1 Abstract

While 2D barcodes are typically used to store links to websites, they can be used to store any type of data. In particular, a 2D barcode attached to an object could store information about how to generate an animation of that object. An author could specify transformations for parts of the object, and these transformations could be encoded into a barcode along with features of the parts of the object to be transformed. Such a system would allow a user to watch a video of the animated object, even if they did not have access to internet.

I propose a new approach to encoding and decoding how to animate an object: using image segmentation. In such a system, the author of the 2D barcode runs an image segmentation algorithm, and specifies transformations for segments. Features of the specified segments are encoded into the barcode. The user also runs an image segmentation algorithm on their photo of the object, and matches the segments specified by the author using the features encoded in the barcode.

The trade-offs between different segmentation algorithms and different choices of features for matching are discussed in this paper. I provide a segmentation and matching system, written in Python, which can integrate with a larger effort in the Yale Graphics Lab to build an end-to-end barcode animation system. I also provide a GUI for the system which greatly simplifies the process of testing the matching algorithm. The system’s performance is illustrated with a variety of different example use cases.
2 Introduction

2.1 Generating Videos From Barcodes

A proof of concept for a 2D barcode animation system was developed as a CPSC 490 Project in Spring 2017\(^1\). The system works as follows:

- The *author* takes a picture of an object they would like to animate.
- The *author* drags polygons onto the image and can specify transformations for these polygonal regions.
  - These transformations are encoded into a QR code, which the author places on the object.
- The *user* takes a picture of the object with the QR code, and decodes the information in the code.
- The *user* generates a video using the encoded information:
  - The specified polygons are matched to shapes in the new image, so that the *user* can view an animation of the object.
  - As the polygons are animated, the background is filled using background inpainting.

One of the most critical aspects of the animation system is the matching of the polygons. If the author does not align the polygons to distinctive edges of the object, the user side will not be able to match regions of the image to the specified polygons. For this CPSC 490 project, I explored how to better solve this matching problem using image segmentation.

2.2 Image Segmentation

Image segmentation algorithms divide an image into segments which are perceptually meaningful. One particular class of segmentation algorithms are superpixel segmentation algorithms. Rather than trying to divide an image into large irregularly shaped parts, superpixel segmentation algorithms oversegment an image, grouping the pixels into roughly uniform superpixels. *Figure 1* shows an example of standard image segmentation\(^2\), while *Figure 2* shows an example of superpixel segmentation\(^3\).

\(^1\)http://zoo.cs.yale.edu/classes/cs490/16-17b/chen.qingyang.qc43/
\(^2\)https://nicolovaligi.com/example.png
\(^3\)http://cs.brown.edu/courses/csci2951-t/finals/ghope/img/slic.png
Figure 1: Standard segmentation example
Figure 2: Superpixel segmentation example
The goal of this project was to use segmentation algorithms to improve the existing system for generating animations from 2D barcodes.

3 Design

3.1 Language

The system was developed in Python 3.5, and made use of the scikit-image library of image processing algorithms for Python. While the previous version of the matching system was developed in MatLab, I chose to use Python instead because of its relative portability to mobile devices, especially when compared to MatLab. Additionally, wrappers for most C/C++ Computer Vision and Image Processing libraries exist for Python.

3.2 Segmentation Algorithms

The scikit-image library provided access to a variety of different segmentation algorithms. The first design decision to be made was the choice between using standard segmentation or superpixel segmentation algorithms.

If standard segmentation algorithms consistently segmented images ”correctly”, using them would simply the process, since only the animation of a single segment or small number of segments would need to be encoded in the QR code. By comparison, encoding the animation of some shape as specified by a set of superpixels would require storing features of each superpixel. This would require storing more data per region of interest, and make matching more difficult.

However, using superpixel segmentation would offer more flexibility to the author. If the chosen segmentation algorithm failed to capture the desired shape in a single segment or group of segments, the author would be forced to take a new picture or specify animations for the available segments. As the number of segments increases, the likelihood of this error in segmentation decreases, as most features consist of multiple increasingly small segments. Given that most segmentation algorithms do not work well for every input, I chose to work with superpixel segmentation algorithms. My goal was to match as well as possible, given an adequate superpixel segmentation.

The two superpixel segmentation algorithms which qualitatively seemed to perform the best were SLIC ⁴ and Quickshift ⁵. I chose to use the Quickshift algorithm because it more exactly

⁴http://ivrl.epfl.ch/research/superpixels
⁵http://vision.ucla.edu/papers/vedaldiS08quick.pdf
tracked the edges of objects, even though it produced less uniform superpixels. Figure 3 displays a comparison between scikit-image implementations of superpixel algorithms, including SLIC and Quick Shift\(^6\).

