Proposing a practical approach to extract and read XML meta-data from sprite sheets using Blob detection algorithm

Marcelo de Barros Barbosa, Cecília de Barros Barbosa, André Freitas Barbosa
Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN)
Natal, RN, Brazil
marcelo.barbosa@ifrn.edu.br, ccilia.barbosa@gmail.com, andre.freitas@ifrn.edu.br

***Abstract*—This paper introduces a tool to semi-automatically generate meta-data from game sprite sheets. *MuSSE* is a tool developed to extract XML data from sprite sheet images with *non-uniform* – multi-sized – sprites. *MuSSE* (Multi-sized Sprite Sheet meta-data Exporter) is based on a Blob detection algorithm that incorporates a connected-component labeling system. Hence, blobs of arbitrary size can be extracted by adjusting component connectivity parameters. This image detection algorithm defines boundary blobs for each individual sprite in a sprite sheet. Every specific blob defines a sprite characteristic within the sheet: position, name and size, which allows for subsequent data specification for each blob/image. Those blobs are carefully optimized through an imbued threshold selection. This work also presents a parser to organize these meta-data into a readable solution for game engines. The parser is built to read XML meta-data generated through *MuSSE* and allow developers to set up game objects that can be used by an engine. Several examples on real images illustrate the performance of the proposed algorithm and working tool.**

***Keywords*—blob detection, sprite sheet, development tool**

I. INTRODUCTION

Digital Games uses sprites for visual representation of characters, objects and entities composing a scene. Sprites are two-dimensional images or animations embedded inside the game and loaded according to the graphical hardware constraints [1]. Sprites are usually represented inside a game by their bounding box rectangle: a shape definition for the corresponding image represented by their position (offset_x, offset_y) and size (width, height). A sprite's bounding box represent the entity that matches that image and is used to manage collision, screen display and other object manipulations. Those images need to be load on screen, but binding the texture is a time-consuming processing in a standard computer. It is ideal to store many smaller images on a larger one and bind it to memory only once, then draw portions of it as many times as needed [2].

A sprite sheet is a collection of sprites (individual game image textures) that can be loaded only once and rendered as many times as necessary. Using sprite sheets allows for better memory allocation and smoother texture rendering [3]. However, in order to load a sprite from a sprite sheet, the game needs to know the respective sprite position and size inside the sheet. Thus, the developer needs to know every sprite information from the sprite sheet to obtain the individual images.

Sprite sheets in which all sprites have identical sizes can be loaded by indexing the sprites according to their position in the sheet: starting at 0 for the top-left image and increasing their width to get adjacent columns and their height for subsequent lines [1]. In Fig. 1, we see an example of a sprite sheet where sprites have the same size.

Fig. 1. Sprites with identical size. Source: Final Fantasy© game.

Nevertheless, developers and artists will usually need to work with sprites with different sizes inside the same sprite sheet. In Fig. 2, we see an example of a sprite sheet with non-uniform sprite sizes:

Fig. 2. Sprites with different sizes. Source: Super Mario Bros 3© game.

Since the programmer does not know every specific sprite position and size, a game engine might need to specify image meta-data for every single sprite in that sheet. That meta-data file can be stored along with the sprite sheet images, among the project *assets* (audio and graphic files for a game), and used at run-time to access each corresponding sprite. This information can be manually composed by a programmer or an artist. Although, that process is not optimal and consumes a high amount of time, since meta-data for each sprite in the sheet must be individually write down on file. This process can also

be done during build-time by using feature detection algorithms to automatically generate the meta-data from sprite sheets but are susceptive to machine errors: the programmer might get different sizes for the same sprite in distinct builds or non-optimal size for the sprite bounding box.

This paper suggests an approach that allows users to semi-automatically extract individual sprite information through a more ambitious feature detection scheme. The tool developed in this work: MuSSE, uses a Blob detection algorithm to identify bounding boxes from sprites inside a sprite sheet and generate meta-data for that information.

As it stands, Blob detection implements a connectivity technique which provides a good threshold selection that can solve some of the machine problems cited. One of the most important advantages is the ability of the algorithm to isolate significant regions in a picture, producing bounding boxes with correct sizes and identifying isolated parts of a sprite as an included segment. Related work is discussed in Section II. The proposed algorithm is described in Section III: Blob Detection Algorithm. The developed tool: MuSSE (Multi-sized Sprite Sheet meta-data Exporter), is presented in details at Section IV.

This paper is an extension of the previous work published on a conference paper in 2015 [4]. The current version, however, come up with considerable improvements. A new view on the differences between MuSSE and other related tools is shown in Session II, bringing forward the contributions more clearly. An improved version of the Table of Label equivalence algorithm and the meanings and importance on defining a correct threshold are presented at Section III, along with multiple examples on threshold selection. Furthermore, a parser developed to read the data from the XML files generated through MuSSE is also introduced in Section IV, alongside a more understanding explanation on the XML format proposed.

II. Related Work

Feature detection refers to methods that aim to solve machine vision problems [5; 6]. According to [5], the central problem of computer vision is to understand an object or scene – from one or a sequence of images of a moving or stationary object – and their corresponding properties. Feature detection works on how to find interest points (features) in an image, computing abstractions of image information and making decisions regarding on how to represent those interest points and how to compare them with other interesting points in the image.

There is currently a series of techniques used for feature detection. These methods are usually applied to specific machine vision problems, like Edge detection and Corner detection [7].

Detecting corners is an essential operation in many computer vision and image processing applications such as motion tracking, shape representation, image registration, camera calibration, object recognition and stereo matching. Corners are important features in two-dimensional (2D) images as they can represent the shape of an object very well [8]. A corner can be defined as a location on an edge where the angle of the slope changes abruptly *i.e.* where the absolute curvature is high.

