Sprite Replacement and Stylization

Aaron Eidelson*
UC Berkeley, Computer Science

Dustin Shean†
UC Berkeley, Computer Science

Abstract

This paper is inspired by previous work on video sprites. A sprite is a two-dimensional image or animation that is integrated into a larger scene. We want to explore methods of creating sprites of a player, and inserting them into an already existing video game. The problem of video game sprite replacement can be split into both a user interface problem and a stylizing problem. We hope to address both of these problems with our system.

1 Introduction

In recent years video games have become a major source of entertainment. A survey released this year showed that the number of video gamers has surpassed the number of movie goers in America. "Almost 64 percent of Americans have played a video game in the past six months versus only 53 percent who have gone out to see a movie, according to a report from market research firm NPD Group." [1] One might ask what makes gamers want to spend extended amount of time in a virtual world? Many would say it is to escape into another world, yet films are also able to capture this theme. We, along with many others, believe the ability to reach an achievement in games is what makes them more entertaining than movies. This fact leaves us trying to answer the question of what can make gamers more enticed to play the already popular games?

Our approach is motivated by the lack of personalization in most games. Games based on virtual items and in-game character customization have become increasingly popular in recent years. But despite market trends and the increasing quality, reliability, and prevalence of capture devices (such as microphones and cameras), there is surprisingly little in the way of automatically customizing games to fit the player. We think that using these capture devices to insert the player into the game is a natural next step for the gaming industry, and will keep players more interested and emotionally engaged.

For exploring possible user interfaces and methods, we chose to build off of the Open Sonic game (or Sonic for short). Sonic is a platform side-scrolling game in a pixel art sprite style. While a sprite can be just about any image that is part of a larger scene, pixel art refers to a specific type of sprite typical of most games from the late 80's and early 90's, where the sprite is drawn by hand. This style originates from hardware limitations of the early gaming industry imposing low resolutions for sprites, but has since been adopted by game makers seeking a retro style, or indie games looking to set themselves apart. An important part of our approach is stylizing the image so that it looks as if it were drawn by an artist for placement in a pixel art sprite game.

(Note: Throughout the rest of this paper, sprite will be used interchangeably with pixel art sprite)

2 System Overview

This paper provides a system for inserting images of a user into a sprite game. The system is formed by five stages:

Input The user captures images of him or herself performing a preset list of actions.

Template replacement Images of actions being performed are mapped by the user to the pre-defined actions.

Preprocessing Images are prepared for stylization and insertion into the game. This includes matting, image fitting, and color adjustments.

Stylization Images of the user performing actions are scaled to the correct size and stylized to fit the look of the game.

Image transfer The modified images of the user are inserted into the correct sprite file, replacing the original game content.

3 Input

In our system, the user is able to choose from using either images, videos, or a combination of the two as input *(see Figure 1)*. For ease during the pre-processing stage of the pipeline, we tried to act in accordance with the following recommendations:

- Images and videos should be taken against a solid background
- Images and videos should be taken at the same time of day or lighting
- The user should not wear any clothing that is the same color of the background
- Videos should be short in length

Good results will be more difficuilt to produce if the above recommendations are not followed.

Figure 1: *Example input photo*

*aeidelson@gmail.com
†shean.dustin@gmail.com

[1] http://news.cnet.com/8301-10797_3-10245437-235.html

A limitation of our current implementation is that the user interface only works with images as input. However, video can be easily converted into a sequence of images for input into our system.

4 Template Replacement

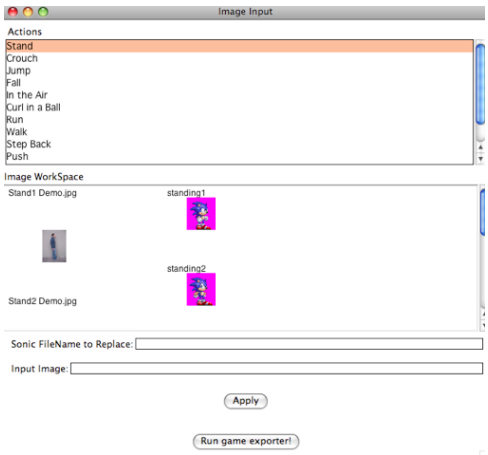


Figure 2: *Template replacement interface. On the top is a list of selectable actions. After the action is selected, the Sonic reference images appear on the right and possible replacement images appear on the left. Entering the file indicies in the text fields creates a mapping from the user's images to the Sonic images to be replaced. Pressing the "Apply" button creates the mapping, and pressing the "Run game exporter" button begins the stylization and image transfer steps.*

