MuSSE: a tool to extract meta-data from game sprite sheets using Blob detection algorithm

Marcelo B. Barbosa Cecília B. Barbosa André F. Barbosa

Federal Institute of Education, Science and Technology of Rio Grande do Norte, Brazil

Abstract

This paper introduces a tool to 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. Several examples on real images illustrate the performance of the proposed algorithm and working tool.

Keywords: blob detection, sprite sheet, development tool

Authors' contact:
marcelo.barbosa@ifrn.edu.br
ccilia.barbosa@gmail.com
andre.freitas@ifrn.edu.br

1. Introduction

A sprite is a two-dimensional image or animation that is embedded in an audiovisual layer inside a game [Damiani 2014]. Sprites are important elements in a digital game, they visually represent a series of entities composing a scene: characters, projectiles, able-to-interact objects etc. Commonly, sprite characteristics must be adjusted to a graphical hardware constraints. Loading an image involves a series of data handling, like: image format, pixel depth, alpha channel and more [Damiani 2014].

In OpenGL, for instance, in order to display images on screen, we need to:

1: bound an image texture to memory;
2: drawn image texture on screen;
3: after drawing is done, free memory so another image can be bound;
4: repeat.

Furthermore, this process needs to be repeated for as much images as needed to be draw on screen [Shneier et al. 2007].

Binding the texture is relatively expensive, so it is ideal to store many smaller images on a larger one, bind the larger texture to memory once, then draw portions of it as many times as needed. Reusing same portions of a bigger image allow us to decrease memory usage while drawing a complete scene [Damiani 2014]. That is the whole concept behind sprite sheets: a collection of sprites (individual game image textures) that can be loaded to memory only once and rendered as many times as necessary.

A game sprite sheet is a set of game sprites embedded in a single bidimensional structure that allows for better memory allocation and smoother texture rendering [Chandler 2012]. These sprite sheets contains a subset of images used to draw a specific scene: being it a splash screen, a menu screen or a game phase, map or scenario. In order to load a sprite from a sprite sheet, the game needs to know the respective sprite position and size inside the sheet.

After the loading is done, sprites are usually represented inside a game by its bounding-box rectangle: a shape definition for the corresponding image. A sprite's bounding-box is used to represent the entity that matches that image. This process is applied to manage collision detection between in-game entities and handling events related to object manipulation: rotation, translation etc. Those bounding boxes are defined by it's:

- position (offset_x, offset_y), and
- size (width, height).

These data must be know by the programmer in order for the game to obtain the sprites from the sprite sheet. Unfortunately, this is not a trivial work, since the programmer needs to know every single sprite information from the sheet.

Sprite sheets on which all sprites have identical sizes are easy to be loaded by a game engine. In that case, one simply needs to specify the x- and y-offset to get a particular sprite in the sheet and the corresponding size for that sprite sheet images. That way, all images are indexed by its own dimensions: first sprite corresponding to 0 (zero), by increasing width to the right we get the next image and by increasing the image height we go to the next line when the sprites on that line ends [Damiani 2014]. In Figure 1, we see an example of a sprite sheet with same size sprites:

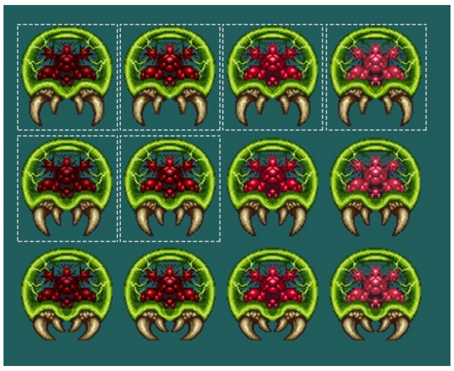


Figure 1: Sprites with identical size. Source: Super Metroid© game.