Edges provide important information towards human image understanding. It is the most important processing step in human picture recognition system. Edge detection procedure is used to find the discontinuities in depth, discontinuities in surface orientation, changes in material properties and variations in scene illumination [9].

Texture images treated by Edge detection and Corner detection algorithms contain their own particularities. At these computer vision problems, usually a given image has a limited extent or window (the "outer scale") as well as a limited resolution (the "inner scale"). These limits are set by the format of the image, *e.g.* the size of the photographic plate and the graininess of the emulsion [10]. Relevant details of images exist only over a restricted range of scale. Hence, it is important to study the dependence of image structure on the level of resolution [10; 11]. It is worth noticing that sprite sheets have fixed-scale: our scale space is pixel-related as opposed to being related to a scale of observation. Furthermore, there is no need for smoothing or image segmentation [12; 13]. Another feature detection technique more suitable for our solution is Blob detection.

Blob detection methods aims at identifying regions in an image that clearly differ in property from their surroundings. A blob is a region of an image that contains approximately close characteristics, such as brightness or color. In image processing, Blob detection main function is to identify same gray level pixels in the image. These pixels are separated into different blobs based on relationships of inter-connection [14]. In other words, a blob is defined as a region of connected pixels. Blob detection is used to identify these regions in images [15]. The algorithm discerns pixels by their values and place them into one of two categories: the foreground (typically pixels with a non-zero value) or the background (pixels with a zero value).

Defining a sprite meta-data from a game sprite sheet is a typical feature detection problem that can be solved by selecting a region of interest or interest points, which is a feature extraction solution approached by Blob detection algorithms.

Even though, to accommodate the huge variety of applications, Hinz [16] argues that a Blob detection algorithm must fulfill a number of general requirements, most notably:

- *Reliability / noise insensitivity*: Clearly, a low-level vision algorithm should be in some way robust against under- and over-segmentation due to noise.
- *Accuracy*: Many applications — especially in vision metrology — need highly accurate results in sub-pixel resolution.
- *Scalability*: The algorithm should be scalable so that primitives of different size can be extracted.
- *Speed*: The algorithm should be applicable also to (near-)real-time processing.

Since our application deals with a specific subset of images, more precisely game sprite sheet images, we can define these limitations as is:

- *Reliability / noise insensitivity*: Sprite sheets have well defined backgrounds. There is no need for noise treatment between image foreground and image background. In our solution, a RGB filter is applied to set a relevant color key for the sprite sheet background. It also enables to dynamically define background settings for each specific sheet.
- *Accuracy*: Our solution uses a connected-component labeling that allows for per-pixel accuracy blob detection.
- *Scalability*: Proposed solution applies a connected-component algorithm with a threshold parameter that allows for specific set up on blob sizes accuracy based on sprites contained on the sheet. Sprites with smaller sub-images will need a bigger threshold. Furthermore, sprites with no sub-images or close-edges can be extracted with smaller thresholds. In other words, the higher the threshold, more precise a blob is to detect separate parts in a single sprite.
- *Speed*: Meantime, no limit definitions for speed are necessary in our solution since it is a pre-processing/build-time activity and not a run-time procedure.

There is also a series of licensed and free tools already built in game frameworks or engines for similar functionality. Although, most of these tools are meant for the opposite purpose: to convert individual sprites into mapped — meta-data referenced — sprite sheets, alias: Texture Atlas. Some of these tools are:

Texture Packer [17]: a licensed software for sprite sheet generation. Packages a series of separate sprites and export into a list of specific formats. Exported sprite sheets can be read by the engines: AndEngine (android), Cocos2DX (mobile), V-Play (cross-platform), Unity and others.

Unity Sprite Packer [18]: this is a built-in tool into Unity game development framework. It is used to pack graphics from several sprite textures tightly together within a single texture sheet, known as an Atlas. This process is applied to increase in-game image reading performance. Sprite Packer is a utility tool to automate the process of generating these Atlases. However, there is no or little documentation on how this is done, at least on how the algorithm is implemented.

LibGDX Texture Packer [19]: LibGDX engine has a Texture Packer tool which is a command line application that also packs many smaller images into larger images. These Atlas files containing sprites meta-data are saved in a “minimal” (simplified) JSON format. There is a parser built-in the LibGDX engine for reading those files.

Sprite Cutter [20]: this is a freeware tool that allows the user to manually cut/crop and export individual sprites into a new sprite sheet (Atlas). It has a built-in auto-cut option, but also no implementation details on internal algorithm functionality.

Construct 2 [21]: Construct 2 game framework has a built-in image editor that allows for automatic bounding box shaping for image sprites with fixed size. It uses Edge detection algorithm (no implementation details acquired) to set a sprite bounding box polygon or an image full size to define their bounding box rectangle. Although it is a helpful tool, it has no size optimization. Also, there is an Animation editor that allows to load full sprite sheets and convert it into individual sprites but – again – works only with fixed size sprite sheets. The user needs to manually adjust sprites positions for multi-sized sprites.

Most of these tools, such as Texture Packer, Unity Sprite Packer and LibGDX Texture Packer, pack and create new sprite sheets. Creating a texture atlas will save memory space in the game build. LibGDX Texture Packer uses multiple packing algorithms, in which case, the most important is based on maximal rectangle algorithm. They all have an internal algorithm for cropping and trimming sprites into optimal size, but don't provide implementation details, except Construct 2 that states using an Edge detection algorithm for bounding box polygons.

Unity Sprite Packer, Sprite Cutter and Construct 2 generates no meta-data for those sprite sheets. The data generated in Unity is used internally only. Texture Packer and LibGDX Texture Packer generates meta-data, but while the former generates the meta-data in no standard format (export to a lot of engines with different formats), the second generates meta-data in JSON that can be read only by LibGDX engine.