Template replacement is a manual stage of the pipeline that occurs in the user interface. In this stage, the user clicks through the known actions (12 actions in the case of Sonic), and matches their input images with the static reference images *(see Figrue 2)*. This step has the most effect on the end output of our system, so it is important to keep the interface as simple and clear as possible to avoid user errors.

A user-facing optimization that we found was neccesary in our interface was to allow users to replace images by just referencing the images through numbered indexes instead of typing the entire filename. Once we added this feature we saw a decrease in the ammount of time for more willing users to find the best match. When trying to find the best match, the system is programed to allow the user to replace an already replaced image with a better match. In order to provide this functionality the static Sonic action images are always going to be displayed even if it was already replaced.

In our "Future Work" section, we note ways that we can still allow for the creative input of the user, while speeding up the matching process.

5 Preprocessing

This stage requires the most work from the user, but we believe there are many ways we can automate the most time consuming tasks. Please see the "Future Work" section for details.

5.1 Alpha Matting

Alpha matting is when there is a construction of an alpha channel, which is used to decipher what part of an image is in the foreground versus the background. During this stage the user must input information depicting this foreground and background difference, thus defining an alpha channel, for each image that was chosen by the user during the Template Replacement stage. Currently the user uses Adobe PhotoShop to construct the alpha channel, producing an image similar to *Figure 3*.

Figure 3: *An alpha matted picture of Aaron performing the "falling" action for replacement of the Sonic sprite. The checkered background represents areas of the image with an alpha of 0, whereas colored parts of the image have an alpha of 1.*

5.2 Image Fitting

After the image is matted, it must be scaled and cropped to the correct proportions. In the case of the Sonic game, the image must be proportional to 40x45, and the person should take up almost the entire height of the image while standing.

5.3 Gamma, Contrast, and Brightness Adjustment

Images taken, even in the same location with the same camera, can exhibit very different lighting effects. This isn't especially noticeable when just viewing the images, but it becomes a problem when sprites are played in rapid succession throughout the playing of a game. Manually adjusting the gamma, contrast, and brightness of each sprite image to match a benchmark image produced better.

6 Stylization

We will rely on a number of stylization heuristics observed across all game sprites to automate the process of converting a high resolution true color image to match the style of a game sprite.

6.1 Properties of a Sprite

When examining sprites *(Figure 4)*XXXXXXXXX, it is apparent they share a number of attributes.

- First, they look very pixelated. This is a byproduct of upscaling from a very low resolution. But despite their low-resolution nature, it is still very clear what the pictures depict.

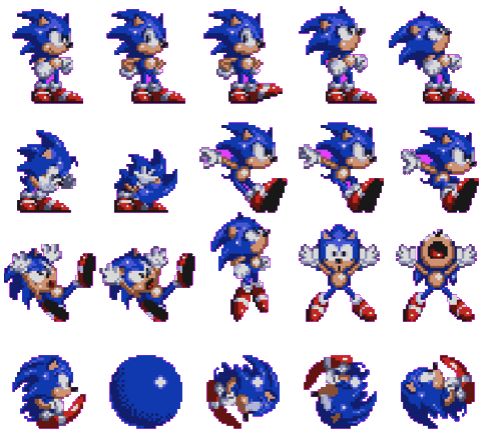


Figure 4

- Second, each sprite is made up of a relatively small number of colors. This keeps the sprite from looking too busy and gives it more of a cartoon appearance
- Third, authentic sprite games operate in the 8 bit color space.

