.4 File Size Impact on Applying MESS ..................... 46
  7.2.5 MESS Use Case Demo: Web-based Fighting Game .... 46

8 Discussion ....................................................... 48

8.1 The Web-based Animator Prototype ........................ 48
  8.1.1 User Access Control .................................... 48
  8.1.2 Smart Features of the Animator ....................... 48
  8.1.3 The Sprite Sheet Gallery ................................. 49
8.2 Meta-pixel Enhanced Sprite Sheets .......................... 49
  8.2.1 Optimal Image Format for MESS ......................... 49
  8.2.2 File Size Impact on Applying MESS ..................... 49
Section 1

Introduction

1.1 Background

This section introduces the background of this master thesis project.

1.1.1 Status Quo of 2D Video Games

With the advent of graphics hardware technology and game engine development, photo-realistic 3D games have dominated the market during the past few years, 2D video games are now no longer as profitable as they once were among the digital entertainment market. However, 2D games are still an active topic among mobile game developers, browser game developers and hobbyist game programmers. Besides, in comparison with 3D games, 2D games are considered easier for novices to begin with, also, it can become a good practice for improving programming skills[1]. Therefore, the field of 2D game still has a bright future.

1.1.2 Problem with Obtaining 2D Game Animations

Despite the easiness in 2D game creation, every game programmer has to face a same challenge — the graphics. Creating pixel-based animations for 2D games is a time consuming process, not to mention it requires decent skills in digital arts. “Lacking of good arts always frustrates programmers to carry on their game projects”[2], a programmer without good arts support is less likely to finish her project alone. That is why 2D game projects today are normally done by a group of people rather than a single person.

Fortunately, with the help of Internet search engine services today, obtaining good looking game graphics is no longer an issue. By simply typing

---

1Including smart phones and hand-held game consoles.
2Video games that can be played on web browsers.
the search keyword “sprite sheet” at Google Images, one can obtain a thousand pages of 2D game animation templates. Moreover, there are several online communities hosting vast amounts of such templates categorized from different game genres. With these shared resources, programmers can now easily create game prototypes with good visual style by themselves. The only problem remaining is the fact that most of these animation templates are not programmatically usable, which means they cannot be used directly by any game engine, and programmers need to spend plenty of time and effort to ensure the animations can be correctly displayed in their own game frameworks. Luckily, there exists several tools to help dealing with such problems, these tools are known as sprite sheet animators.

1.1.3 Sprite Sheet Animator in A Nutshell

A sprite sheet animator is a small application that can make a sprite sheet programmatically usable, it commonly has two essential features:

1. The first feature is to provide users with a handy interface to perform positioning and sequencing on a sprite sheet. The data of positioning and sequencing forms up the animation metadata. In the context of this paper, we use the notion metadata to denote such data. In short, the first essential feature of a sprite sheet animator is to provide interface to create metadata.

2. The second feature is to allow users to export the created metadata, so that the data can be imported by some game engines to reconstruct correct animation data structure in their game runtime. In other words, a sprite sheet animator should allow users to obtain metadata in certain file formats.

In sum, the above two features can be generalized as metadata creation and metadata acquisition.

Existing sprite sheet animators only focus on enriching their feature as standalone desktop applications. Those features can help users to save time and efforts at the stage of metadata creation, however, there is another way to save even more, that is, skipping the metadata creation stage, and directly access to the metadata acquisition stage, which means allowing users to obtain the metadata if someone has already created it. Besides, most of the animators are platform dependent, and thus not accessible by every computer user.

---

3 A bitmap image storing 2D game animation sequences, see Section 2.1.3
4 An image search engine provided by Google.
5 Directly usable by computer programs, see Section 2.1.5
6 Data for reconstructing sprite sheet animations, see Section 2.2
In order to solve all these problems in one shot, we can build a web-based sprite sheet animator that is integrated with an online sprite sheet gallery. In this way, the animator not only has a high accessibility, but also encourages users to share animation metadata with each other.

1.1.4 Meta-pixel Enhanced Sprite Sheet

For most of the sprite sheet animators, the generated metadata for a sprite sheet is always stored in a separate file, which means the graphics and the metadata are separated. In this way, users always have to take care of both the metadata file and the graphics file in order to make them work together as a programmatically usable sprite sheet, and this is somewhat inconvenient for the users especially when they want to share their sprite sheets together with the metadata on the web. Therefore, in this project, apart from building a web-based sprite sheet animator, we also propose a technique called meta-pixel enhanced sprite sheet (MESS) which can enforce the metadata to be always attached with the sprite sheets. We anticipate it will be achieved by injecting the metadata into sprite sheet image files as bitmap pixels, so that a single bitmap file can carry both the graphics content and the metadata.

1.2 Research Questions

The project will answer the following research questions:

- How to implement a web-based sprite sheet animator that allows users to share programmatically usable sprite sheets?
- How to implement the proposed MESS technique to enforce metadata to attach along with sprite sheets?
- What is the optimal image format for achieving MESS, and What are the advantages and disadvantages when we are applying it?

1.3 Goals

In this project, the primary goal is to design and implement a web application that allows users to create and share programmatically usable game animations on the web. The secondary goal is to see if the proposed MESS technique is truly applicable on our web application.

1.4 Delimitation

Since developing a web application involves client-side and server-side, either side could cost a big effort to implement, therefore, in this project, we will
put more weights on client-side development over the server-side. Besides, in order to let us contribute to the project within a reasonable scope, the application might just be a working prototype instead of a full featured commercial grade application.
Section 2

2D Game Graphics and Animation Metadata

2.1 2D Game Graphics

A general introduction of 2D game graphics.

2.1.1 Game Graphics in General

In general, the graphics rendering technique used in a game can be categorized into three branches:

- **3D Graphics**
  3D graphics is the de facto graphics solution of the video game industry nowadays. By utilizing the power of the latest GPU\(^1\) technology, the gaming experience has come to a cinematic fashion, which is capable of making gamers fully immerse into a virtual reality. Creating graphics for 3D games requires decent skills in 3D modelling and skeletal animating. For beginners who wanted to learn game development, it could be a huge barrier to get themselves started.

- **2D Graphics**
  2D graphics can be categorized as vector graphics and raster graphics, both of them are easy to create. Also, creating animations for 2D graphics is very straightforward, this would be the easiest way for beginners to start with.

- **2.5D Graphics**
  2.5D graphics is also known as *isometric* graphics. It uses orthogonal projected graphics as 2D tiles to construct a 2D graphic game scene.

\(^1\)Graphics processing unit.
that driven by 3D game logic. The advantage of this technique over conventional 2D graphics is that, it allows gamers to sense the spatial depth of the game scene which can make gamers perceive the graphics as solid objects. However, creating isometric game art requires more work than conventional 2D graphics, since the artist need to create 4 or more orientations of an game object for each of its animation sequence.

2.1.2 Vector Graphics vs Raster Graphics

Vector graphics and raster graphics are two distinct techniques for displaying graphics in 2D games.

2D vector graphics, essentially a set of geometry rendering instructions, are easy for performing geometrical transformation like scaling and rotating, but difficult for artist to apply their regular painting techniques. However, it has a notable advantage over raster graphics. That is, it allows key-frame animations which can automatically interpolate transition frames between two distinct frames, this could save the game artist a great deal of efforts.

In contrast, raster graphics (known as bitmap) is a 2D array of coloured square dots called pixels. Drawing on pixels array is like painting, artists can easily perform their painting techniques to make complex visual effects. However, raster graphics has its own disadvantages: As Figure 2.1 has shown, when the original graphic gets zooming up in the context of vector and raster, the raster graphic to the right gets ragged at resizing when the vector graphic on the left still maintains the details. Besides, animation in raster is a bit tedious since the artist has to draw the graphics frame by frame without the aid of key-frame interpolation.