In short, those tools provide little to no algorithm information and they mainly engage on building sprite sheets through packing. Our goal is just the opposite: to acquire data from sprites. When they do provide sprite’s data, the meta-data generated is engine sensitive, binding their use to specific frameworks and platforms. Meanwhile, MuSSE strictly generates meta-data through Blob detection algorithm, but the advantage is that it produces data in a standard pattern that can be used by any engine.

III. Blob Detection Algorithm

Our Blob detection approach consists of two major tasks. The first one comprises the extraction of potential blobs in sub-pixel precision using a connected-component algorithm (Section A). The second task outlined in Section B consists of reconstructing the boundary around a given point based on a table of label equivalence.

Both tasks can be presented as a sequential image detection algorithm. This algorithm is defined by the following steps:

(1) Check if a pixel is foreground or background (transparency).

(2) Apply a connected-component labeling. It will define the current pixel's label according to nearby labels (neighborhood) or assign a new label.

(3) Reduce labels based on a table of label's equivalence.

(4) Define minimum boundaries around target image based on identified blob.

The proposed algorithm uses a connected-component labeling with an imbued threshold for applying minimum pixel distance for subsequent label equivalence. A threshold is used to detect candidate blobs and label it accordingly [22; 23]. Usually, we will have sprites with full connectivity within their parts – generating easy-to-represent blobs – as seen in Fig. 3:

Fig. 3. Sprites where all pixels are connected. Source: Super Mario World© game.

In Fig. 3, there are no "loose" points: every pixel in the image is connected. Thus, checking for label equivalence is trivial. Although, that is not always what we see in individual sprite images. Some sprite sheets may contain sprite images with separated small parts (Fig. 4). That is why we need an imbued threshold: to adjust our algorithm to these situations and correctly identify which parts relate to the same blob.

Fig. 4. Sprites in which some small parts are detached. Source: Undertale© game.

Another example can be seen in Fig. 5. Character sprites may have visual effects that somehow detach from the main part of the sprite.

Fig. 5. Sprites in which some small parts are detached. Source: Nuclear Throne© game.

Having a minimum pixel threshold for equivalence is important to deal with sprite images with small, isolated parts, or sprites with elements that have no connectivity to the "main body". Another example (Fig. 6):

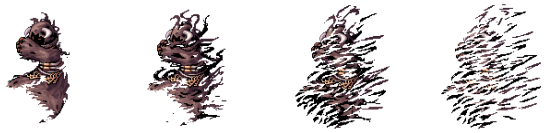

Fig. 6. Sprites where most pixels are detached from the others. Source: Ragnarok Online© game.

In the sections below, we fully describe our algorithm. Given that the algorithm is implemented in distinct parts, we will present each separately for better understanding. In Section A: Connected-component Labeling, we describe steps 1 and 2 of the algorithm, while on Section B: Table of Label Equivalence, we describe steps 3 and 4.

A. Connected-component Labeling

The first step of our Blob detection consists on a pixel-based image detection algorithm that uses sub-pixel precision to extract potential blobs. This step has two main tasks:

I. Check if a pixel is foreground or background (transparency).

II. Apply a connected-component labeling. It will define the current pixel's label according to nearby labels (neighborhood) or assign a new label.

The first step is responsible to find parts of the picture — sprite sheet — that differ significantly from the background (likely sprites). The second is used to find common connected parts among these images, based on a labeling system.

Connected-component labeling is used in computer vision to detect connected regions in binary digital images, although color images and data with higher dimensionality can also be processed [24]. A common set of labels is defined among an image. Blob extraction is generally performed on the resulting subtracted label set.

Connected-component labeling is a fundamental task common to virtually all image processing applications in two and three dimensions [24]. For a binary image, represented as an array of d-dimensional pixels or image elements (Fig. 7), connected-component labeling is the process of assigning labels to the BLACK image elements in such a way that adjacent BLACK image elements are assigned the same label [25; 26].

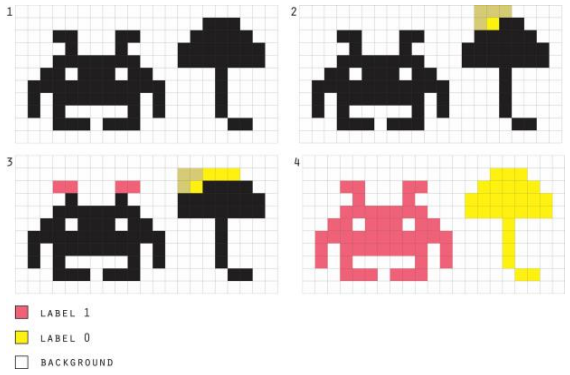

Fig. 7. Labeling example.

In our proposed solution, instead of specific BLACK and WHITE images, we use a verification algorithm to define if a pixel is foreground (BLACK) or background (WHITE). A pixel is classified as background if it is transparent or matches the color key defined for the sprite sheet background. Otherwise, the pixel is classified as foreground, e.g. it belongs to one sprite.

The connected-component part of our algorithm works by building a succession of small boundaries around a given point using a specific threshold. Further, it is possible to label that point using a very simple method based on the labels obtained from the region around that position.

Threshold selection involves choosing a search area adjacent to the given point, based on a fixed number of pixels around it. Here, "adjacent" may mean 4-adjacent or 8-adjacent [24]. In our algorithm, we use an 8-adjacent search path, likely: *8-neighborhood connectivity* (Fig. 8), to ensure more precise accuracy.