6.2 Scaling

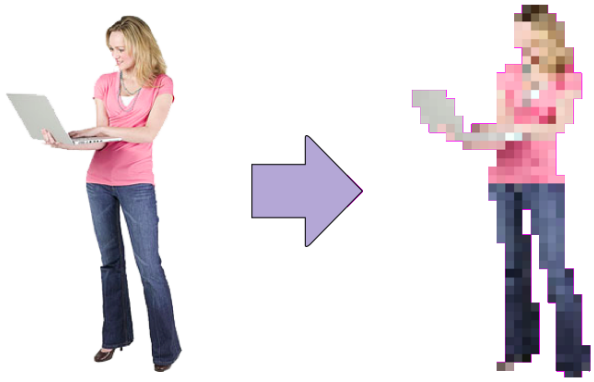


Figure 5: *The result of scaling a test image from full-size to 40x45, using a nearest-neighbor scale.*

As previously stated, sprites generally have a very low resolution. In the case of Sonic, this resolution is 40x45pixels. As a first step, the input images are scaled to the correct sprite size *(Figure 5)*. This is done as a nearest-neighbor scale, as we want to preserve hard edges and color regions.

6.3 Clustering Algorithm

While our image is the correct size, it contains too much detail. In our experience, an image looks odd and out of place when inserted into Sonic after only a scale. To remedy this, we will return to our second property of sprites; we must reduce the number of colors that make up the sprite. One of the most well-studied ways of doing this is through a clustering algorithm, and we decided to use k-means clustering. Our application of the k-means algorithm is as follows:

We define the distance between two colors as the least squares difference in LAB space.

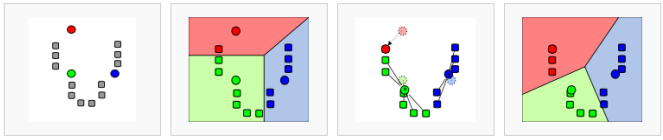


Figure 6: *A visual representation of k-means clustering. From left to right, Image one: Initial means are chosen, Image two: Pixels are assigned to the closest cluster, Image three: Cluster mean, Image four: The process is repeated*

1. Pick an initial mean color for each cluster
2. Assign each pixel to the cluster with the closest mean color
3. Re-calculate color means of each cluster based on the pixels belonging to that cluster
4. Repeat 2 and 3 until convergence

(see Figure 6)

There are two items we must supply to this algorithm: the number of clusters and an initial mean for each cluster. We decided that the number of initial clusters should be supplied by the user. As for the initial means, we wanted to pick means which encouraged clusters with commonly occurring colors, but discouraged making the initial means of any two clusters too similar. We feared that if two initial colors are too similar then most of the detail that humans deem important (i.e. facial features) would be averaged out.

To try and solve this problem, we first tallied up every color occuring in the image, and then applied a scoring equation to produce a score for each color:

$$d = \text{minimum distance from clusters chosen so far} \quad (1)$$

$$score = log(\#\text{ of occurrences} + 1) + c \cdot log(d + 1) \quad (2)$$

c is a user-defined constant, and the logs serve to mitigate the effects of a large number of occurrences or minimum distance. At each iteration of picking a new cluster mean, the scores of the remaining colors to be considered are re-scored and the color with the highest score is chosen and removed.

In practice, because we are dealing with such low resolution images, varying c gave unpredictable (but different) results. Please see the "Future Work" section for ideas on helping the user pick appropriate values for the number of clusters and c.

After the clustering algorithm is done, we re-color each pixel to the mean color of the cluster it belongs to.

6.4 8-Bitification

Most RGB images today are represented in 24 bits (8 for red, 8 for green, and 8 for blue), producing over 16 million colors. But as mentioned earlier, authentic game sprites use 8 bits for color, making 256 colors. We scale and round each pixel's color to using 3 bits for red, 3 bits for green, and 2 bits for blue, and then re-scale the values back up to using all 24 bits for display. This has the effect of snapping each color to its closest 8 bit color.