However, a game sprite sheet not always will have same (fixed) size sprites within its content. More commonly, developers and artists will have to deal with multi-sized sprites in a sheet. In other words, sprites with non-uniform (different) sizes inside the same sprite sheet. In Figure 2, we see an example of a sprite sheet with non-uniform sprite sizes:

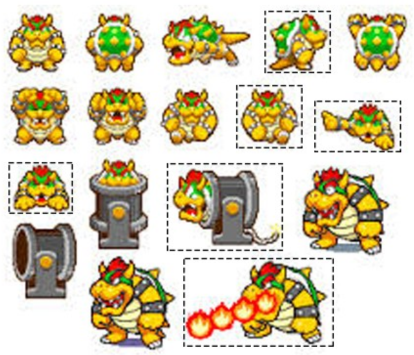


Figure 2: Sprites with different sizes. Source: Paper Mario© game.

Reading sprites with different sizes from a sprite sheet in execution time is a high cost activity, even for newer and faster *GPUs* (*graphics processing units*): 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 during image loading.

A common approach is to keep the bounding boxes information for every sprite in the sheet into a serialized file. That meta-data file can be stored among the project *assets* (audio and graphic files for a game), along with the sprite sheet image. At run-time, the engine loads both the sprite sheet and the meta-data file with the information needed 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. This process can also be done during build-time, by using feature detection algorithms to automatically generate the meta-data from sprite sheets. Specific tools can be developed to achieve that but, since it is an image recognition problem, it may occur in machine errors, such as: different sizes for the same sprite in distinct builds, non-optimal size for the sprite bounding box or missing small parts of a sprite if they are detached or too far away from the sprite's "main body".

This paper suggests an approach that allows users to automatically extract these 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 2. The proposed algorithm is described in Section 3: Blob Detection Algorithm. The developed tool: MuSSE (Multi-sized Sprite Sheet meta-data Exporter), is presented in details at Section 4.

2. Related Work

Feature detection refers to methods that aim to solve machine vision problems [Brown 2014; Cristinacce and Cootes 2006]. According to Brown [2014], 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 its corresponding properties. Feature detection targets on how to find interest points (features) in an image, computing abstractions of image information and making decisions regarding on how to represent that 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 [Possa et al. 2014].

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 [Teng et al. 2015]. 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 [Jena et al. 2015].

Texture images treated by Edge detection and Corner detection algorithms contains its 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. by the size of the photographic plate and the graininess of the emulsion [Koenderink 1984]. 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 [Koenderink 1984; Ikeuchi 2014]. It is worth noticing that sprite sheets has fixed-scale, since our scale space is pixel-related, in opposite to be related to a scale of observation. Furthermore, there is no need for smoothing or image segmentation [Collins 2003; Lindeberg 1998]. 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 its surrounding. A blob is a region of an image that contains approximately close characteristics, such as brightness or color. In processing, Blob detection function mainly considers to identify same gray level pixels from the image. And these pixels are separated into different blobs based on relationship of inter-connection [Wang et al. 2008]. In other words, a blob is defined as a region of connected pixels. Blob detection is used to identify these regions in images [Sookman 2006]. 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 solution, as is a feature extraction solution approached by Blob detection algorithms.

Even though, to accommodate the huge variety of applications, Hinz [2005] 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 define a relevant color key for the sprite sheet background. It also enables to dynamically set background definitions 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, higher the threshold, the more precise a blob is to detect separate parts in a single sprite.
- *Speed*: In the 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 [TexturePacker 2015]: a licensed software for sprite sheet generation. Packs a series of separate sprites and export into a list of specific formats. Exported sprite sheets can be read by engines like: AndEngine (android), Cocos2DX (mobile), V-Play (cross-plataform), Unity and others.

Unity Sprite Packer [Unity 2015]: 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 internal algorithm functionality.

LibGdx Texture Packer [LibGDX 2015]: LibGDX engine has a Texture Packer tool which is a command line application that also packs many smaller images on to 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 [SpriteCutter 2015]: 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 [Construct 2015]: 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 it's 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 to individual sprites, but again, it does not work for multi-sized sprites, only with fixed size sprite sheets.

3. 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 2.1). The second task outlined in Section 2.2 consists of reconstructing the boundary around a given point based on a table of label equivalence.

