CPSC 490 Project Report: *Generating Videos from Barcodes*

Qingyang Chen — Advisor: Holly Rushmeier

May 3, 2017

1 Abstract

Imagine being able to scan a visual code on any object and see that object come to life. QR codes, for example, can be used to encode URLs for videos that can be watched over the Internet. However, this requires the video be stored on a remote server and the viewer to be connected to the Internet.

This project solves these two issues by generating the video directly from an image of the object itself that contains the visual code. The prototype involves encoding animations into the visual code and using computer-vision techniques to process an image and animate its parts.

Code: https://github.com/coollog/VideoBarcode

2 Vision

Imagine a still picture or an inanimate object coming to life just by taking a photo of it. Imagine taking a picture of an object and seeing how that object is used. Imagine taking a picture of a poster and seeing the still image turning into an animation. Imagine going to an art installation and seeing the different pieces interact with each other.

This report details a prototype of a computer vision-based technology that can turn any real-life still object or image into an animated video, without the use of any external information or communication (such as the Internet). This means that the information for the video is encoded in the object or image itself.
The vision is for manufacturers to be able to show how to use a product without separate instructions, for advertisers to be able to create low-cost still-image billboards and posters that can animate without the need for an electronic screen, and eventually for augmented reality headsets to be able to transform a scene rather than just overlay holograms.

3 Solution

The goal is to encode a representation of the animation or video into the photographed object itself. QR codes, for example, can be used to encode URLs for videos that can be watched over the Internet. In this case, QR codes will be used to encode the video itself.

The main problem to solve is how to encode a video into a visual code. For example, a 49x49 QR code can store about 194 bytes of data with low error correction [1]. Even a minimally acceptable video bitrate of 16 kbps [2], a one-second video would already be over 10x the storage capacity of this QR code.

The solution is to reduce the amount of data to be encoded in the visual code as much as possible. Specifically, the visual code should encode just the animation of shapes in a photo taken of the object. A computer vision-based algorithm then applies the encoded animation to the photo to produce the resulting video.

4 Methodology

Overview

The methodology is split between the *creator* and the *viewer*. The *creator* designs the animation to be encoded and applies the QR code to the target object. The *viewer* photographs the object and watches the object come to life as defined by the encoded animation.

A use flow would thus look like:

1. The *creator* uses an animation tool to define polygon outlines and animation transitions applied to those outlines over certain periods of time.
2. The *creator* exports this as a visual code to be printed onto the designated object.
3. The *viewer* takes a photo of the visual code along with the object.
4. An image processing tool uses the visual code to:
    a. Match the polygon outlines to the shapes in the image.
    b. Apply the animation transitions to the matched parts of the image.
    c. Fill in any holes in the image.
    d. Render the animation for the *viewer* to view as a video.

Animation Creation Tool

The animation tool allows *creators* to build animations of simple polygons consisting of basic transformations like translation and rotation. *Creators* build the polygons graphically. They define keyframes with set values for each transform. The animation thus consists of the interpolation of the transformations for frames between adjacent keyframes.

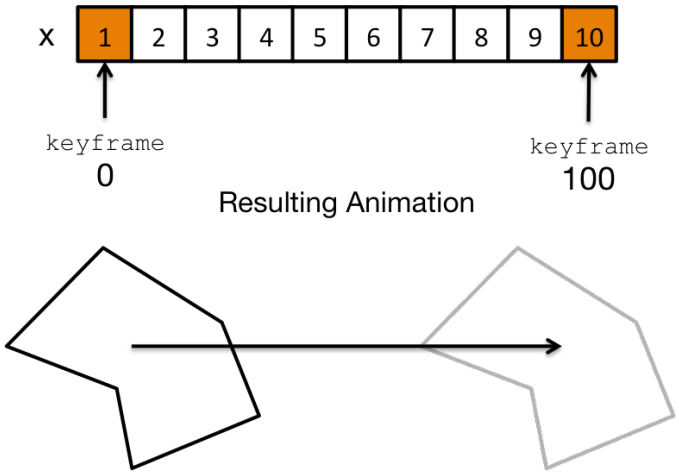


Figure I. For example, a keyframe for an x position of 0 at frame 1 and 100 at frame 10 animates the polygon in a line from left to right in 10 frames.