In recent game development, some developers use a hybrid approach, that is, using raster graphics as textures and using 2D vector algorithms to deform or transform the textures to present game graphics, so that the raster animation can utilize the key-frame interpolation that vector graphics had.

2.1.3 Sprite Sheet

A sprite is an interactive graphics object in a 2D game scene, it acts as a movable animation unit on the screen. For each static animation frame of a sprite, we call it a sprite object. A sprite sheet is a collection of sprite object sequence in a large bitmap. As Figure 2.2 shows, in (a), you see the basic sprite object zoomed by a scaling factor of 2. In (b), the sprite sheet, you see all possible variations of the graphic. And in (c), you see one of the variations entered into a scene.

3Game graphics from Super Mario Bros, copyrighted by Nintendo.
Figure 2.1: **Top:** Original star icon. **Left:** Zoomed in vector format. **Right:** Zoomed in raster format.

Figure 2.2: (a) A sprite object zoomed by 2x. (b) A sprite sheet example. (c) A sprite in a 2D game scene.
2.1.4 How Sprite Sheet Works in Game

- **Source Rectangle and Destination Rectangle**
  The extraction of a sprite object from sprite sheet to game scene is done via the `blit` operation. The operation has two required parameters, the _source rectangle_ and the _destination rectangle_. Both rectangles are consisted of four attributes (x, y, width and height) where the size (width and height) of both rectangle are identical. The source rectangle works on the sprite sheet when the destination rectangle works on the game scene. The source rectangle decides which sprite object to copy from; The destination rectangle decides which position shall the copied sprite object be pasted to. In Figure 2.3 the green rectangle on the left represents the source rectangle, the yellow one on the right represents the destination rectangle. The x and y attributes represents the top-left corner of the rectangles.

![Figure 2.3: Source rectangle(left) and destination rectangle(right)](image)

- **Sprite Animation**
  By changing the position of the source rectangle, one can animate the sprite object in game. For example, in Figure 2.4 the upper figure shows the source rectangle and its corresponding result of the current frame. The lower one shows the next frame, when the source rectangle move to another sprite, the character on the game scene change his movement correspondingly.
• **Misplacing The Source Rectangle**
  Misplacing the source rectangle can lead to unreasonable displaying of the graphics. As can be seen in Figure 2.5, the source rectangle is placed over two sprites on the sprite sheet, the game scene end up displaying two game characters in halves.

Thus, guiding the source rectangle to correct position with correct size is vital for displaying correct graphics.
2.1.5 The Notion of Programatically Usable

In order to guide the source rectangle in a correct and easy manner, people might align sprite objects into some regular patterns, such as grids.

As is shown in Figure 2.6 with grid-patterned alignment, each sprite object can now be easily extracted by a simple mathematical formula using a given ID number as argument. By switching the ID number at runtime, animations can be created, and this is what it means to be a programmatically usable sprite sheet.

Unfortunately, existing sprite sheet templates on the web are mostly arranged in irregular patterns. For example, in Figure 2.7 the uploader of the sprite sheet has put an effort to arrange the frames in a logical order, so that downloaders can have a clear overview of the animation sequence. However, his effort was not enough to make it programmatically usable, since the sprites are not aligned in a regular pattern.

---

4Game graphics from *Final Fantasy Tactics*, copyrighted by *Square Enix.*
2.1.6 Sprite Sheet Communities

There are several online communities where people collect and share sprite sheet templates. However, most of the sprite sheets are arranged in irregular patterns, and thus non-programmatically usable. In order to make these sprite sheets usable by third-party game engines, it is vital that they come with their animation metadata. However, animation metadata are not common in the communities since it has not been standardized, people have not yet a protocol to agree on.

2.2 Sprite Sheet Animation Metadata

This section talks about how animation metadata are used for representing game animations.

2.2.1 Usage of Animation Metadata

Animation metadata is used to guide the source rectangle on a sprite sheet. As is shown in Figure 2.8, the metadata is essentially an array of predefined rectangles where each rectangle represents an animation frame with a unique ID number.

With the metadata, sprite sheets arranged in irregular patterns have now become programmatically usable.

---

5Game graphics from Castlevania: Harmony of Dissonance, copyrighted by Konami Entertainment.
Figure 2.8: With the metadata, the source rectangle can now be easily guided by using a corresponding ID number.

Furthermore, as Figure 2.9 shows, the rectangles can be sorted into different groups according to the animation content they present, so that the animation sequences can make sense to game programmers.

2.2.2 Positioning and Sequencing

Positioning and sequencing are the two main process involved when creating animation metadata for a sprite sheet.

- **Positioning**
  Positioning is the process of defining metadata rectangles. A rectangle basically can be determined by two points, the start point and the end point. Conventionally, the start point is at the top left corner while the end point is at the bottom right corner. An example can be seen in Figure 2.10, the red circle on the top-left corner of the 7th frame represents the XY coordinate of start point, where the bottom right one represents the end point.
  
  Each rectangle represents an animation frame of a sprite object, if it is not placed carefully, it could lead to glitches as Figure 2.5 shows in earlier sections.

- **Sequencing**
  Sequencing is the process of adjusting playing order of the animation frames. If the animation frames are not arrange correctly, it could
Figure 2.9: Enrich the metadata by giving semantics to different groups of animation sequence.

Figure 2.10: Red circles indicating the start point and end point of a rectangle.
cause the sprite objects playing unsmooth animations or performing illogical actions.

2.2.3 Metadata Representation

For storing metadata in file systems, people usually export the metadata in forms of XML[8] or JSON[9]. These formats are both human readable and machine readable. They are specialized in presenting complex structural data, and are frequently used by many applications as a medium for data exchange. These formats are often used for storing animation metadata by existing sprite sheet tools.
The Idea of Meta-pixel Enhanced Sprite Sheet

In order to make an ordinary sprite sheet file programmatically usable, it is imperative that its metadata must always be alongside with it. Otherwise, computer programs will know nothing about the sprite sheet besides its image resolution and pixel colours. But in real world application, a sprite sheet and its metadata are conventionally stored in separate files, and that makes it inconvenient to enforce the metadata and the sprite sheet be always alongside with each other.

3.1 Problem with The Conventional Way

Here is an use case example of the conventional way when sharing a programmatically usable sprite sheet via the web:

1. Mario.png is an ordinary sprite sheet, PersonA load Mario.png into a sprite sheet animator and create animation metadata for Mario.png.

2. Then the metadata are exported as Mario.xml. No changes has been made to Mario.png. Now, both Mario.png and Mario.xml are on the same directory of PersonA’s local file system.

3. PersonA upload Mario.png and Mario.xml to the server.

4. PersonB sees Mario.png and Mario.xml on the server and download them to his/her local file system.

5. PersonB import Mario.png and Mario.xml from the local file system to a game engine, the game engine successfully reconstructs the game animations.
In the above use case, Step 1 and 2 are about metadata creation, Step 3 and 4 are about metadata distribution via the web. Step 5 is about how the metadata actually got used by game programmers.

In Step 3, when distributing a programmatically usable sprite sheet on the web, the distributor has to first ensure the PNG and XML file has the same file name “Mario”. Secondly, besides the Mario.png itself, the distributor has to ensure its metadata counterpart Mario.xml has been uploaded to the same server directory. But even if the distributor has done their part well, in step 4 the downloaders have to be very careful about not to omit either of the files, otherwise, they will not be able to reconstruct the animations in step 5.

3.2 MESS — Meta-pixel Enhanced Sprite Sheet

In order to make it easier for distributing programmatically usable sprite sheets via the web, we come up with an solution to enforce the metadata to be always attached with the sprite sheet.