Both tasks can be presented as a sequential algorithm for tracking images. This algorithm is defined in the following steps:

1. Check if a pixel is foreground or background/transparency.
2. Apply a connected-component labeling. It will define current pixel's label according to nearby labels (neighborhood) or an assigned new label.
3. Reduce labels based on a table of label's equivalence.
4. Define minimum boundaries size 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 [Shneier 1983; Kaspers 2011]. Usually, we will have sprites with a full connectivity within its parts, as seen in figure below:

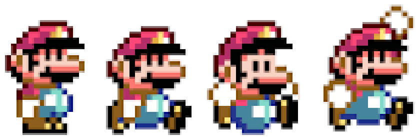


Figure 3: Sprites where all pixels are connected. Source: Super Mario World© game.

In Figure 3, there is no "loose" points: every pixel in the image is connected. Although, that is not always what we see in individual sprite images.

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". An example:

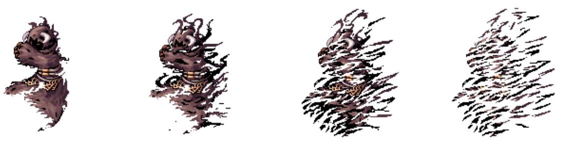


Figure 4: Sprites where some pixels are detached from the others. Source: Ragnarok Online© game.

In sections below, algorithm is described at its entirety.

3.1 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 current pixel's label according to nearby labels (neighborhood) or an assigned 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 [Dillencourt et al. 1992]. A common set of labels are 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 [Dillencourt et al. 1992]. For a binary image, represented as an array of d-dimensional pixels or image elements (Figure 5), 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 [Pordel and Hellström 2015; Shreekanth and Udayashankara 2014].

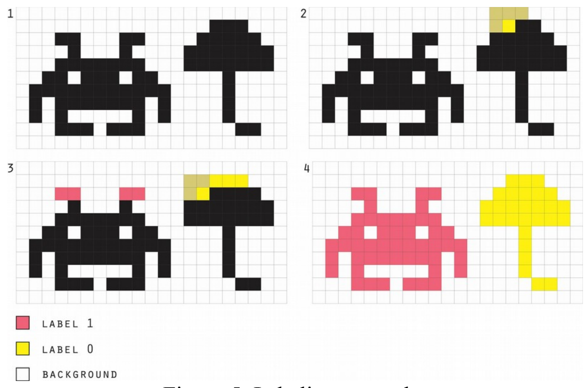


Figure 5: Labeling example.

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. In other case, the pixel classifies as foreground, e.g. it belongs to one sprite.

The connected-component part of our algorithm involves 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 given 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 [Dillencourt et al. 1992]. In our algorithm, we use an 8-adjacent search path, likely: *8-neighborhood connectivity* (Figure 6), to ensure more precise accuracy.

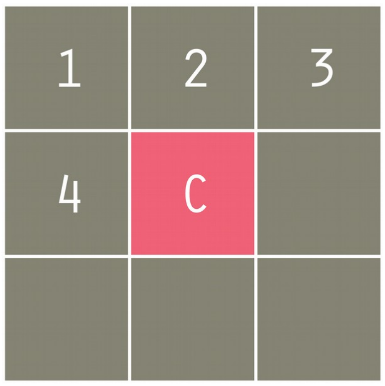


Figure 6: *8-Neighborhood connectivity* search path.

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.

First stage is to circle through all the pixels inside the selected area, verifying corresponding label numbers from neighbor pixels. Figure 7 shows an example on how it is applied:

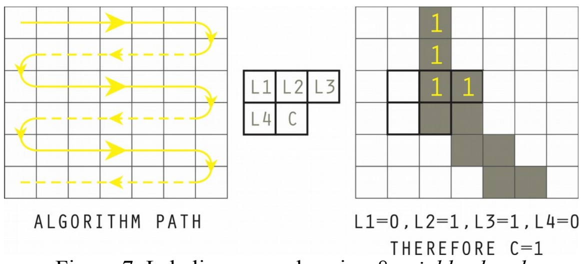


Figure 7: Labeling example using *8-neighborhood connectivity*.