After stylization is complete, we reach a sprite similar to the one in *Figure 7*

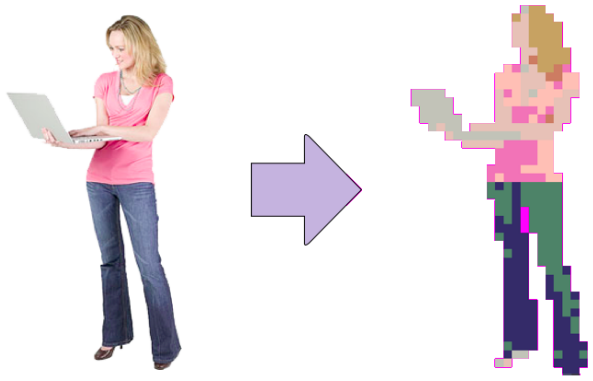


Figure 7: *The result of scaling, color clustering, and 8-bitification on a test image.*

7 Image Transfer

During this stage, all of the stylized images are imported into the desired location in the sprite file. It is the only direct interaction between the interface and the game. Thus if we wanted to extend our approach to a different game, the inner workings of this step would need to be change. However we would want the user to still follow the same approach and therefore have made this step an automated black box to the user.

8 Results

Although our approach works for both videos and images, we originally had the plan of using only videos in order to allow an easier interface for the user. However through experimenting fully with images, and fully with videos, we found that each implementation had its pros and cons in the character output gameplay. We decided that for our demonstration we would use a combination of both images and videos as input.

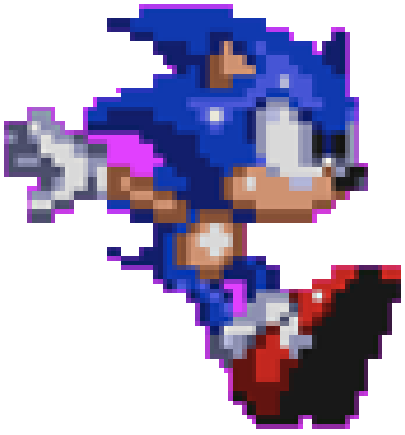


Figure 8: *Sonic's stopping sprite.*

The majority of the video input was taken from actions that provided a very fluid motion. For example the walking and running replacement was taken from videos. On the other hand, while using the Sonic Game, we discovered that some of the movements

Figure 9: *Aaron's imitation of Sonic's stopping sprite. This picture had to be captured overhead with Aaron on the ground, because it was so difficuilt to balance.*

made by Sonic are so overly exaggerated that they are not physically possible for a human user. One example of this was when Sonic throws his arms back and also his legs forward in order to stop his movement *(Figure 8)*. In order to get a resembling result we had to position the user on the ground and take an aerial view *(Figure 9)*.

We found that sprites which were made of fluid motion (such as walking or running) were most easily replaced by video, whereas sprites with just a few frames or where the player is in a specific pose is more easily replaced by still images.

Overall, the end result is very satisfactory. The Sonic sprites are successfully replaced by stylized versions of the input images. The replacement is demonstrated in *Figure 10* and *Figure 11*.

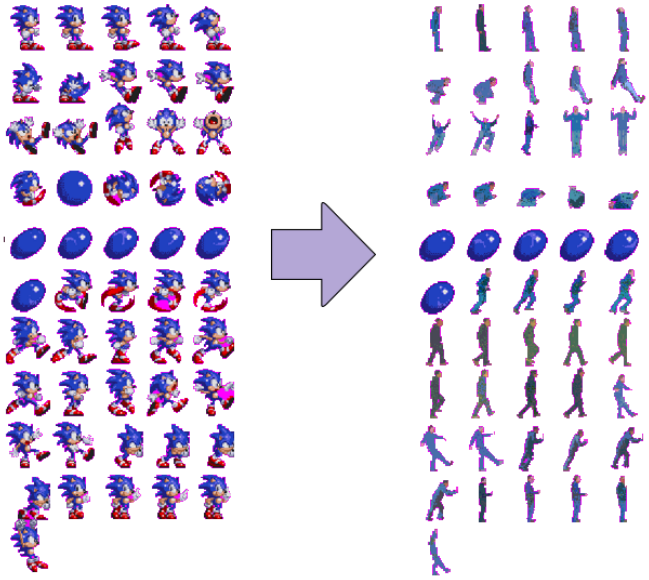
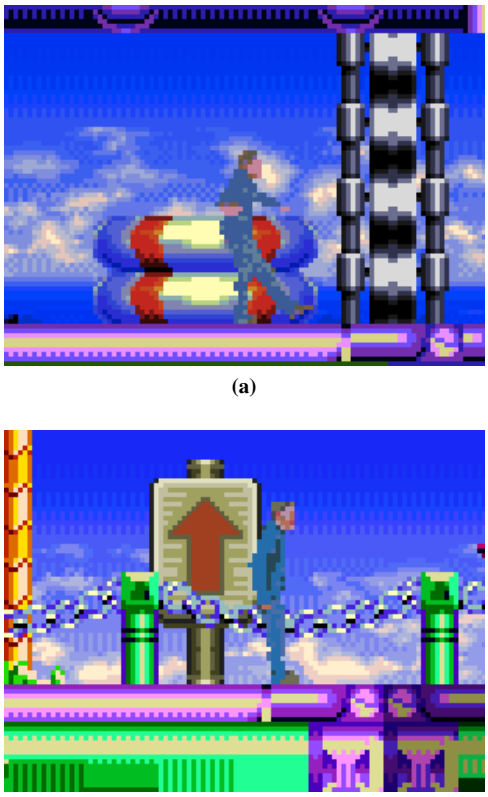