\(^6\)http://scikit-image.org/docs/0.13.x/images

Figure 3: Scikit-image superpixel segmentation examples

Felzenszwalb’s method  SLIC

Quickshift  Compact watershed
3.3 Feature Matching

A select number of features are encoded into the QR code for each segment to be animated. Since the storage of a 49x49 QR Code is limited to about 194 bytes\(^7\), the number of stored features is also limited and depends on the maximum number of segments which we allow to be used in an animation.

The system assigns a weight \(w_i\) to each feature \(f_i\) and minimizes the euclidean distance between weighted feature vectors. Let \(f\) be the feature vector to be matched from the author image, \(w\) be the vector of weights for each feature, and \(F_m\) be the set of all feature vectors \(m\) in the user image; then a match is given by:

\[
\arg \min_{m \in F_m} ||(f \cdot w) - (m \cdot w)||
\]

One goal of this project was to determine what features would be useful for this matching process. In the following sections, I list the features that I tested, and suggest how useful they may be (and therefore how they should be relatively weighted) based on qualitative results.

3.3.1 Position

I began by storing the average pixel position of each segment, but I found that absolute position was not always sufficient to accurately match segments. I was able to improve performance by using relative positions. This was facilitated by the existence of a QR code in every image: since the position of the QR code must be detected by the system in order to decode the data, the position may be matched relative to one of the position markers in the QR code.

Note that using relative segment position as a feature does complicate the system, since the author will not be able to take a picture of the QR code before generating it. Therefore, the author must use a placeholder QR code when taking the initial image, and replace that QR code with one that holds the encoded feature data.

3.3.2 Size

Since superpixel algorithms aim to produce segments of similar size, size was not necessarily a useful feature for matching. In some cases adding size as a feature lead to a successful match, and in other cases (when a large object was over-segmented in different ways in each image) matching based on size degraded performance. One can see an example of this issue in Figure 4 and Figure 5: while the left hand diamond is correctly matched, the middle diamond will not be matched correctly as it is divided into to superpixels in one image, and one in the other.

\(^7\)http://www.qrcode.com/en/about/version.html
Figure 4: Diamond pattern
Figure 5: Diamond pattern, segmented
3.3.3 Color and Intensity

I found color and intensity to be useful features to encode, if possible. Encoding color as a feature is “expensive”, given that it requires encoding 3 integers, so intensity may be a good substitute depending on storage constraints. Figures 6, 7 illustrate a water heater example which fails without encoding color features.

3.3.4 Segmentation Parameters

The main input parameters to the Quickshift segmentation algorithm are the kernel size $k$, and the max distance between pixels $m$. Increasing either of these parameters will increase the average size of the generated superpixels. Since these parameters can themselves be encoded in a QR code, they can be varied based on the detail of the image. Most of the simple examples used in this report work well with $k = 11$, $m = 40$, but there exist other cases for which different parameters are required. Figures 8, 9 illustrate an example which yields a reasonable match for $k = 5$ instead, given the detail of the image.

I also found that matching performance can be improved in some cases by using smaller segments on the author image, and allowing more than one segment in the author image to match to the same segment in the user image (by default, matches are bijective). I think this could be an interesting technique to explore further in the future.

4 Deliverables

4.1 Integration with an End-to-End System

Over the course of the semester, the project expanded to include the development of a new end-to-end system which would take advantage of matching via-segmentation as explored in this project. Other students working in the Graphics Lab built systems for QR code detection, animation specification, and background inpainting, while I extended my segmentation and matching system to integrate with the other work that was being done. To that end, the system required a command line interface for running segmentation and segment matching, as well as a schema for the files input to the matching system and output by the segmentation and matching systems.
Figure 6: Yale water heater
Figure 7: Yale water heater, segmented
Figure 8: My clarinets
Figure 9: My clarinets, segmented
4.1.1 Author Side

The typical usage is `python3 superpixels.py -i /path/to/image`. The `-i` flag specifies the path to an image. On the authoring side, only segmentation is required, so no matching occurs. The index matrix, which specifies a segment index for each pixel in the image, is written to `index_matrix.csv` while the feature data for every segment is written to `raw_matching_data.csv`. These are used as input to the next stages of the full system.