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, is filled up by 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 given 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
```

Figure 8 (below) shows an input monochrome image and the result of the labeling process. It is immediately obvious 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 several distinct is often interpreted by the algorithm as a single blob [Wang et al. 2008].

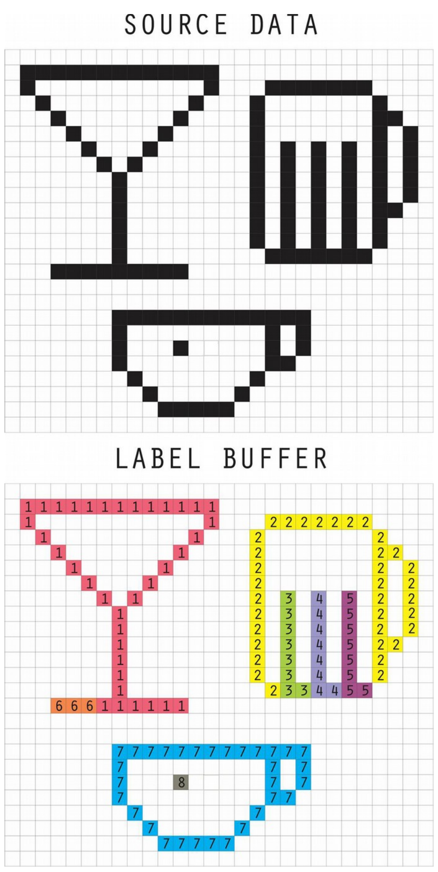


Figure 8: Labeling example (with more than one label per blob).

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 is is easily solved.

Solution is to create a table to keep note of which labels refer to the same blob when the two labeled sections eventually connect. This is described in the next section.

3.2 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 acquired labels based on a table of label's equivalence.
II. Define minimum boundaries size around target image based on identified blob.

Our label connectivity is made based on a two-pass algorithm. The first pass is given by our connected-component algorithm presented in Section 3.1. During that step, the algorithm fills a table of equivalence between associated labels, checking all neighbors to mark equivalence: if multiple neighbors match but got different label marks, assign pixel to first label found and indicate that all of these regions are equivalent.

The filled table of equivalence created in the first pass contains every label in the image and the labels from it's surrounding neighbors too. That makes an equivalence.

After that, the algorithm has a second pass. 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 equi-
      valence label
   end if

done
```

Succeeding that process, the algorithm creates a list for every blob in the selected area and match 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 for the examples shown in Section 3.1. Resulting in corresponding table of label equivalence:

Table of Label Equivalence for Example Figure 8.

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 the illustration (regarding our algorithm, they would both be reduced to label 7). See Figure 9 for an example:

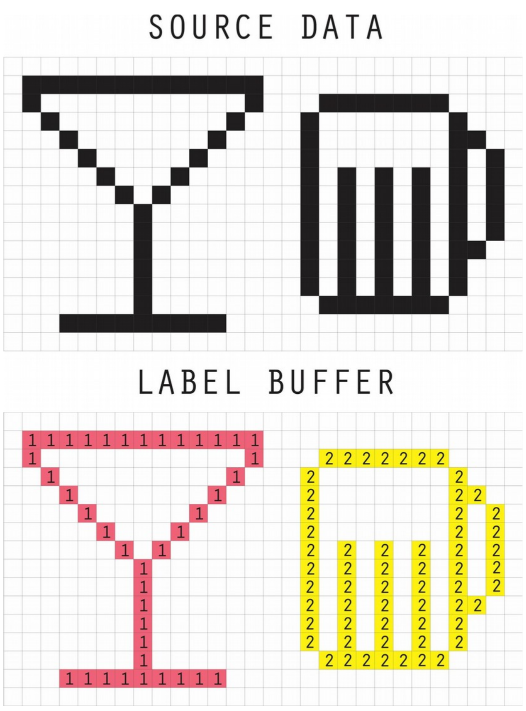


Figure 9: Labeling example (with correct label equivalence).

4. 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 a JAVA program with a simple Swing interface. 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 3 and a parser 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.

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

1. by manually selecting a target area from the sheet; or

2. by selecting the option "Cut entire sheet" from the "Actions" menu.

Option 1 can be achieved if the user clicks and drags the mouse over the image. That way, the Blob detection algorithm is only applied to that sprite sheet subset (Figure 10).