In digital image steganography\(^\text{[10]}\), text messages are able to be stored inside images so that the messages can be hidden from third-party people in plain sight. This trick is achieved by converting the text to a binary stream and embed each bit to the least significant bits of the image pixels. Although the pixel colour are altered by the embed bit stream, the colour changes are too small to be noticed by the human eyes. In this way, only people who engages in this business knows how to extract these secret messages.

Inspired by this technique, similar idea can be applied to our sprite sheets. We can convert the animation metadata into a bit stream and embed it into the sprite sheet pixels. We can call these pixels Meta-pixels because they are sprite sheet pixels that carrying information for animating the sprite sheet itself. And since the sprite sheet is enhanced by these meta-pixels and gain the ability to reconstruct animations for itself, we can name this type of sprite sheet as Meta-pixel Enhanced Sprite Sheet to distinguish from ordinal sprite sheets. For convenience of use, we use its acronym MESS to denote the term.

After replacing the conventional way of distributing metadata with MESS, we can expect the previous use case will change to follows:

1. Mario.png is an ordinary sprite sheet, PersonA load Mario.png into a sprite sheet animator and create animation metadata for Mario.png. (No change)
2. Then the metadata are embedded into Mario.png. The Mario.png now has some random colored pixels prepended to the top and has its file size increased. Mario.png become a MESS. (Changed)

3. PersonA upload Mario.png to the server. (Changed)

4. PersonB sees Mario.png on the server and download it to her local file system. (Changed)

5. PersonB import Mario.png from the local file system to a game engine, the game engine successfully reconstructs the game animations. (Changed)

With this MESS solution, we no longer need to worry about the XML or JSON files. However, there could be potential practical problems in applying this solution, and this will be discussed in later chapters.
4. A Brief Study on Sprite Sheet Animators

4.1 Usage of Sprite Sheet Animators

As is mentioned in earlier sections, the two main features of a sprite sheet animator are creation and acquisition of the metadata. Here shows how a sprite sheet animator works. In Figure 4.1, a sprite sheet in PNG\(^1\) format is loaded into the animator application, the user create metadata for the this sprite sheet via the GUI, then export the metadata as an XML file. Now the XML file together with the original PNG file are programmatically usable by some game applications.

More details about creation and acquisition of metadata:

- **Metadata Creation**
  The metadata creation process mainly involves positioning, sequencing and grouping of the metadata rectangles. This process highly depends on the graphical user interface. The better the interface is designed the more efficient the creation progress might be.

- **Metadata Acquisition**
  When a user finishes creating the metadata, the user want to export the metadata from the application’s memory to the file system in some document format, so that the metadata can be read by third-party applications.

\(^1\)Stands for Portable Network Graphics.
4.2 Related Work

There are three existing software projects that have the similar purpose of this project, each of them supports creation and acquisition of metadata:

- **Sprite Buddy**
  *Sprite Buddy*\(^2\) is a freeware, it provides both online version and desktop version. The desktop version has only released for *Microsoft Windows* platform while the online version is accessible by any web browsers that supports *Flash*\(^2\) plugin.

- **Sprite Vortex**
  *Sprite Vortex*\(^2\) is yet another sprite sheet animator that can be downloaded free of charge. It is developed within *XNA*\(^3\) framework, it has an easy to use user interface, yet heavily depend on Microsoft’s *.Net* framework.

- **Dark Function**
  *Dark Function*\(^2\) is a commercial sprite sheet tool-set. It has the richest features among the three. Since it is developed in Java, it is accessible by multiple platforms.

\(^2\)Adobe’s 2D animation solution.
\(^3\)Microsoft’s game development framework.
Despite each of them has its own nice features, they are all sharing a common drawback, that is, they are all standalone applications which does not encourage users to share metadata with each other.

4.3 Notable Features Of Existing Tools

Based on our observation, the three tools mentioned in the previous section has the following features in common:

- **Sprite Object Scanner**
  In conventional image editors, when users want to select a sprite object on a sprite sheet, they need to carefully drag out a rectangular region to select all the pixels of the sprite. But this could be inconvenient at times due to users cannot precisely select their desired region without zooming the viewing panel. Therefore, some developers came up with an idea to let the computer assist with this task. As Figure 4.2 shows, by using an algorithm that scans through all the sprite sheet pixels line by line, the program will be able to identify all pixel clusters\(^4\) and create an axis-aligned bounding box for each cluster, this bounding box will then be regarded as a metadata rectangle. This allows users to handle the sprites in object-wise manner rather than pixel-wise manner, and therefore speed up the metadata creation process.

  ![Figure 4.2: Sprites can be automatically detected by a pixels scanning algorithm.](image)

- **Onion Skinning**
  Onion Skinning is a 2D graphics displaying technique, it allows several animation frames being displayed at one time. It is usually achieved by transparentizing each animation frame and overlay them against each other. An example can be seen in Figure 4.3 where three frames of sprite objects are blended into one.

\(^4\)Pixel cluster, a group of 4-way connected foreground pixels.
Onion skinning is a good way to show a short animation sequence, and it allows users to compare between the current frame and the previous frame, which is a great help for performing the positioning and sequencing task through the user interface. This feature helps metadata creation.

• Export to XML
  Despite each tool follows a distinct protocol on parsing and composing their animation metadata, all of them allow the metadata to be stored in forms of XML on the file system. This allows the metadata to be read by third-party program if the program has integrated the correct parser. This is a way of achieving metadata acquisition.
Section 5

The Basics of Developing A Web Application

5.1 Overview

A simple example of web application can be seen in Figure 5.1. As the diagram shows, the web interface (or web client) is rendered from HTML\(^1\) and CSS\(^2\) code. When a user performs an operation on the web interface, a callback routine will be invoked, and the corresponding Javascript code will be executed. Sometimes, some routines might involve communicating with the server side, such as when the user performs an upload operation, the Javascript routine will send an upload request to the web server. Then, in response to the request, a CGI\(^3\) script on the web server such as a script written in PHP\(^4\) will handle the upload request. The uploaded data will be extracted from the request and written to the data store by the PHP script. If the uploaded data has been successfully written, the PHP script will return a response of succeeded message to the browser side Javascript. Then the Javascript will update the web interface to show the upload operation has been successfully executed.

5.2 Client-side Technology

This section talks about some caveats for client-side development.

---

1 Abbreviation for Hypertext Transfer Markup Language
2 Abbreviation for Cascade Style Sheet.
3 Abbreviation for Common Gateway Interface.
4 A programming language commonly used for server side scripting.
5.2.1 Web Browsers

A web browser is a client side application, its usage is to display web contents as well as to allow users to interact with the web. In order to access a web application, a user needs a web browser to do that. There are various implementations of web browsers among the consumer market, most of them are free of charge. Since each implementation uses different approaches to follow the W3C\(^5\) standard, it leads to the so called cross-browser discrepancy which is a term to denote the phenomenon that the same client-side code being interpreted and presented differently across different browser implementations. Therefore, when starting to build something for the frontend, it is always a good idea to pick a target browser beforehand. To determine which browser is more worthy to start with, a general way is to analyse the browser market share.

According to recent studies\(^6\) statistics shows the market share of the top 4 browsers for the time being are Internet Explorer 28.6%, Chrome 28.3%, Firefox 23.1% and Safari 13.3%.

On top of that, by seeing the growing trend in Figure 5.2, the next dominant browser can be easily predicted. In that figure, it shows the Chrome browser has grown constantly over the years. With such growth rate, it is promising that it will soon become the top one in the market.

---

\(^5\) Abbreviation for World Wide Web Consortium.