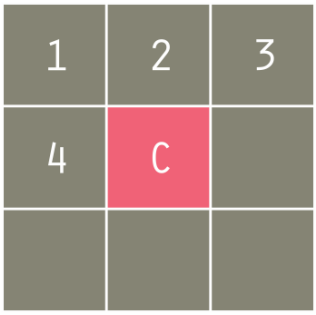

Fig. 8. *8-Neighborhood connectivity* search path.

The overall goal of connected-component is to label each pixel within a blob with the same identifier. These identifiers are represented by label numbers.

The first stage is to circle through all the pixels inside the selected area and verify the corresponding label numbers from neighbor pixels. In the 8-neighborhood connectivity, we only check the upper-left labels because those are the ones guaranteed to be labeled in our search path. That happens because we circle the image using a raster scan: from top to bottom, left to right. Fig. 9 shows an example on how it is applied:

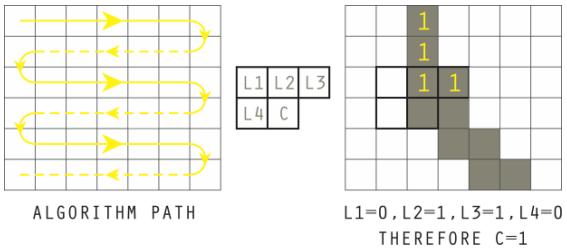

Fig. 9. Labeling example using *8-neighborhood connectivity*.

All the labels must be stored in a matrix of equal dimension as the original image. This way, we can set one label entry per pixel in the image. This matrix starts completely unlabeled and, as the algorithm iterate through the image, we fill the matrix using the 8-neighborhood connectivity method.

The algorithm is applied per given pixel in the image thus identifying the labels of neighborhood pixels L1, L2, L3 and L4, and applying to the position C.

The connected-component labeling algorithm is described below:

```
for each pixel in selected area
   if current pixel is foreground
      check nearest neighbors for a valid
      label
      if there's no valid label on neigh-
      borhood
         get a new label
      else
         get neighbor's label
      end if
      save pixel label on the list
   end if
done
```

Fig. 10 shows an input monochrome image and the result of the labeling process. We can see that the labeling process sometimes gives multiple labels to the same blob. One critical question is that the blob which is easily identifiable by human eye as a single blob is often interpreted by the algorithm as several distinct [14].

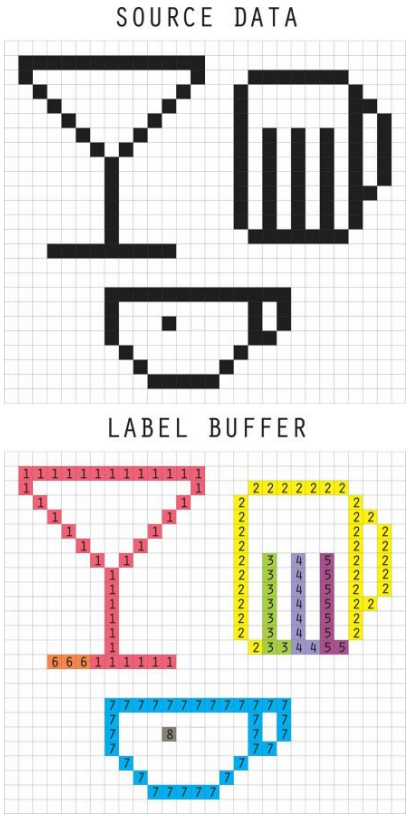

Fig. 10. Labeling example (with more than one label per blob).

To understand why this happens, take a look at the blob comprising labels 2, 3, 4 and 5. When the algorithm reaches the first pixel labeled 3, it has no way of knowing at this stage that it is connected to those labeled 2. Same applies to pixels labeled 4 and 5. However, this can be solved.

The solution is to create a table to keep note of which labels refer to the same blob when the two labeled sections eventually connect. During this step, the surrounding labels for each labeled pixel will also be added to a list. The algorithm fills a table of equivalence between associated labels, checking all neighbors to mark equivalence. If we have multiple neighbors with different labels, we assign for that pixel the first label found and indicate that all the other ones are equivalent. In short, the filled table contains every label in the image and the labels from their surrounding neighbors too. That makes an equivalence.

B. *Table of Label Equivalence*

The second step of our Blob detection consists on reconstructing the boundary around a given point based on the connected-component labeling algorithm. This step also has two main tasks:

I. Reduce labels based on a table of label's equivalence.

II. Define minimum boundaries around target image based on identified blob.

Our label connectivity is made based on a two-pass algorithm. The first pass uses the table of equivalence generated by our connected-component algorithm presented in Section A. In this step, the algorithm first reduces the labels to their corresponding representatives. A representative is given by the minimum value in the table of equivalence, *i.e.* minimum label *id* compared to surrounding neighbors. That process is described below:

```
for each pixel in selected area
   if pixel is labeled
      relabel pixel with the lowest equivalence
      label
   end if
done
```

In the second pass, the algorithm creates a list for every blob in the selected area and matches it accordingly to the reduced labels. This step of the connected-component labeling algorithm is described below:

```
for every label in list
   if this is first time checking this
   label id
      create a new blob for that label
   else
      add label to existing blob (repre-
      sentative) based on their table of
      equivalence
   end if
done
```

That will guaranty that all equivalent labels are assigned the same region value. We can apply this to the examples shown in Section A. The example in Fig. 10 has the following table of label equivalence:

TABLE I. TABLE OF LABEL EQUIVALENCE FOR EXAMPLE FIG. 10.