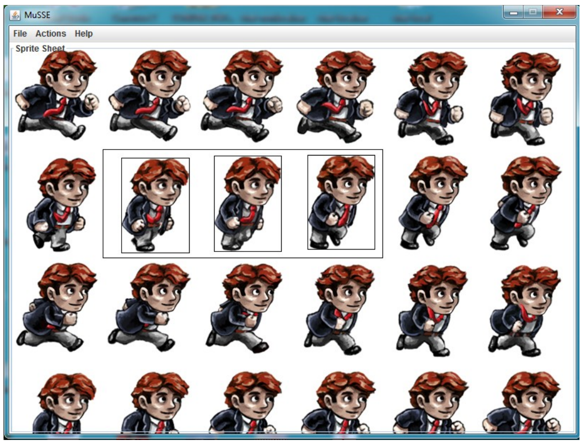


Figure 10: Select target area using MuSSE. Source: MuSSE with sprite sheet image from the game Braid©.

If the user chooses Option 2, the algorithm is applied to the whole sprite sheet image. Figure 11 illustrate the process:

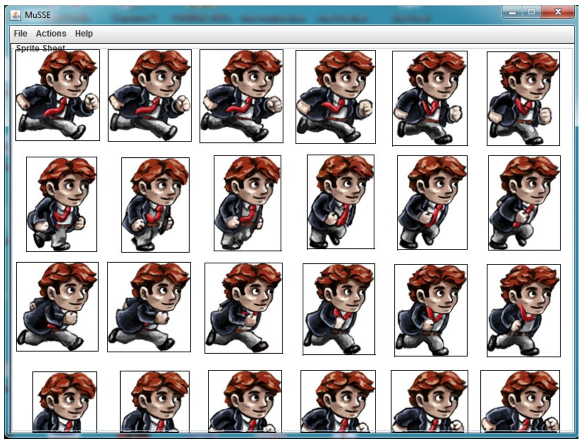
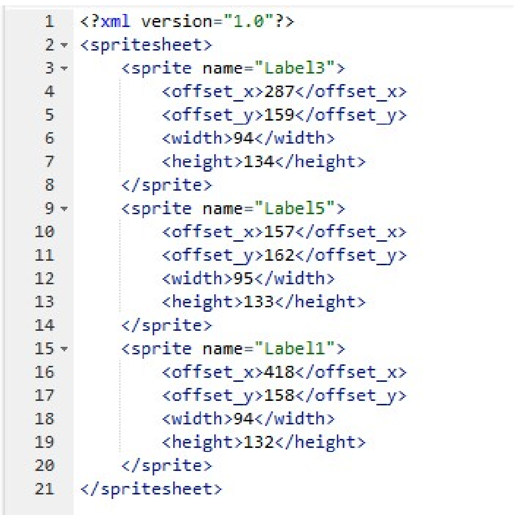


Figure 11: Select entire sprite sheet using MuSSE. Source: MuSSE with sprite sheet image from the game Braid©.

The XML file generated by MuSSE, containing the meta-data from the sprite sheet, has the following format:



```xml
<?xml version="1.0"?>
<spritesheet>
    <sprite name="Label3">
        <offset_x>287</offset_x>
        <offset_y>159</offset_y>
        <width>94</width>
        <height>134</height>
    </sprite>
    <sprite name="Label5">
        <offset_x>157</offset_x>
        <offset_y>162</offset_y>
        <width>95</width>
        <height>133</height>
    </sprite>
    <sprite name="Label1">
        <offset_x>418</offset_x>
        <offset_y>158</offset_y>
        <width>94</width>
        <height>132</height>
    </sprite>
</spritesheet>
```

Figure 12: Example XML created using MuSSE for sprite sheet selection in Figure 10.