The animation tool outputs the QR code that encodes the built animation. A main goal is to encode the animation with as few bits as possible. This allows for more complicated and higher resolution animations to be encoded into the limited capacity of QR codes. Polygons are encoded with their vertices. Animations are encoded with values at set keyframes.

The bit-encoding scheme is as follows:

1. Repeat for each polygon:
    a. First 4 bits encode the number of vertices. This can define up to 16 types of polygons.
    b. Repeat for each vertex:
        i. First 8 bits encode the x coordinate.
        ii. Last 8 bits encode the y coordinate.
        iii. This can define coordinates on a 256 x 256 grid with a 1 cell resolution.
    c. Next 6 bits encode the number of translation keyframes. This supports up to 64 frames, which can be viewed at any desired speed.
    d. Repeat for each translation keyframe:
        i. First 6 bits encode the frame index of the keyframe.
        ii. Next 8 bits encode the x position at that keyframe.
        iii. Next 8 bits encode the y position at that keyframe.
    e. Next 6 bits encode the number of rotation keyframes.
    f. Repeat for each rotation keyframe:
        i. First 6 bits encode the frame index.

ii. Next 9 bits encode the rotational angle. This encodes an angle in the range $[-2\pi, 2\pi)$ to the integers 0 - 511.
iii. This can define rotations within a 4π range with a resolution of $\frac{\pi}{128}$ radians.

With this encoding scheme, a basic animation of a polygon rotating and translating to a new position would consume 138 bits.

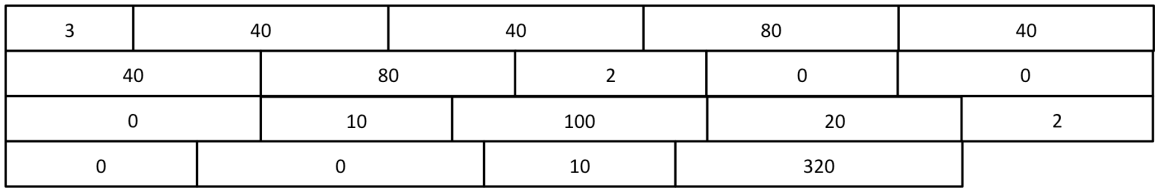


Figure II. Bit representation of a basic animation of a polygon rotating and translating to a new position. The size of the box represents the bit-count. This animation takes a total of 138 bits.

In *Figure II*, the bits represent:

3 – the first (and only) polygon has 3 vertices
40 – the x coordinate of the first vertex
40 – the y coordinate of the first vertex
80 – the x coordinate of the second vertex
40 – the y coordinate of the second vertex
40 – the x coordinate of the third vertex
80 – the y coordinate of the third vertex
2 – the number of translation keyframes
0 – the frame index of the first translation keyframe
0 – the x position of the first translation keyframe
0 – the y position of the first translation keyframe
9 – the frame index of the second translation keyframe
100 – the x position of the second translation keyframe
20 – the y position of the second translation keyframe
2 – the number of rotation keyframes
0 – the frame index of the first rotation keyframe
0 – the rotational angle of the first rotation keyframe
9 – the frame index of the second rotation keyframe
320 – the rotational angle of the second rotation keyframe (π or 90°)

Figure III. The QR code would be similar to this.

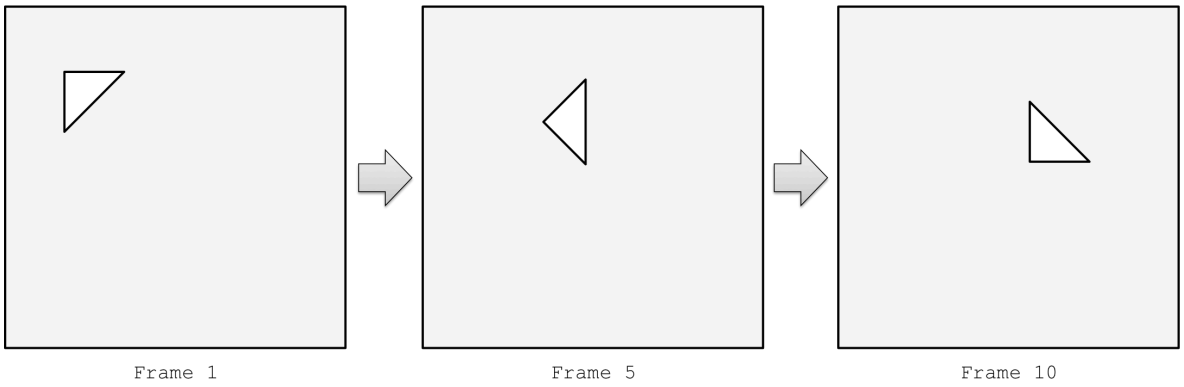


Figure IV. The resulting animation.