Label *id*	**Equivalent Labels**
1	1,6
2	2,3,4,5
3	2,3,4,5
4	2,3,4,5
5	2,3,4,5
6	1,6

As a result, the algorithm will reduce labels 1 and 6 to label 1, and labels 2, 3, 4 and 5, to label 2. Notice that labels 7 and 8 were excluded to simplify this illustration. See Fig. 11 for an example:

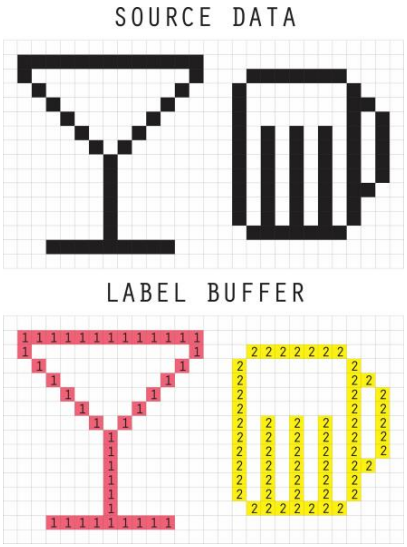

Fig. 11. Labeling example (with correct label equivalence).

Unfortunately, that won't work for all images. According to our algorithm, the images referenced by labels 7 and 8 (Fig. 12) would both be reduced to label 7, however, the threshold value must be adjusted first.

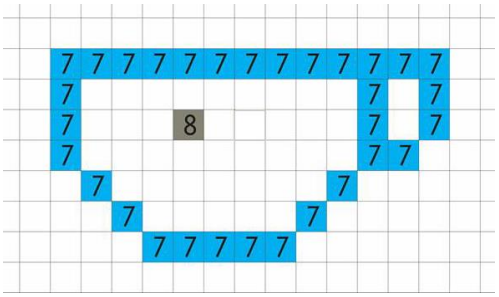

Fig. 12. Labeling example (with wrong threshold).

Selecting a small threshold will split the image into two different blobs: one for the whole image corresponding to label 7 and one for the small image corresponding to label 8. This will result in the following table of label equivalence:

TABLE II. TABLE OF LABEL EQUIVALENCE FOR EXAMPLE FIG. 12.

Label *id*	Equivalent Labels
7	7
8	8

This happens mainly because we will be looking only at a small area around each pixel. Thus, we need to increase the threshold to look further away from each pixel, identifying separated parts: elements that are detached from the "main body" of the image. Applying the process to the example in Fig. 12 will result in the corresponding table of label equivalence:

TABLE III. TABLE OF LABEL EQUIVALENCE FOR EXAMPLE FIG. 12.

Label *id*	Equivalent Labels
7	7, 8
8	8, 8

As a consequence, the algorithm will correctly identify labels 7 and 8 as belonging to the same sprite. This generates the blob displayed in Fig. 13. A more practical example on threshold values will be shown in Section IV-A.

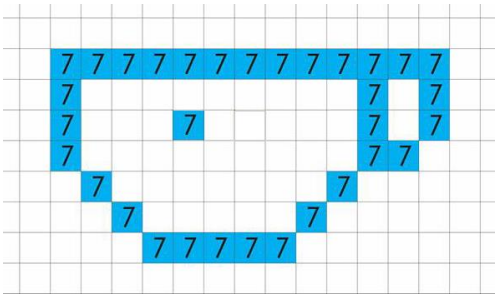

Fig. 13. Labeling example (with correct threshold).

IV. MUSSE

MuSSE is a tool developed to extract meta-data from sprite sheets with non-uniform sprites. *MuSSE* (Multi-sized Sprite Sheet meta-data Exporter) is an open-source Groovy program with a simple Swing interface, available online [27] and distributed under the terms and conditions of the zlib/png license [28]. It can load sprite sheets from multiple image formats, including: *jpg/jpeg, png, bmp, gif* and *wbmp*.

The software implements the Blob detection algorithm described in Section III and a document generator that exports the meta-data obtained from the sprite sheet into an XML file. The data extracted through MuSSE contains: position, name and size from the individual sprites contained in the sprite sheet.

XML (Extensible Markup Language) is a widely used international text processing standard [29] and is used to create human-legible, clear and straightforward data documentation. According to [30], an XML document doesn't do anything by itself: it must be combined with an application program that does something useful with it.

In our case, MuSSE will save the extracted meta-data into an XML file which can be read through a parser and used by the developer preferred engine. A parser is the interface between an XML document and the application program that uses it. The parser reads XML documents and provides application programs with access to the documents' internal structure and content [30].

A. MuSSE Interface

The user has two options to apply the Blob detection algorithm on the sprite sheets:

(1) manual selection of a target area from the sheet; or

(2) "Cut entire sheet" option from the "Actions" menu.

Option 1 can be activated by clicking and dragging the mouse over the image. This way, the Blob detection algorithm is only applied to that selection of the sprite sheet (Fig. 14).

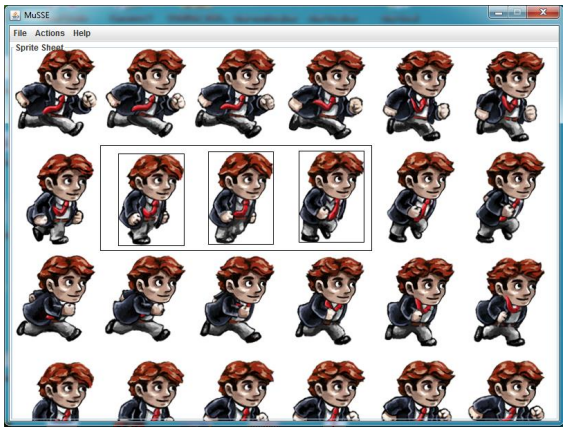

Fig. 14. Select target area using MuSSE. Source: MuSSE with sprite sheet image from the game Braid©.