Figure 10



(a)

(b)

Figure 11: *A picture of the final game after Sonic's sprite is replaced by Aaron's.*

8.1 Efficiency

Efficiency was not our main concern in this project, but we did make some notable high level optimizations. It takes around 15 seconds to stylize approximately 40 images (including clustering), despite using straight Python for clustering and 8 bit conversion. We credit this speed to downsizing the images to sprite size before performing any stylization.

9 Future Work

One possible improvement to our input stage would be to automate all inputs to work with our interface. Currently the user must manually convert videos to image sequences using an application such as VLC. We could provide an extra button on the interface labeled "Convert Videos", which would automate this process. This would hopefully speed up and simplify our user interaction steps.

We believe that our "template replacement" stage would benefit by making use of the Microsoft Kinect and its API. The Kinect would allow this stage to become fully automated. Since the Kinect can provide applications with approximate human skeletons, our application would only need to compare the Kinect-provided skeletons with known sprite skeletons to replace all necessary action images *(see Figure 12)*. As a fail-safe, there could be an optional correction phase to deal with failed matches.

Since the alpha matting portion of the preprocessing stage is one of our worst time bottlenecks for the user, we think it would be beneficial to have a matting implementation built into the interface. It would be relatively easy to implement, given the large number

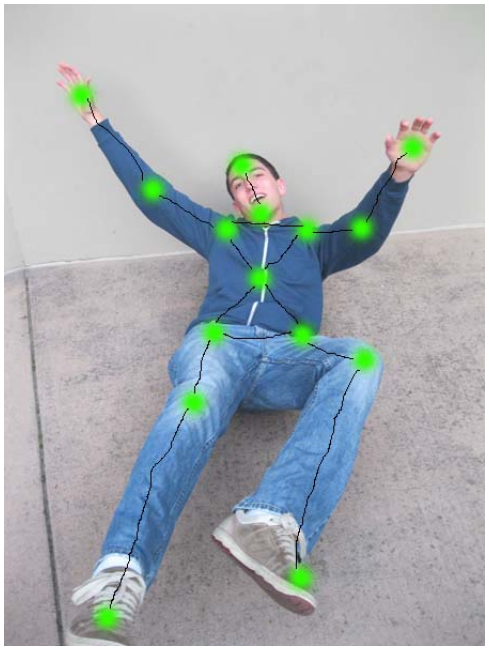


Figure 12: *Possible bone structure generated by the Kinect. Green dots are joints and black lines are bones.*

of well-developed algorithms. Integrating it it into the the interface could also allow the user to itteratively improve the matting, seeing the end result in real-time *(see Figure 13)*.

A different approach would be to use optical flow on video input. This approach would allow us to combine improvements in the "Template Replacement" and "Alpha Matting" steps. In this scenario the user would draw a skeleton on the first frame of the video (or we would use the Kinect skeleton) and color in themselves in order to be labeled as foreground *(see Figure 14)*. Using a method similar to this one, we could close-to-automate both the "Template Replacement" and the "Alpha Matting" steps. However, the method might end up incurrng some user adjustments (due to the limitations of optical flow), as well as drastically increasing the computation time.