4.1.2 User Side

The typical usage is `python3 superpixels.py -i /path/to/image -d datafile.txt -m`. The `-d` flag specifies the matching data, the information stored in the QR Code. The `-m` flag specifies that a mask should be generated, in which targeted segments are colored white and the rest of the image is colored black. Figure 10 illustrates an example of a mask output by the program. If multiple keyframes (regions of interest) are specified, a separate mask is generated for each.

4.1.3 Matching Data Format\(^8\)

`datafile.txt` is formatted as follows:

The first line contains 8 floating point numbers rounded to 1 decimal place: these give the \((x, y)\) coordinates of each of the 4 landmarks in the QR Code.

The second line contains an integer \(m\), which gives the number of keyframes (distinct groups of segments with associated transformations).

Each of the following \(m\) lines begins with an integer \(n\), giving the number of segments in the keyframe. It is followed by \(kn\) space separated integers, where \(k\) is the number of features. So, the first \(k\) integers will be the features for the first segment to be matched, etc.

Example file \((k = 3)\):

```
497.5 442 496.5 347.5 592 343 581 426.5
2
1 118 170 11689
2 517 170 11081 230 300 7845
```

\(^8\)Credit to Zeyu Wang for this format
Figure 10: Example mask
4.1.4 Other Flags

- **-di**: By default the output of the system is a series of files, `output_0.png`, `output_1.png`, ... . When the `-di` flag is used, the system will instead display the output images as a series of plots using Matplotlib.

- **-s pixel_distance**: This flag may be used to specify the maximum distance between pixels in a given segment (larger values will yield a smaller number of superpixels).

- **-q qrfile.txt**: This flag may be used to specify the position of marker(s) in the QR code, so that positions of segments in each image can be compared relative to a known correspondence.

4.2 Matching Interface

I found that while the command line interface was necessary for integrating with the larger system, testing features and specifying segments and QR code locations using the command line interface was tedious. In response I designed a Graphical User Interface (GUI) for the testing of matching algorithms. The GUI does not support the full process of encoding animations, but instead is meant to be used as a tool for further development and testing of the matching system on new examples.

*Figures 11-16* show the user flow for the GUI. The program can be launched by running the command `python3 gui.py`

I developed the interface using the Tkinter library, the standard library for user-interface development in python. This was a learning experience for me, as I not previously worked with Tkinter or with many event-driven front-end frameworks.

5 Conclusion

Overall, this segmentation-based matching system seems to perform better than the polygon-based system previously utilized. As seen in the example figures, the matching works well for simple images, though more work will be needed in order to succeed at matching regions of interest in more complex images. My initial goal was to match as well as possible given an adequate segmentation. While I do not think that I have exhausted the possibilities for good features to use when matching, I suspect that more attention may need to be paid to the segmentation aspect of the system. That is, there may be an upper bound on how well matching can perform, given a questionable segmentation.

I’ve included a final example as *Figures 17-18*. 
6 Future Work

As discussed above, the segmentation and matching system will be used as one part of a larger end-to-end system. The end goal of the user side system to produce an Android app; therefore the current python code will need to be migrated using available libraries for python-to-java/android app translation.

I was motivated to build an interface in python rather than using web development tools because of the possibility of creating an interface structured around hierarchical segmentation: multiple layers of segmentation. Such an interface might allow the author to choose between multiple segmentation "levels" based on which one allows them to most accurately encode their region of interest.

If enough pre-matched images were created, the feature weights could be calculated using machine learning. At the moment, such pre-matched images do not exist, but I believe that they could be constructed using datasets available for other problems in Computer Graphics. For example, sets of pictures aligned using structure-from-motion algorithms could be used as input. Pairs of these images taken of an object from similar positions would have known correspondences, and approximate or closest segment matches could therefore be calculated. These matches might be used to train a model to be used as a matching algorithm.

Figure 11: The user selects an author image
Figure 12: The user clicks on a pixel in the top left marker of the QR Code
Figure 13: The user selects segments to match
Figure 14: The user loads a user image to match
Figure 15: The user clicks on a pixel in the top left marker of the QR Code.
Figure 16: The selected segments in the author image are matched to segments in the user image.
Figure 17: Hand sanitizer example
Figure 18: Hand sanitizer example with matched segments
7 Acknowledgements

I'm very grateful to Professor Rushmeier for advising me over the course of the semester and for organizing weekly meetings with other CPSC 490 students working on graphics-related projects. I’d also like to thank Zeyu Wang and Sherry Qiu for working on the end-to-end system with me.