If the user chooses Option 2, the algorithm is applied to the whole sprite sheet image. Fig. 15 illustrate the process.

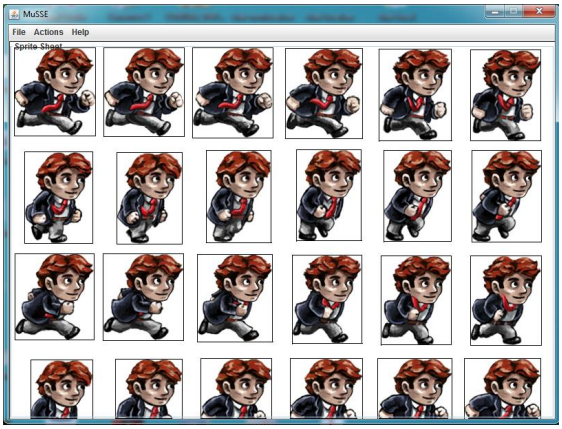

Fig. 15. Select entire sprite sheet using MuSSE. Source: MuSSE with sprite sheet image from the game Braid©.

Every time the user cuts a selection, MuSSE saves it as a new animation inside the XML file. Every image inside the selection is saved as a new sprite for that animation, with their corresponding name, position and size. An animation is the quick succession of game images (sprites) to give the illusion of movement. Thus, those various animations are used inside a game to give entities the idea of movement: running characters, attacks, special effects, moving scenarios, and others. Fig. 16 shows an example XML with two animations.

```
<?xml version="1.0"?>
<spritesheet>
    <animation name="walk">
        <sprite name="Label2">
            <offset_x>157</offset_x>
            <offset_y>162</offset_y>
            <width>95</width>
            <height>133</height>
        </sprite>
        <sprite name="Label4">
            <offset_x>673</offset_x>
            <offset_y>159</offset_y>
            <width>101</width>
            <height>133</height>
        </sprite>
    </animation>
    <animation name="run">
        <sprite name="Label3">
            <offset_x>274</offset_x>
            <offset_y>310</offset_y>
            <width>109</width>
            <height>127</height>
        </sprite>
        <sprite name="Label1">
            <offset_x>412</offset_x>
            <offset_y>311</offset_y>
            <width>101</width>
            <height>128</height>
        </sprite>
    </animation>
</spritesheet>
```

Fig. 16. Example XML created using MuSSE with multiple animations.

Example applications of MuSSE are shown in Fig. 17 and 18, where the Blob detection algorithm is applied to entire sprite sheet and to a subset of sprites:

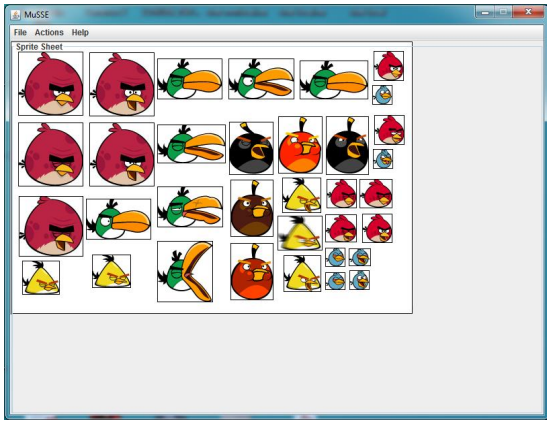

Fig. 17. Sprite sheet with high connectivity. Source: MuSSE with sprite sheet image from the game Angry Birds©.

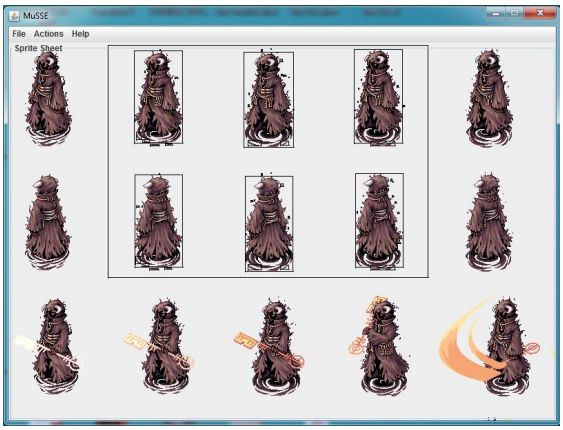

Fig. 18. Sprite sheet with low connectivity. Source: MuSSE with sprite sheet image from the game Ragnarok Online©.

Fig. 19 shows a close-up from Fig. 18. The sprite sheet used for this example has sprites with low connectivity. In other words, sprites with detached parts from the "main body". As described in Section III, that particularity causes the smaller – disconnected parts – to be mapped as new blobs:

Fig. 19. Sprite sheet with low connectivity (close-up). Source: MuSSE with sprite sheet image from the game Ragnarok Online©.

As pointed out before, smaller thresholds for our connected-component method will result in a higher number of blobs in the selected image, as seen in Fig. 19. That problem can be solved by selecting a more suitable threshold for the sheet.

The user can choose a specific threshold for the Blob detection algorithm in MuSSE by selecting the option "Sensibility (threshold)" from the "Actions" menu (Fig. 20).

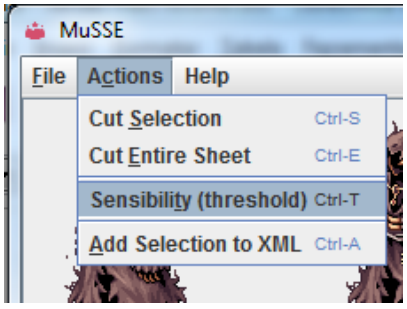

Fig. 20. MuSSE's menu.