Example applications of MuSSE are shown in Figure 13 and 14, where the Blob detection algorithm is applied to entire sprite sheet and to a subset of sprites:

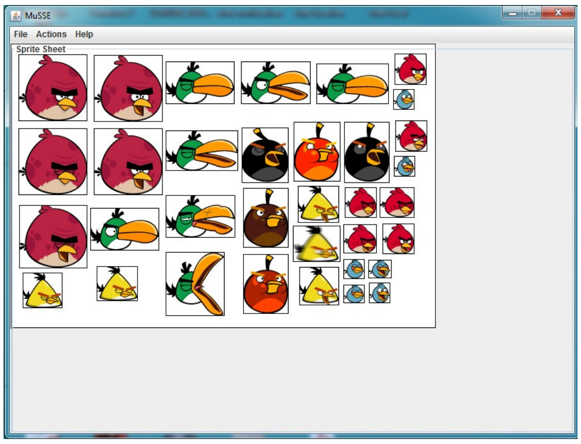


Figure 13: Sprite sheet with high connectivity. Source: MuSSE with sprite sheet image from the game Angry Birds©.

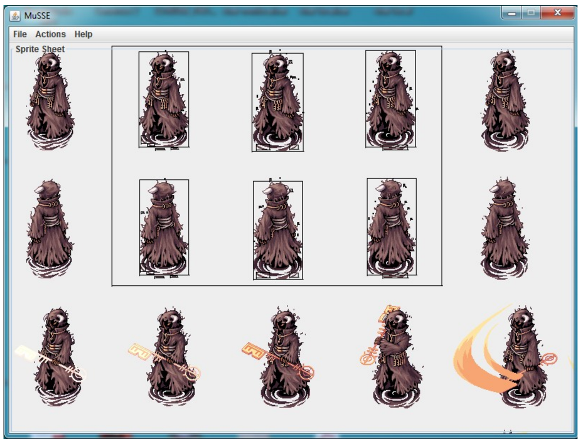


Figure 14: Sprite sheet with low connectivity. Source: MuSSE with sprite sheet image from the game Ragnarok Online©.

Figure 15 (below) shows a close-up from Figure 14. 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 3, that particularity causes the smaller – disconnected parts – to be mapped as new blobs:

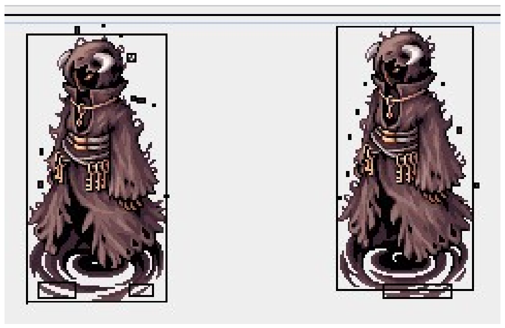


Figure 15: Sprite sheet with low connectivity (close-up). Source: MuSSE with sprite sheet image from the game Ragnarok Online©.

As pointed out before, smaller thresholds for our connected-component method will result in a higher number of blobs in selected image, as seen in Figure 15. That problem can be easily solved by selecting a more suitable threshold for the sheet.

In MuSSE, a user can choose a specific threshold for the Blob detection algorithm by selecting the option "sprite size" from the "Actions" menu (Figure 16).

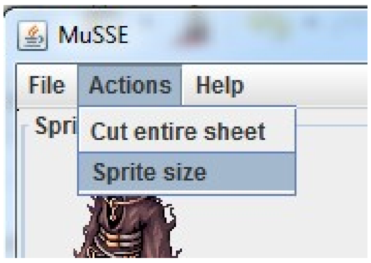


Figure 16: MuSSE's menu.

By selecting a higher threshold, MuSSE will correct the bounding boxes for sprites with isolated parts. As a result, the following sheet can be attained:

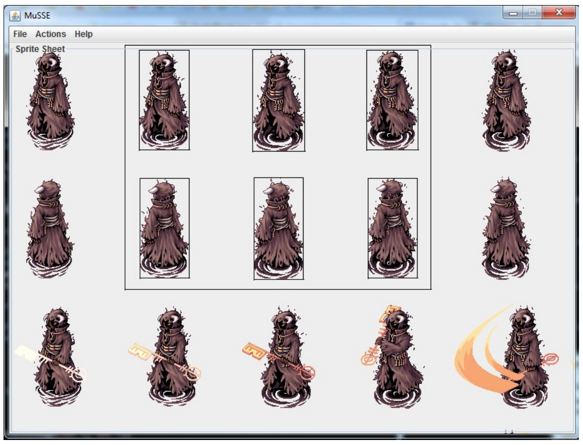


Figure 17: Example from Figure 14 using correct threshold.