\(^6\) Browser market share, source from: [http://bits.rahilparikh.me/2012/09/01/private-browsing-a-flawed-privacy/](http://bits.rahilparikh.me/2012/09/01/private-browsing-a-flawed-privacy/)

Browser market trend, source from: [http://www.w3counter.com/trends](http://www.w3counter.com/trends)
5.2.2 RIA Frameworks

RIA\(^8\) is a term to describe a web application that has many characteristics in common with conventional desktop applications. An RIA framework determines the way of developing and running the web client. It has been quite some time since front-end developers start to argue about what framework should be used for the web client. The debate were mostly focus on the comparison between HTML5 and Flash framework.

- **HTML5**

  HTML5 is one of the latest popular markup language for presenting web contents, it is supported by most prevalent browsers as a built-in feature. Its first working draft was published in 2008. Compare with its previous revisions, the current version introduced a canvas feature which enables web developers to access and manipulate image pixels. This new feature makes the raster graphics interface possible in our project.

  The HTML5 markup alone is not enough to make a web client, it needs the help of CSS to decorate the interface’s appearance and needs Javascript to control the client-side logic. Since this HTML5/CSS/Javascript framework is supported by most modern browsers by default, it does not require extra browser plugin to run it, thus many developers are in favour of it.

---

\(^8\)Rich Internet Application.
• Adobe Flash
Flash is owned by Adobe, it has been popular as an RIA solution for years, especially under the help of the Flex SDK. Over the years, it has a bunch of improvements in terms of development tool-kit support as well as runtime performance. Compare with HTML5, it is a more mature and dedicated RIA solution to work with. However, in order to run it on a browser, it requires the user to install the Flash Player plugin. The problem with this plugin mechanism is that, once the company stops developing or upgrading the plugin for a certain platform, that platform might no longer be able to access the newly web application developed in Flash framework.

• Other Alternatives
Besides Flash, there are many other alternatives for presenting the web client, such as Microsoft’s Silverlight and Oracle’s JavaFX, but each of them are like Flash, they all require a plugin to run on a web browser.

5.2.3 Client-side Scripting
The client-side script plays a key role on providing great interactivity. The scripting language on mainly depends on what RIA framework is chosen. As is mentioned earlier, Javascript is the default scripting language on all major web browsers.

• Javascript
Every major browser has a built-in Javascript interpreter, therefore Javascript can be executed on any of them. The Javascript itself is a event driven language, which allows each user request be responded independently. Furthermore, it allows programmers to access and manipulate the DOM, which enables programmers to partially update a web page without refreshing the entire one, the famous Ajax technique is also depending on this feature.

• Alternatives
Besides Javascript, there are also other higher level language such as Coffeescript and Haxe that provides further fancy features beyond standard Javascript, these language can be later compile down to Javascript for their execution on browsers.

Besides, if the programmer has chosen a plugin-based RIA framework in the following Table 5.1, the client-side scripting language will totally refrain from any Javascript involvement:

---

9 Apache Flex software development kit, initially released in 2004.
10 Document Object Model.
11 Asynchronous Javascript and XML.
Table 5.1: Scripting language for other frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Flash</td>
<td>Actionscript</td>
</tr>
<tr>
<td>Microsoft Silverlight</td>
<td>C#</td>
</tr>
<tr>
<td>Oracle JavaFX</td>
<td>Java</td>
</tr>
</tbody>
</table>

5.2.4 Browser Supported Image Formats

Modern web browsers will generally support the following image formats:

- **BMP**
  Image stored with BMP is usually with large size due to its uncompressed coding, few people will actually use this format to display graphics on the web. However, since it is lossless, this format is suitable for storing sprite sheets if the file size is not the issue.

- **GIF**
  GIF is short for graphics interchange format, it is often used for displaying non-interactive animations on web pages. It has a high compression ratio, but has a limit of 256 colour in its palette and thus inadequate for presenting photo-realistic images. This format is good enough for storing sprite sheets as long as the total usage of colour does not exceed 256.

- **PNG**
  The Portable Network Graphics (PNG) is a lossless format, it has a high compression ratio as good as GIF. Furthermore, since it does not have any colour limitations on its palette, that makes it excels GIF for storing 2D game sprites.

- **JPEG**
  JPEG stands for Joint Picture Expert Group, it has a higher compression ratio than any other browser image formats. It is often used for presenting realistic photographs. However, since it uses a lossy compression algorithm that degrades the image quality, it is not a suitable format for storing sprite sheets.

5.3 Server-side Technology

This section talks about some basic components of the server-side.

5.3.1 Server-side Scripting

Server-side scripts are responsible for handling requests from client-side and performing corresponding subroutines. The language for scripting depends
on what interpreter is installed on the server. In today’s Internet, the most widely used scripting language for the server-side is PHP\(^\text{12}\).

- **PHP**
  PHP is designed for generating dynamic web pages. Since it is used by a lot of people today, it has a lot of well organized tutorials on the Internet. Besides, there are lots of application development framework written in PHP, by using those frameworks, one can speed up the development process dramatically.

- **Alternatives**
  Besides PHP, there are other scripting languages also being used for server-side scripting such as ASP.NET\(^\text{13}\) JSP\(^\text{14}\) Perl, Ruby, Python and Erlang.

### 5.3.2 Web Servers for PHP

The Apache\(^\text{15}\) server has been used by most of the PHP web sites due to its easiness for installation and configuration. However, in recent years, another web server called Nginx\(^\text{16}\) has gradually become a replacement for Apache server due to its superior runtime performance.

### 5.3.3 Data Store

Database and file system are the two major ways for storing web application data:

- **Database**
  Commonly, a PHP application will use a MySQL\(^\text{17}\) server for its data storage since PHP and MySQL are usually attached together as a software bundle for kick starting a project. However, when the database backend is distributed, and scalability becomes the key requirement of the web application, developers might turn to NoSQL\(^\text{18}\) databases instead.

  The advantage of using a database for storing data is that, it allows programmers to perform advanced operations on the stored data via query languages.

---

\(^{12}\)PHP hypertext processor, a scripting language for generating dynamic web pages.
\(^{13}\)Microsoft’s active server pages dot net.
\(^{14}\)Java server pages.
\(^{15}\)Apache web server.
\(^{16}\)Nginx web server.
\(^{17}\)MySQL, an open source relational database management system.
\(^{18}\)NoSQL, a broad class of key-value store distributed databases.
• **File System**

The file system is a means to organize data on a storage device\(^{19}\) so that computer programs can read and write data conveniently. There exists lots of formats of file systems, the most famous one nowadays is NTFS which is used by Microsoft Windows OS. PHP scripts have the ability to read and write data to file systems. Compare with the database approach, the advantage of using file system is that, it reads and writes faster when the application is dealing with large files.

\(^{19}\)Such as a hard-drive, a USB flash drive or even a tape drive.
Design and Implementation of the Prototype

6.1 Design Overview

The basic work-flow of our prototype can be seen in Figure 6.1. In this figure, the white file on the bottom-left corner represents an ordinary sprite sheet. After going through several steps of process, the ordinary sprite sheet becomes the grey one which indicates a MESS file.

From users’ perspective, the entire application can be divided into three parts, the animator, the gallery and the metadata codec:

- **Animator**
  The animator provides a user interface for creating metadata for an ordinary sprite sheet, as well as for modifying existing metadata. After the metadata is created, it will be written into the original sprite sheet via the built-in metadata codec. Then the MESS file can be either saved to the user’s local file system or uploaded to the server.

- **Gallery**
  The gallery provides an interface for browsing all MESS files that are hosted on the server, it is a platform for users to share programmatically usable sprite sheets with each other.

- **Metadata Codec**
  The metadata codec has two functionalities, encoding and decoding. The encoding function is to write animation metadata into a ordinary sprite sheet and make it a MESS file. On the other hand, the decoding function is to extract the animation metadata from a MESS file and turn the metadata into a native data structure for the calling program.
Figure 6.1: An overview of our prototype
In our prototype, the metadata codec is integrated in two places. One is in the animator, its purpose is to read and write MESS files for the animator. The other one is in the user’s game framework, it allows user’s game engine to extract animation metadata from MESS files.