Threshold value starts at 1 (one) pixel and must be manually set according to the distance between the small parts and the sprite body. MuSSE will correct the bounding boxes for sprites with isolated parts if the threshold is adjusted to a higher value. For this example, a threshold of 8 (eight) pixels where necessary. As a result, the following sheet (Fig. 21) can be attained:

Fig. 21. Example from Figure 19 using correct threshold. Source: MuSSE with sprite sheet image from the game Ragnarok Online©.

MuSSE's XML is ready to be used into developers' engine but – as practical as it is – some compatibility adjustments are necessary in the suggested XML structure. These adjustments relate mostly to missing meta-data that might be required by some engines.

Most engines graphic component will need the color-key value of the sprite sheet background to adjust correct transparency for the image files. Those values can be extracted in the first part of our algorithm (described in Section III-A) and saved to the generated XML file.

Some other meta-data such as the sprite sheet image path might also be added into the XML file in order to concentrate information in the document. Facilitating the developer's work of having to pass it inside the code.

Regarding the Animation, two improvements might be made:

(1) Add the duration of each individual sprite in the animation to the sprite meta-data in the XML.

Every frame in the animation may have different duration timers. This will make some sprites stay longer on the screen or display them faster in the animation.

(2) Add an anchor point to the sprite images.

Given that sprites may have different sizes, inappropriate overlapping of the images during the animation may occur. Particularly, sprites with different sizes may be rendered by the engine without being centered to one other. Adding an anchor point to these images will identify in which position they need to be drawn during the animation.

B. Using a Parser for MuSSE's XML

MuSSEXMLParser is an open source XML parser available online [31] in C++ and Java to read XML meta-data generated through MuSSE. A Parser is a program that receives input in the form of structured files or scripts and breaks them up into parts that can be treated in a particular context. In other words, a parser is a program to help programmers extract data from XML and other document files [32] and may also check the validity of those files.

In our case, MuSSEXMLParser was developed to break MuSSE XML into data files that could be used by an engine.

MuSSEXMLParser works like this:

```
open sprite sheet from xml file
for every animation in list
   register animation name
   for every sprite in the animation
      register sprite name
      register sprite size and position
   done
done
```

The parser organizes data in a way that developers can easily set up the corresponding game objects and later use these objects as game sprites and animations.

While registering animation data, MuSSEXMLParser extracts an object structure that can be organized by the developer in their preferred programming language. In C++, developers would need to do something like the structure shown in Fig. 22. In which SPRITE_QUANTITY is the number of sprite entities inside the animation and is automatically provided by the parser.

```
struct Animation {
  string name;
  Sprite[SPRITE_QUANTITY] sprite_list;
}
```

Fig. 22. Example object structure for animations.

MuSSEXMLParser also extract object structures for the individual sprites inside the animation. In C++, that object definition is shown in Fig. 23.

```
struct Sprite {
  string name_or_id;
  int offset_x, offset_y;
  int width, height;
}
```

Fig. 23. Example object structure for sprites.

Thus, developers could organize animation into objects and arrange lists of sprites for each respective animation. Implementation of these objects would be adaptable to the users' engine.

Association with an engine could be made by creating a sprite sheet interface connecting animations to the parsed files. Developer could make a dependency from the animation to the

parser trough this interface. An example is shown in the Class Diagram in Fig. 24.

Fig. 24. Class Diagram for a possible implementation.

In this example, the MuSSEXMLParser is the component that realizes the SpriteSheet interface. The Animation class is connected to the SpriteSheet interface with a dependency relationship line and uses this interface to access the parser services.

V. Conclusion

MuSSE is a development tool intended to create XML files containing meta-data for a set of sprites on a sprite sheet. The software applies a Blob detection algorithm to the image – be it a subset or an entire sprite sheet – and extracts the information for the contained sprites. Both tool and algorithm are open source and available online [27].

The Blob detection algorithm is used to extract important sprite characteristics: position in the sprite sheet, name reference for the image and matching sizes (width and height). That meta-data is used to build the corresponding bounding box rectangles for the sprite images. Those shapes are really important in-game as they are meant to be used for collision detection between entities and to manipulate objects in a scene.

MuSSE's Blob detection algorithm is based on a connected-component labeling method and a table of label equivalence. The connected-component labeling identifies every pixel inside an image with a label, where matching labels corresponds to a unique blob. The proposed algorithm also uses a table of label equivalence to identify different labels that belongs to same blob, reducing errors and optimizing the results. Blob detection implements a connectivity technique which provides a good threshold selection that can solve some machine vision problems, presenting important advantages over other implementations.

This paper presents an academic view on using feature detection techniques to develop a tool to generate sprite sheet information. Studied tools with similar functionality presents poor documentation regarding the techniques applied. Also, most tools are used for the opposite purpose: to convert individual sprites into sprite sheets. New researches should further investigate MuSSE algorithm efficiency against Construct 2 and Unity Sprite Packer algorithms.

A parser to read and manage sprite sheet meta-data is also presented. MuSSEXMLParser open up XML files created using MuSSE and break up the information into readable – easy to manipulate – data, ready to convert into sprite and animation objects.

As future work, we aim at improving MuSSE's interface by adding a color pallet for the user to dynamically select the background color from the sprite sheet. And also improve the tool by complementing functionality in two paths:

First one, implementing a threshold detection method to automatically search for a minimum (optimal) pixel range threshold. That is intended to avoid overlaying blobs in auto-detect mode and may be achieved by applying a function that takes on account the number of blobs intersections *vs* the threshold level. We can solve that by finding a minimum threshold level that has no blobs inside one another.

As a second option, MuSSE can also be improved by constructing a packing algorithm. Sprites can be packed – arranged into smaller sprite sheets – by applying a series of rotation and translation operations. This way, MuSSE can produce more compact and optimized sprite sheets.