Matcher

On the *viewer* side, the viewer would hold a camera up to the object with the QR code. The QR code will be first scanned to provide an overlay of the encoded polygons so that the user can match the polygons to the object in the camera view as close as possible and take a photo. Since this manual matching is imperfect, the matching part of the algorithm uses computer vision techniques to correct any imperfections and extract the correct parts of the object.

Figure V. For example, in this stapler example, we want the polygon to match the top part of the stapler and nothing else in the image.

The matching algorithm takes the image as input and has 6 stages:

1. Decode the QR code in the image into the defined polygons.
2. Find lines in the image.
    a. Filter with edge detection.
    b. Apply Hough transform for line detection.
3. Match detected lines to the defined polygonal edges.
4. Extend the matched lines to obtain the matched polygons.
5. Extract the shapes by masking out the matched polygons.
6. Inpaint the background.

The algorithm outputs the background along with the parts of the image that were matched and extracted.

Decoding

The QR code in an image can be decoded into its contents using a bar code reading library such as ZBar [8]. The contents are then decoded into the polygons and animations using the bit representation scheme defined in the encoding section.

Find Lines – Edge Detection

First, a mask is created using the defined polygons. This mask is dilated to include a large swath of surrounding pixels. The original image is converted to grayscale and masked using this initial mask. Then, a Canny edge detection [3] filter is applied to obtain a black-white image.

The tuning parameters that can be adjusted in this stage are:

- The extent of dilation
- The threshold for the Canny edge detection
- The type of edge detection filter to use

Find Lines – Hough Transformation

Next, a Hough transform is applied on the black-white edge detection image. A high-pass filter is applied to obtain the most significant peaks. Using these peaks, lines segments are extracted by applying a Hough line-segment extraction algorithm [4].

The tuning parameters that can be adjusted in this stage are:

- The high-pass threshold for the Hough transform
- The FillGap and MinLength for Hough line-segment extraction

Match Hough Lines to Polygon Edges

This step takes each edge of the defined polygons and calculates a difference score for each of the extracted Hough line segments. The line segment with the lowest difference score is used as the match for that edge.

The difference score is calculated with a two-threshold method. First, the difference score is calculated as a weighted sum of the difference in endpoints and difference in angle:

$$score_1 = w_{endpoints}\sqrt{diff_{endpoints}} + w_{angle}diff_{angle}$$

where $w_{endpoints}$ and w_{angle} are the weights and $diff_{endpoints}$ and $diff_{angle}$ are the differences for the endpoints and angles, respectively.

The difference for the endpoints are calculated by:

$$diff_{endpoints} = \min\left(\left|a_{p1} - b_{p1}\right|\left|a_{p2} - b_{p2}\right|, \left|a_{p1} - b_{p2}\right|\left|a_{p2} - b_{p1}\right|\right)$$

where a and b are the two line segments, and $p1$ and $p2$ are the two endpoints in $\begin{bmatrix} x \\ y \end{bmatrix}$ form.

Since the two endpoints can be switched, the minimum is taken between the differences using either orientation. The two Euclidean distances are multiplied rather than added to prefer smaller discrepancies in differences between the two endpoints. Thus, in the $score_1$ calculation, the square root of $diff_{endpoints}$ is taken before weighting.

The difference for the angle is calculated by:

$$angle(\alpha) = arctan2\left(\alpha_{p2_y} - \alpha_{p1_y}, \alpha_{p2_x} - \alpha_{p1_x}\right)$$
$$diff'_{angle} = \big(angle(a) - angle(b)\big) \bmod \pi$$
$$diff_{angle} = \min\big(diff'_{angle}, \pi - diff'_{angle}\big)$$

where $arctan2$ performs $arctan$ but accounts for when the difference in x is 0.

The modulo and minimum is taken to obtain the smallest angle between the line segments.

Since some Hough lines will be much shorter than the polygonal edge they match to (and thus have a high $diff_{endpoints}$), a failsafe second score is calculated that doesn't require exact endpoint matches.