Our system would provide more fluid transitions in sprite animations if there was consistent color between sprite images. The problem is that while a shirt's color may not change, the lighting conditions or the point of the image used by the camera for white balance might change. This can result in very different lighting conditions for sprites in the same sprite animation sequence *(see Figure 15)*. To deal with this we propose two modifications to our system: first, the scaled images should be automatically adjusted to have a similar color range as some baseline image. It may be helpful to try and fit the image's color histogram. After the colors of the images are as close as possible, we then perform the clustering algorithm across all of them at the same time. If we use the same cluster means acrosss all the sprites at the same time, the final sprites will all be made up of the exact same colors. We believe these changes will provide a kind of temperal coherence between sprite frames.

We believe there is also a lot more potential for more stylization of the user before insertion into the game. Our initial attempt was to directly replicate the player in the game, but we have noticed that virtually every cartoon and sprite character has much more exaggerated features (hands, shoes, head) than humans normally do. Skeletal information could allow us to isolate these features and

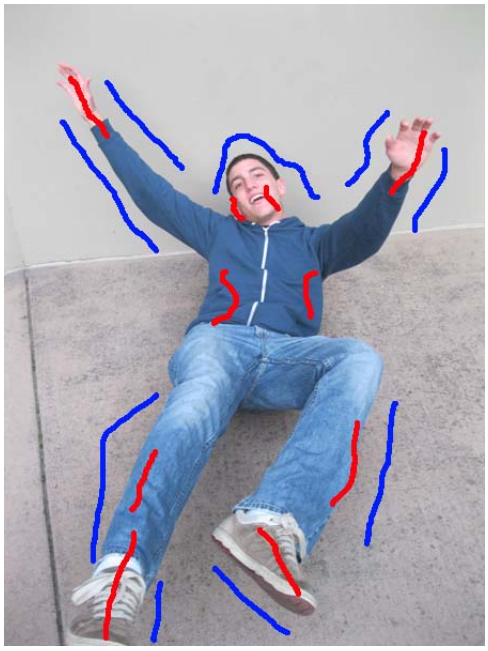


Figure 13: *Possible alpha matting interface. Blue strokes identify background and red strokes identify foreground.*

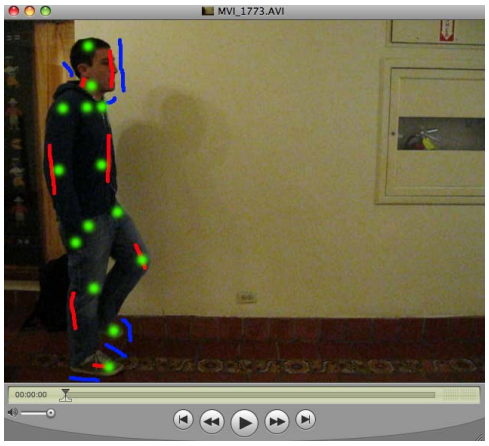


Figure 14: *A possible interface for using optical flow. Green dots represent joints and red/blue strokes identify the foreground and background.*

increase their size.

Currently, picking values for the number of clusters and the c term in the scoring equation requires the user to guess and check. The user would benefit from seeing a few options for each term and being able to pick the best.

10 Conclusion

We have demonstrated an interactive system for mapping and stylizing an alpha matted real-world object for placement into an 8-bit game.

Although our system is fully functional, the implementation of our "Future Work" section would greatly improve the user experience. While in its current incarnation the system requires a good amount of user input, we believe our approach gives an example of how a simple and intuitive user interface in combination with powerful algorithms can open the door to a more personalized gaming experience.

11 Acknowledgements

The authors would like to thank our testing users for their contribution of providing ample feedback on the stylization of the sprite replacing gameplay. The authors would also like to thank the makers of Grapefruit (color space converter), PMW, and Python Image Library for providing the authors with valuable tools. Lastly the authors would like to thank the developers of Open Sonic for creating a fun virtual world to test our approach.

(a)

(b)

Figure 15: *Two images of the same person performing different actions, exhibiting very different final colors.*