6.2 The Animator

The animator is implemented by in-browser Javascript together with HTML5 and CSS. Our client-side code is targeted on Google Chrome since it utilizes the V8 Javascript engine\(^1\), which makes its Javascript performance excels other browsers.

6.2.1 Animator User Interface

As is shown in Figure 6.2, the user interface mainly consists of one menu bar and four panels:

1. **Sheet Panel**
   
The sheet panel displays the entire sprite sheet, as Figure 6.3 illustrates, it allows users to select a sequence of sprites by simply dragging a scan-line vector over them. As soon as the user release the

\(^{1}\)V8 Javascript engine, it compiles Javascript into native machine code for execution.
mouse button, an axis-aligned bounding box will be generated for the
selected sprites respectively, and each sprite will be added to the layer
panel as an animation frame. The bounding boxes are interactive, the
users can actually click on them to focus on an individual sprite. The
color of the bounding boxes help users to distinguish different groups
of animation sequence. With this sheet panel, the selection of sprites
becomes more intuitive.

![Before](image1)

![After](image2)

Figure 6.3: Scan-line and interactive bounding boxes

2. **Animation and Zoom Panel**

   The animation panel displays the graphics for the currently focused
   sprite. By switching the focus along a sprite sequence in a consecu-
   tive manner, the focused sprite sequence can be animated within the
   panel window. Besides, the animation panel allows the user to realign
   the current sprite’s position via the keyboard. animation panel also
   supports onion skinning The zoom panel is the same as the animation
   panel except that it doubles the size of the original graphics, it can
   help the user to see the pixels more precisely when they are realigning
   the sprite’s position.

3. **Layer Panel**

   The layer panel mimics the timeline controller of the Adobe Flash
toolkit(see Figure 6.4). It gives the freedom to manipulate the dis-
playing order of an animation frame sequence. In this prototype, the
layer panel only allows users to insert or delete animation frames, it
does not really support features such as multi-layer and drag-and-drop
rearrangement as Flash does.
4. **Action Panel**
   In the animator, each animation sequence represents an action sequence of the game character. The action panel allows users to select an action sequence by an action name.

5. **Menu Bar**
   The reason of having the menu bar is just to follow the convention of desktop applications. The menu bar provides drop down menus for accessing basic commands of the animator.

### 6.2.2 Involved Image Analysis Algorithms

The sprite object scanner feature on the sheet panel has involved two image analysis processes:

1. **Automatic Detection of Background Colour**
   Before the sprite object scanning algorithm can be applied to the sprite sheet, we need to separate the sprites from the background, that is, to tell the background pixels apart from the sprite pixels. The most straightforward way to achieve this is to iterate through all the image pixels and find the biggest pixel cluster, the colour of this cluster is likely to be the background colour of the sprite sheet.

   However, scanning through all the image pixels is a computation-intensive process. In order to save performance on the client-side in the light of user experience, we chose a more lightweight approach — just sample a few pixels on the sprite sheet and make a colour histogram, the colour with the highest rating should be the background colour. The sampling points are scattered at all four corners of the sprite sheet,
this is based on our empirical observation that most sprite sheets have their background colour on their corners.

Unfortunately, there is still a slight chance the lightweight detection algorithm could fail. In that case, the application shall as well offer users a contingency way to determine the background colour manually.

2. Sprite Object Scanner

After separating background pixels from sprite pixels, we can now treat the entire sprite sheet as a binary image. In this image, we label sprite pixels with white colour and background pixels with black. The objective of the sprite object scanner algorithm is to compute bounding boxes for those white ones. The most straightforward way to get the bounding box of a pixel cluster is to perform a flood fill search to find out the boundaries. However, iterate through all the pixels is not really necessary for finding the boundaries, simply iterate through all the pixels on the sprite contour will be enough to get the bounding box. The algorithm for judging if a pixel is a contour pixel is as follows:

\[ \text{neighbours}(P) \rightarrow \{P(-1,0), P(1,0), P(0,-1), P(0,1)\} \]

\[ \text{isContourPixel}(P) \rightarrow \exists(\text{neighbours}(P) \in \text{BlackPixels}) \]

As Figure 6.5 shows, the scanner algorithm scans through each pixel from the start point towards the end point along the scan-line. Once the scanning point hits a contour pixel of a sprite object, it stops following the scan-line and begin to follow the contour instead. By travelling along the contour, the algorithm is able to collect the left-most, right-most, top-most and bottom-most pixel coordinates of the sprite object. These four coordinates form up a rectangle and become the bounding box of the sprite object.

6.3 The Gallery

The purpose of this gallery is to demonstrate how the MESS files can be shared on the web. In the current iteration of our prototype, the gallery merely supports very basic features such as uploading, downloading and deleting of the sprite sheets. As is shown in Figure 6.6, the its interface only displays a list of thumbnails and a single sprite sheet.

The gallery itself is developed based on a PHP web application framework named CodeIgniter. The framework has a MVC structure, the main

\(^2\)A recursive algorithm which loops through each connected pixel that has the same colour.

\(^3\)Model-view-controller software architecture.
Figure 6.5: The scanning detail of the sprite object scanner

Figure 6.6: The gallery user interface
reason of choosing it for our project is due to its popularity and easy learning curve.

In order to speed up the development process even more, we used some gallery project template for CodeIgniter as our baseline. We choose Nginx as our web server due to its superior performance among other PHP servers. Meanwhile, since the prototype gallery is not designed to support database searching features, we use file system as our data store to evade the cost of effort for setting up a database.

6.4 The Metadata Codec

The codec is used for decoding and encoding metadata into meta-pixels. Normally, in a bitmap canvas, a pixel has RGBA four channels, each channel can hold an 8-bit value ranging from 0 to 255. Considering some image applications might ignore the support of alpha channel, therefore we skip the alpha channel and only use the RGB channels to carry the serialized metadata, this means each pixel can store 24 bits of data. Take Figure 6.7 for example, the sprite sheet in this figure is prepended with a line of meta-pixels on top. The first pixel on the top line has an RGB value of (115, 112, 99) which is an ASCII representation of the trigram “spc”. When reading a sprite sheet, the trigram “spc” here act as a file signature to tell the codec whether there is any meta-pixel to be parsed as animation metadata within this sprite sheet. If the sprite sheet has no such “spc” in its first pixel, the codec will not apply any decoding process to it. Similar to the first pixel, the rest of the meta-pixels are containing various attributes of the metadata, such as action sequence and metadata rectangles. The coding scheme for the meta-pixels is shown in Figure 6.8. From what we can see in the diagram, each block represents a meta-pixel:

- The 1st pixel is the signature block, as we mention earlier, the trigram “spc” is placed in this block.
- The 2nd pixel is the genre block, it indicates which game genre this sprite sheet belongs to, it might be useful for categorizing the sprite sheets in the future. But for now, this block has no any actually usage, just consider it as a preserved block.
- The 3rd pixel is the actions block, it stores the total number of actions so that the codec knows how many action sections are going to be parsed. This block can store an integer ranging from 0 to 16777215 which means this coding scheme can support up to 16777215 sets of action.

4Original code of our gallery can be found at: http://superdit.com/2010/06/27/basic-image-gallery-with-codeigniter/
5Red, green, blue and alpha colour channels, alpha is for transparency.
• From 4th to 7th pixel, the 4 of them form up an *action name block* which can hold at most 12 ASCII characters for an action name. This block indicates the beginning of a new *action section*.

• The 8th pixel is a *frames block* which carries the total number of frames for the current *action section*. The maximum number this block can hold is also 16777215 which means an *action section* can at most have 16777215 animation frames.