5. 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.

That process 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 manipulating 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.

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 get to that point 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.

At last, as a complementary project, we intend to provide an open source common parser library in C++ and/or Java to read XML meta-data generated through MuSSE.

References

BROWN, C., 2014. Advances in computer vision. Psychology Press.

CHANDLER, H. M., 2012. Manual de produção de jogos digitais. Bookman.

COLLINS, R., 2003. Mean-shift blob tracking through scale space. *In: Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on (Vol. 2, pp. II-234). IEEE.*

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

CRISTINACCE, D., AND COOTES, T. F., 2006. Feature Detection and Tracking with Constrained Local Models. In: *BMVC* (Vol. 1, No. 2, p. 3)

DAMIANI, E. B., 2014. Programação de Jogos Android: Crie seu próprio game engine!. Novatec Editora.

DILLENCOURT, M., SAMET, H. AND TAMMINEN, M., 1992. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM (JACM)*, *39*(2), 253-280.

HINZ, S., 2005. Fast and subpixel precise blob detection and attribution. *In: Image Processing, September 2005. ICIP 2005. IEEE International Conference on* (Vol. 3, pp. III-457). IEEE.

IKEUCHI, K., 2014. *Computer Vision: A Reference Guide*. Springer Publishing Company, Incorporated, 701-713.

JENA, K. K., MISHRA, S. AND MISHRA, S. N., 2015. Edge Detection of Satellite Images: A Comparative Study.

KASPERS, A., 2011. Blob Detection. *Biomedical Image Sciences, Image Sciences Institute, UMC Utrecht..*

KOENDERINK, J. J., 1984. The structure of images. *Biological cybernetics*, *50*(5), 363-370.

LIBGDX, 2015. LibGDX Game Engine. Available from: github.com/libgdx/wiki/Texture-packer [Accessed 18 June 2015].

LINDEBERG, T., 1998. Feature detection with automatic scale selection. *International journal of computer vision*, *30*(2), 79-116.

PORDEL, M. AND HELLSTRÖM, T., 2015. Semi-Automatic Image Labelling Using Depth Information. *Computers*, *4*(2), 142-154.

POSSA, P. R., MAHMOUDI, S. A., HARB, N., VALDERRAMA, C. AND MANNEBACK, P., 2014. A multi-resolution fpga-based architecture for real-time edge and corner detection. *Computers, IEEE Transactions on, 63(10), 2376-2388.*

SHNEIER, D., WOO, M., NEIDER, J. AND DAVIS, T., 2007. OpenGL (R) programming guide: The official guide to learning OpenGL (R), version 2.1. Addison-Wesley Professional.

SHNEIER, M., 1983. Using pyramids to define local thresholds for blob detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on, (3), 345-349.*

SHREEKANTH, T. AND UDAYASHANKARA, V., 2014. An Application of Eight Connectivity based Two-pass Connected-Component Labelling Algorithm For Double Sided Braille Dot Recognition. *International Journal of Image Processing (IJIP)*, *8*(5), 294.

SOOKMAN, S., 2006. INSPECTION-Image Analysis Software-Blob Analysis and Edge Detection in the Real World. *Evaluation Engineering, 45(8), 46-49.*

SPRITECUTTER, 2015. Sprite Cutter. Available from: spritecutter.sourceforge.net/manual.html [Accessed 21 June 2015].

TENG, S. W., SADAT, R. M. N. AND LU, G., 2015. Effective and efficient contour-based corner detectors. *Pattern Recognition*, *48*(7), 2185-2197.

TEXTUREPACKER, 2015. Texture Packer. Available from: www.codeandweb.com/texturepacker [Accessed 20 June 2015].

UNITY, 2015. Unity3D Game Engine. Available from: docs.unity3d.com/Manual/SpritePacker.html [Accessed 20 June 2015].

WANG, F., REN, X. AND LIU, Z., 2008. A robust blob recognition and tracking method in vision-based multi-touch technique. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on* (pp. 971-974). IEEE.