The parser can also be improved. Some new meta-data might also be added to the MuSSE's XML files to improve integration with game engines, such as the color-key values for the background, the sprite sheet image file path and the sprites duration and anchor point in the animation.

Finally, we can also make the generated XML editable before being saved. Adding an interface to easily change the animation and the sprite names before saving the file may ease the programmer's work while loading the data.

References

[1] E. B. Damiani, Programação de Jogos Android: Crie seu próprio game engine!. Novatec Editora, 2014.

[2] D. Shneier, M. Woo, J. Neider and T. Davis, OpenGL (R) programming guide: The official guide to learning OpenGL (R), version 2.1. Addison-Wesley Professional, 2007.

[3] H. M. Chandler, Manual de produção de jogos digitais. Bookman, 2012.

[4] M. B. Barbosa, C. B. Barbosa and A. F. Barbosa, "MuSSE: a tool to extract meta-data from game sprite sheets using Blob detection algorithm," in Proc. XIV Simpósio Brasileiro de Jogos e Entretenimento Digital. Porto Alegre: SBC, 2015.

[5] C. Brown, Advances in computer vision. Psychology Press, 2014.

[6] D.Cristinacce, and T. F. Cootes, "Feature Detection and Tracking with Constrained Local Models," in BMVC, 2006, vol. 1, no. 2, p. 3.

[7] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama and P. Manneback, "A multi-resolution fpga-based architecture for real-time edge and corner detection," in Computers, IEEE Transactions, 2014, pp. 2376-2388.

[8] S. W. Teng, R. M. N. Sadat and G. Lu, "Effective and efficient contour-based corner detectors," in Pattern Recognition, 2015, vol. 48, no. 7, pp. 2185-2197.

[9] K. K. Jena, S. Mishra and S. N. Mishra, "Edge Detection of Satellite Images: A Comparative Study," in IJISET, 2015, vol. 2, no. 3, pp. 2015.

[10] J. J. Koenderink, "The structure of images," in Biological cybernetics, 1984, vol. 50, no. 5, pp. 363-370.

[11] K. Ikeuchi, Computer Vision: A Reference Guide. Springer Publishing Company, Incorporated, 2014, pp. 701-713.

[12] R. Collins, "Mean-shift blob tracking through scale space," in Proc. 2003 IEEE Computer Society Conference. Computer Vision and Pattern Recognition, 2003, vol. 2, pp. II-234.

[13] T. Lindeberg, "Feature detection with automatic scale selection," in International journal of computer vision, 1998, vol. 30, no. 2, pp. 79-116.

[14] F. Wang, X. Ren and Z. Liu, "A robust blob recognition and tracking method in vision-based multi-touch technique," in Parallel and Distributed Processing with Applications, 2008. ISPA'08. IEEE, International Symposium, pp. 971-974.

[15] S. Sookman, "INSPECTION-Image Analysis Software-Blob Analysis and Edge Detection in the Real World," in Evaluation Engineering, 2006, vol. 45, no. 8, pp. 46-49.

[16] S. Hinz, “Fast and subpixel precise blob detection and attribution,” in Image Processing, ICIP 2005. IEEE International Conference, vol. 3, pp. III-457, September 2005.

[17] TexturePacker. (2015, June 20). Texture Packer [Online]. Available: www.codeandweb.com/texturepacker.

[18] Unity. (2015, June 20). Unity3D Game Engine [Online]. Available: docs.unity3d.com/Manual/SpritePacker.html.

[19] LibGDX. (2015, June 18). LibGDX Game Engine [Online]. Available: github.com/libgdx/wiki/Texture-packer.

[20] SpriteCutter. (2015, June 21). Sprite Cutter [Online]. Available: spritecutter.sourceforge.net/manual.html.

[21] Construct. (2015, June 21). Construct 2 Game Engine [Online]. Available: www.scirra.com/manual/48/image-and-animations-editor.

[22] M. Shneier, “Using pyramids to define local thresholds for blob detection,” in Pattern Analysis and Machine Intelligence, IEEE Transactions, 1983, no. 3, pp. 345-349.

[23] A. Kaspers, Blob Detection. Biomedical Image Sciences. Image Sciences Institute, UMC Utrecht, 2011.

[24] M. Dillencourt, H. Samet and M. Tamminen, “A general approach to connected-component labeling for arbitrary image representations,” Journal of the ACM (JACM), 1992, vol. 39(2), 253-280.

[25] M. Pordel and T. Hellström, “Semi-Automatic Image Labelling Using Depth Information,” Computers, 2015, vol. 4, no. 2, pp. 142-154.

[26] T. Shreekanth and V. Udayashankara,. “An Application of Eight Connectivity based Two-pass Connected-Component Labelling Algorithm For Double Sided Braille Dot Recognition,” International Journal of Image Processing (IJIP), 2014, vol. 8, no. 5, pp. 294.

[27] MuSSE. (2016, February 22). MuSSE [Online]. Available: https://github.com/marcelomesmo/MuSSE.

[28] Zlib. (2016, February 22). Zlib/png License [Online]. Available: zlib.net/zlib_license.html.

[29] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau,. (1998). Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210 [Online]. Available: www.w3.org/TR/1998/REC-xml-19980210.

[30] J. Roy and A. Ramanujan, XML schema language: taking XML to the next level. IT professional, 2001, vol. 3, no. 2, pp. 37-40.

[31] MuSSEParser. (2016, February 22). MuSSE XML Parser [Online]. Available: https://github.com/marcelomesmo/MusseXmlParser.

[32] A. V. Aho, S. Ravi and D. U. Jeffrey. Compilers, Principles, Techniques. Addison wesley, 1986.