• From 9th to 11th pixel, the 3 of them form up a *metadata rectangle block* represents a metadata rectangle for an animation frame. As the lower part of the diagram shows, each metadata rectangle has 6 sub-attributes, X, Y, Width, Height, OffsetX and OffsetY. The 72 bits of the entire block are divided among these sub-attributes.

With this coding scheme, the animator can use the codec to write meta-pixels to sprite sheets, and game engines can use the codec to parse meta-pixels from the MESS files.

The example above is just one of the many ways to design a metadata codec. By employing a different coding scheme, one can let the codec has different capabilities.
Figure 6.8: Meta-pixel Coding Scheme
7 Experiments and Results

7.1 Experiments

The test environment of our prototype is a laptop computer which has a Intel Core i3 CPU in 2.27GHz frequency. The operating system installed is Ubuntu Linux 12.04. The client-side HTML5 and Javascript code are running on Chrome 21 web browser. Meanwhile, the server-side PHP code are running on Nginx 1.3.6 web server.

7.1.1 Animator Testing

The idea behind this test is to make sure the animator can create and acquire the metadata correctly. The testing steps are as follows:

1. Open the animator interface and load a PNG sprite sheet from the local file system. If it works correctly, the sprite sheet should be displayed in the sheet panel.

2. For each row in the sheet panel, create an action in the action panel and drag a scan-line to scan through each sprite object on the row. If the scanner is functioning properly, it should generate bounding boxes for each sprite object, and there will be three actions created in the action panel.

3. Export the sprite sheet to the local file system as a MESS in PNG format. If the encoding function of the codec is working, the PNG image should have a line of meta-pixels prepended on the top.

4. Restart the animator and load the meta-pixel enhanced PNG file into it. If the entire codec is implemented correctly, the state of the animator should be exactly the same as the state after performing step 2.
7.1.2 Gallery Testing

The gallery test is to confirm if the server-side can store and display sprite sheets correctly.

1. Open the animator and load a MESS file from the local file system, then upload it to the gallery server.

2. Open the gallery interface, if the server-side scripts work correctly, the uploaded sprite sheet should be able to be seen in the gallery.

7.1.3 Optimal Image Format for MESS

As for answering one of our research questions, here we create some proves for later discussion.

1. Export a MESS file as PNG, BMP, GIF and JPG format respectively, then compare their file size.

2. Import each file back to the animator and see if the metadata can be reconstructed correctly.

7.1.4 File Size Impact on Applying MESS

Try to compare the difference between before and after applying MESS in terms of file size.

1. Download 16 ordinary sprite sheets from an online community and use them as our test samples.

2. Turn each sprite sheet into a MESS file using the animator.

3. Compare the file size of the MESS files with their originals. Each file from the originals is including its metadata stored in a separate JSON file.

7.1.5 MESS Use Case Demo: Web-based Fighting Game

In order to demonstrate how MESS can be used by third party game engines, we made a simple web-based multi-player online fighting game engine which allows tens of users to play simultaneously in a synchronous fashion. The frontend of the game engine is implemented by HTML5 canvas and Javascript. The real-time network communication is achieved by Websocket and JSON. The backend socket server is using Python Twisted library. The test steps are as follows:

1. Create metadata for two fighting game characters in the in the animator and export them as MESS.
2. Open two browsers, each of them go to the game frontend that connected to the socket server. The two MESS files will be loaded by each frontend and displayed their corresponding characters in the game scene.

3. Move the game characters around in one of the browsers.

This demo is aimed to give people a small picture of what MESS can do in real world production.

7.2 Results

This section shows the outcome of each testing set.

7.2.1 Animator Test Results

1. The loading is working properly, the sprite sheet is displayed in the sheet panel upon loading.

2. The sprite scanner is working properly, each sprite object has its bounding box around it.

3. After downloading the sprite sheet as MESS in PNG format, the meta-pixels are visible on its top line.

4. After restarting the animator and loading the MESS file we just exported, the animator recovers all the actions and bounding boxes we added before. That proves the entire codec is functioning properly.

7.2.2 Gallery Test Results

1. A MESS file is successfully loaded from the local file system to the animator and upload command is triggered.

2. After opening the gallery interface, we can see a thumbnail of the uploaded MESS file. By clicking on the thumbnail, the sprite sheet is enlarged, and we can see the entire sprite sheet displayed on the interface. That means the server-side scripts are working properly.

7.2.3 Optimal Image Format for MESS

1. The file size in each format are as follows:

2. After trying to import each file back to the animator, it turns out that only PNG and BMP can successfully reconstruct the metadata. The GIF fails to reconstruct the metadata and JPG fails to reconstruct both the original graphics and the metadata.
Table 7.1: MESS file size for browser supported image formats.

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Original</th>
<th>PNG</th>
<th>JPG</th>
<th>BMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIF</td>
<td>23 kb</td>
<td>43 kb</td>
<td>81 kb</td>
<td>773 kb</td>
</tr>
<tr>
<td>Original</td>
<td>37 kb</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.2.4 File Size Impact on Applying MESS

The details of the test results can be seen in Appendix B. In the result data, the growth ratio at the worst case is 246.45%, the best case is 102.61%, the median and average case are 160.02% and 158.92% respectively. A comparison between the MESS files and their originals is shown in Figure 7.1. We sort the results based on the file size of the original PNG files in an ascending order. Furthermore, the growth ratio of each sample can be seen in Figure 7.2. The growth ratio here is calculated by the MESS file size dividing the original file size.

Figure 7.1: Before and after applying MESS

7.2.5 MESS Use Case Demo: Web-based Fighting Game

As is shown in Figure 7.3, the fighter data that are carried by the MESS files can be loaded and displayed in our fighting game engine. And the fighters in both browsers are moving synchronously.
Figure 7.2: The growth ratio of file size

Figure 7.3: MESS fighting game demo, fighter sprites from *Bare Knuckle III* copyrighted by Sega.
Discussion

8.1 The Web-based Animator Prototype

From the test result of our application prototype, we have confirmed all desired features of the application are working properly. The animator can import and export MESS files and the gallery can host the MESS files on the web. Here we discuss some notable issues of our prototype.

8.1.1 User Access Control

Since the application is still at prototype stage, we have not yet implemented user access control, which means any user can delete whatever sprite sheets hosted on the server without registering or logging in an user account, and this leaves a big threat in our data security. Therefore, before we could use it to set up an online community, we first need to finish the user access control feature to eliminate the threat.

8.1.2 Smart Features of the Animator

We tried to make the animator automate its tasks as much as possible by adding smart features to it. Techniques like background detector and sprite sheet scanner really save a lot of work for the users. So far, we are well aware of there is at least one part of our animator interface could have been improved. That is the realignment between consecutive frames in the animation panel. Currently, we allow users to realign the frames manually by using WASD keystrokes, however, this feature could have been replace by an automatic approach that is using machine vision algorithm like feature point matching, so users no longer need to realign the frames manually. But considering implementing such feature might be a bit out of our project scope as developing such feature could have been another project on its own, we decide to leave it the way it is for now.
8.1.3 The Sprite Sheet Gallery

At current stage, the gallery only display a list of thumbnails for the available sprite sheets regardless of what genre they belong to. For a more mature gallery, it could have have been supported search feature based on game genre or game series. However, in this project, those features are of the lowest priorities since we only aim to deliver a prototype.

8.2 Meta-pixel Enhanced Sprite Sheets

In this section, we will talk about the the optimal image format as well as the drawback when applying MESS.

8.2.1 Optimal Image Format for MESS

From the result of our experiment, we have observed that JPG and GIF failed to reconstruct the metadata via the decoder, this is because:

- **JPG**
  
  Since JPG uses *discrete cosine transform* to approximate the original image to gain high compression ratio, the meta-pixels will be disrupted by the approximation as soon as the image is converted to this format. Since the value of each pixel has been altered, the decoder was not even able to recognize the “spc” signature. As a conclusion, lossy compression image format like JPG is not applicable for MESS.

- **GIF**
  
  GIF in our test case has the best compression ratio over others. However, since it has a limitation of 256 distinct colours on its palette, as the colour exceed that limit, its algorithm will perform some colour reduction to make sure the colour limit will not be exceeded. And as a result, some colours in the meta-pixel region have been compromised and lead to the loss of our metadata. As a conclusion, GIF is not suitable for MESS either.

Thus, BMP and PNG are so far the most MESS-compliant image formats that are available on web-browsers. But since BMP is an image format with almost no compression, as our test result shows, for the same sprite sheet, BMP is taking 773kb while PNG is only taking 43kb, the superiority of PNG is obvious. Therefore, we can say, the PNG format is so far the best image format for applying MESS.

8.2.2 File Size Impact on Applying MESS

The result of the file size testing shows our proposed MESS technique could cause a considerable penalty on PNG compression ratios. From Figure 7.1...
we see for most of the time, the MESS will not cause much compression penalty on the PNG file, but in some special case, such as sample 12, it doubled the size of the PNG file. We try to find out if the compression penalty has a linear relationship with the original file size, but the random bumps of the growth curve in Figure 7.2 indicate that it has nothing to do with the original file size, otherwise, it should at least had an observable growing trend. For a more general understanding of the nature of MESS in terms of file size impact, we have done some simple statistics: Among the all test samples, the worst case enlarges the file size up to 2.5 times of the original. For average case, it enlarges the file size by a factor of 1.6. That is to say, a trade-off has to be made when we are applying MESS to the PNG files.
Conclusion and Future Work

9.1 Conclusion

By gathering everything we have covered so far, we are now able to answer our research questions:

- How to implement a web-based sprite sheet animator that allows users to share programmatically usable sprite sheets?

  **Answer:** As our implementation showed, we used a stack of HTML5, Javascript, PHP, Nginx and Linux. Where HTML5 and Javascript are used for creating the animator and library at the frontend, PHP was used as server-side scripting language to handle the frontend request and backend file storage.

- How to implement the proposed MESS technique to enforce metadata to attach along with sprite sheets?

  **Answer:** It can be achieved by encoding metadata in forms of image pixels and paste these pixels into the sprite sheet that is in lossless image format. It requires a careful design of coding scheme which ensures the metadata to be encoded and decoded correctly.

- What is the optimal image format for achieving MESS, and What are the advantages and disadvantages when we are applying it?

  **Answer:** So far, as we have retrieved evidence from our experiments on all available browser supported images, the result showed PNG is the best candidate for applying MESS. The advantage of MESS is that, it allows existing lossless image format to carry extra metadata that can be used to reconstruct animations. The disadvantage is that the compression ratio might suffer a random degradation.
In this project, we have developed a prototype application which is capable of creating and sharing programmatically usable sprite sheets on the Internet. It also successfully demonstrated our proposed MESS technique which can enforce metadata and sprite sheet to always attach together. 2D game hobbyists might now have a more effective platform to obtain graphical resources for their game programming.

9.2 Future Work

Since the application is still at its prototype stage, in its design, we might have omitted a lot of practical problems in real world applications. Therefore, we will need to do further study to figure out what features we truly need to make it merits a real world application. And based on the study, we might reconsider the design for better user experience.

Apart from improving user experience, it is possible that we will integrate some fancy features that coming from common game engines, as our project has a great potential to be developed into a self-contained 2D game rapid prototyping environment:

- **Sprite Sheet Search Engine**
  Enables users to search sprite sheets by declaring searching constraints either within our web site or outside the site. This would be of the highest priority in the feature list.

- **Level Editor**
  The animator shall not only supports making sprite animations for game characters, but also support making game scenes for game characters to hop in.

- **Game Physics Support**
  Support composite collision proxies\(^1\) in the sprite sheet metadata, so that the game character can perform more precise collision detection in the game environment.

- **Game AI Scripting Environment**
  Provide a visual scripting environment for developing artificial intelligence of the game characters. This will allow a MESS file to carry a completely autonomous game character.

- **Game Character Sandbox**
  This feature will allow putting several autonomous game characters within a virtual arena to see how their AI behave in game.

This integration might come in handy for those who already know 2D game programming and those who are on their way to 2D game programming.

\(^1\)Collision proxies are non-rendered geometries for performing collision detection.
References


Acknowledgements

I would like to thank my supervisor Anders and reviewer Lars for helping me to improve my thesis writing. At the same time, I’m really grateful they gave me the time and chance to carry out my very own ideas.

During the project, due to my own curiosity, I’ve been trying out various other programming languages such as Python, Ruby, Perl, NodeJS, CoffeeScript, Erlang, Clojure, Haskell, Lua to write web applications. This gives me an overall understanding of how and why people are using these dynamic languages or frameworks to make web applications. In addition, during my study of web 2.0 applications for the project, I’ve learned the concept of NLP\(^2\) web crawling, semantic web, ontology and cognitive surplus of the netizens, it gives me a big picture of how the web could become more intelligent and how it could be used to bring more convenient services to our daily life. This thesis project will be a great influence to my choice of career in the coming future.

\(^2\)Natural language processing
Appendix A

Metadata Codec Source Code

```javascript
function Codec() {
    var sheetObj;
    var sheetCanv;
    var header = "spc";
    var idx = 0;
    var pixIdx = 0;
    var frameIdx = 0;
    var headerLen = header.length;
    var canv = document.createElement('canvas');
    var outCanv = document.createElement('canvas');
    var lengths;
    var ctx;
    var outCtx;
    var data;
    var numToChar = String.fromCharCode;

    function encode() {
        var len, act;

        lengths = getActionsLength();
        canv.width = sheetCanv.width;
        canv.height = Math.ceil(lengths[0]/canv.width);
        ctx = canv.getContext('2d');
        ctx.fillStyle = '#6c6';
        ctx.fillRect(0, 0, canv.width, canv.height);
        var metaImgData = ctx.getImageData(0, 0, canv.width, canv.height);
        data = metaImgData.data;

        idx = 0;
        pixIdx = 0;
        setChars(header);
        setChars('plt');
        setActionsNum(lengths[1]);
        for (var j in actions) {

        }
```

act = actions[j];
len = act.length;
setActionSection(j);
setActionsNum(len);
for (var i = 0; i < len; i++) {
  setRect(act[i][0]);
}
}
ctx.putImageData(metaImgData,0,0);
}

function decode(sheetObj) {
  var sheetCanv = sheetObj.imgCanv;
  var sheetImg = sheetObj.img;
  idx=0;
pixIdx=0;
  ctx = sheetCanv.getContext('2d');
  var metaImgData = ctx.getImageData(0,0,sheetCanv.width,sheetCanv.height);
  data = metaImgData.data;
  var hd = getChars();
  var out, actLen, actName, arr, aNum, r;
  if(header!==hd){// err no header
    return;
  }
  var type = getChars();
  out={};
  actLen=getActionsNum();
  for (var j = 0; j < actLen; j++) {
    actName = getActionSection();
    arr = [];
    out[actName] = arr;
    aNum = getActionsNum()
    for (var i = 0; i < aNum; i++) {
      arr.push([getRect()]);
    }
  }
  actions = out;
  for(var i in actions){
    i = i;
    curAct = i;
    sheetObj.treeAddItem(i);
    sheetObj.listToSvg(actions[i]);
  }
  lengths = getActionsLength();
  var w = sheetCanv.width;
  var h = Math.ceil(lengths[0]/w);
  ctx.clearRect(0,0,w,canv.height);
  ctx.drawImage(sheetImg,0,0);
  return out;
}
function setRect (r) {//3 pixels
    var x = r[0],
        y = r[1],
        w = r[2],
        h = r[3],
        ox = r[4],
        oy = r[5],
        hx = x>>8,
        lx = x&255,
        hy = y>>8,
        ly = y&255,
        hw = w>>2,
        lw = w&3,
        hh = h>>4,
        lh = h&15,
        hox = ox>>6,
        lox = ox&63,
        hoy = oy>>8,
        loy = oy&255;

    data[idx] = hx;
    data[idx+1] = lx;
    data[idx+2] = hy;

    data[idx+4] = ly;
    data[idx+5] = hw;
    data[idx+6] = (lw<<6)|hh;

    data[idx+8] = (lh<<4)|hox;
    data[idx+9] = (lox<<2)|hoy;
    data[idx+10] = loy;

    pixIdx+=3;
    idx = pixIdx<<2;
}

function getRect () {
    var hx = data[idx],
        lx = data[idx+1],
        hy = data[idx+2],

        ly = data[idx+4],
        hw = data[idx+5],
        lw = data[idx+6]>>6,
        hh = data[idx+6]&63,

        lh = data[idx+8]>>4,
        hox= data[idx+8]&15,
        lox= data[idx+9]>>2,
        hoy= data[idx+9]&3,
        loy= data[idx+10],

        x = (hx<<8)|lx,
y = (hy << 8) | ly,
w = (hw << 2) | lw,
h = (hh << 4) | lh,
ox= (hox << 6) | lox,
oy = (hoy << 8) | loy;
pixIdx += 3;
idx = pixIdx << 2;
return [x, y, w, h, ox, oy];
}

function setActionSection(name) { // 4 pixel
    for (var i = 0; i < 3; i++) {
        data[idx + i] = name.charCodeAt(i);
        data[idx + 4 + i] = name.charCodeAt(i + 3);
        data[idx + 8 + i] = name.charCodeAt(i + 6);
        data[idx + 12 + i] = name.charCodeAt(i + 9);
    }
pixIdx += 4;
    idx = pixIdx << 2;
}

function getActionSection() {
    var out = '';
    var end = idx + 16;
    for (var i = idx; i < end; i += 4) {
        var k = data[i];
        if (k === 0) break;
        out += numToChar(k);
        k = data[i + 1];
        if (k === 0) break;
        out += numToChar(k);
        k = data[i + 2];
        if (k === 0) break;
        out += numToChar(k);
    }
pixIdx += 4;
    idx = pixIdx << 2;
    return out;
}

function getActionsLength(actions) {
    var pixlen = 2;
    var actionNum = 0;
    for (var i in actions) {
        pixlen += 5; // name + len
        pixlen += actions[i].length * 3; // rect
        actionNum ++;
    }
    return [pixlen, actionNum];
}

var getFrameIdx = function () {
    return frameIdx;
}
var checkHeader = function (input) {
    return input === header;
}

var setChars = function (chars) {//first pixel carries the header
    for (var i = 0; i < 3; i++) {
        data[idx+i] = chars.charCodeAt(i);
    }
    pixIdx++;
    idx = pixIdx<<2;
}

var setActionsNum = function (len) {//1 pixel
    var a = len >> 16,
    b = (len >> 8) - (a << 8);
    c = len - (a << 16) - (b << 8);
    data[idx] = a;
    data[idx + 1] = b;
    data[idx + 2] = c;
    pixIdx++;
    idx = pixIdx<<2;
}

var getChars = function () {
    var out = '1';
    for (var i = 0; i < 3; i++) {
        out += numToChar(data[idx+i]);
    }
    pixIdx++;
    idx = pixIdx<<2;
    return out;
}

var getActionsNum = function () {
    var out = (data[idx] << 16) + (data[idx + 1] << 8) +
    data[idx + 2];
    pixIdx++;
    idx = pixIdx<<2;
    return out;
}
Appendix B

Sprite Sheets for File Size Testing

Figure 9.1: Thumbnails of sprite sheet samples.
## File Size Impact of Applying MESS

<table>
<thead>
<tr>
<th>File name</th>
<th>Size before</th>
<th>Size after</th>
<th>Growth ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ryuhayabusa.png</td>
<td>7366 bytes</td>
<td>9841 bytes</td>
<td>133.60%</td>
</tr>
<tr>
<td>general.png</td>
<td>8673 bytes</td>
<td>10871 bytes</td>
<td>125.34%</td>
</tr>
<tr>
<td>maincharacter.png</td>
<td>10464 bytes</td>
<td>13465 bytes</td>
<td>128.67%</td>
</tr>
<tr>
<td>billylee.png</td>
<td>11992 bytes</td>
<td>26229 bytes</td>
<td>218.72%</td>
</tr>
<tr>
<td>ryoneutral.png</td>
<td>12902 bytes</td>
<td>31798 bytes</td>
<td>246.45%</td>
</tr>
<tr>
<td>albatross.png</td>
<td>14786 bytes</td>
<td>33218 bytes</td>
<td>224.65%</td>
</tr>
<tr>
<td>ninjacrusaderssheet.png</td>
<td>18503 bytes</td>
<td>24067 bytes</td>
<td>130.07%</td>
</tr>
<tr>
<td>solbraver.png</td>
<td>18509 bytes</td>
<td>32766 bytes</td>
<td>177.02%</td>
</tr>
<tr>
<td>alen.png</td>
<td>25132 bytes</td>
<td>35975 bytes</td>
<td>143.14%</td>
</tr>
<tr>
<td>sadler.png</td>
<td>43745 bytes</td>
<td>54728 bytes</td>
<td>125.10%</td>
</tr>
<tr>
<td>zeroknuck.png</td>
<td>60575 bytes</td>
<td>85364 bytes</td>
<td>140.92%</td>
</tr>
<tr>
<td>capncommado.png</td>
<td>64099 bytes</td>
<td>156452 bytes</td>
<td>244.07%</td>
</tr>
<tr>
<td>cyclops.png</td>
<td>93657 bytes</td>
<td>135994 bytes</td>
<td>145.20%</td>
</tr>
<tr>
<td>fuuma.png</td>
<td>103238 bytes</td>
<td>128949 bytes</td>
<td>124.90%</td>
</tr>
<tr>
<td>alucard.png</td>
<td>145311 bytes</td>
<td>192165 bytes</td>
<td>132.24%</td>
</tr>
<tr>
<td>lancelot.png</td>
<td>348367 bytes</td>
<td>357471 bytes</td>
<td>102.61%</td>
</tr>
</tbody>
</table>

Table 9.1: File size growth of applying MESS.
List of Figures

Vector graphics vs raster graphics ........................................... 8
Sprite in game ........................................................................ 8
Source rectangle and destination rectangle ................................. 10
Sprite animation ..................................................................... 11
Misplacing the source rectangle ............................................... 11
Programmatically usable sprite sheet ....................................... 12
Sprite sheet with irregular pattern .......................................... 13
Metadata rectangles ................................................................ 13
Metadata rectangles grouped by semantics ............................... 14
Start point and end point of a rectangle .................................. 14

Animator overview .................................................................... 20
Sprite object scanner ................................................................ 22
Onion skinning ....................................................................... 22

Web application overview ....................................................... 24
Browser Market trend ............................................................. 25

Prototype Overview ............................................................... 31
Animator User Interface .......................................................... 33
Scan-line and interactive bounding boxes .................................. 34
Timeline controller of Adobe Flash .......................................... 34
Scanning detail ....................................................................... 36
Gallery user interface ............................................................. 36
Prepended metadata pixels ..................................................... 38
Meta-pixel Coding Scheme ..................................................... 39

Before and after applying MESS ............................................. 44
The growth ratio of file size .................................................... 44
Before and after applying MESS ............................................. 44

Sprite sheet samples ............................................................. 59