Therefore, if $score_1$ exceeds a certain threshold, $score_2$ is used instead.

$$score_2 = w_\rho diff_\rho + w_\theta diff_\theta$$

where $diff_\rho$ and $diff_\theta$ are the differences in ρ and θ represent the line segments extended to infinity in the Hesse normal form $\rho = x\cos\theta + y\sin\theta$.

In the worst-case scenario, if none of the Hough line segments produce a score below a low-pass threshold, the original polygonal edge is used as a last resort.

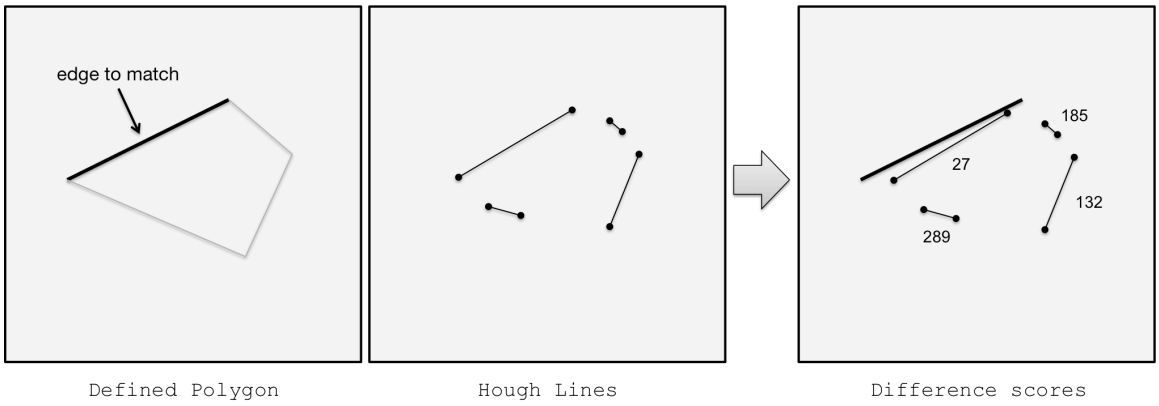


Figure VI. Example matching using difference score calculation. The line segment with score 27 is chosen as the match.

The tuning parameters that can be adjusted in this stage are:

- The weights $w_{endpoints}$, w_{angle}, w_{ρ}, and w_{θ}
- The threshold between $score_1$ and $score_2$
- The last resort threshold

Extend matched lines

Once the matched lines for the polygonal edges are obtained, these line segments are each extended to infinity. The algorithm then calculates the intersection points of the resulting lines to obtain the final vertices of the matched polygon in the image.

Extract regions

Next, the algorithm creates masks using the matched polygons to mask out regions of the original image that correspond to each polygon. These are the parts that are to be animated. Of course, the masks are blurred slightly using a Gaussian filter to create smoother edges for the extracted parts.

The tuning parameters that can be adjusted in this stage are:

- The amount of smoothing

Background Inpainting

Finally, the algorithm combines all the masks of the matched polygons and applies the combined mask inversely to the original image to extract the background. The mask is eroded slightly to erase any remaining edge artifacts from imperfect extraction of the polygonal parts. The holes left in the background are then inpainted using an inpainting algorithm such as basic interpolation via computing the discrete Laplacian over regions and solving the Dirichlet boundary value problem [5] or other methods such as Navier-Stokes or Telea [6].

The tuning parameters that can be adjusted in this stage are:

- The amount of erosion
- The method of inpainting

Video rendering

Once the photo passes through the matching algorithm, the extracted parts and the inpainted background are obtained. The final rendering places the background on a canvas along with the extracted parts in their initial position in frame 1. The video plays at a desired frames-per-second (60 fps for standard monitor refresh, unlimited for however fast the processor can handle the playback, or 23.976 fps for a filmic look). The renderer uses the animation data from the decoded QR code and interpolates the transformation data between keyframes. A desired playback speed can be used by designating the time interval between each animation frame (as opposed to playback frame). For example, since there are 64 frames total, if we wish to have the video last 6.4 seconds, the time interval between each animation frame would be 100 ms. Playback here at 60 fps would yield 384 playback frames.

5 Deliverables and How to Use

Animator

Overview

The browser-based animation tool supports the creation of polygonal animations with translational and rotational transformations.

Instructions

Run on a *Google Chrome* browser: */animator*

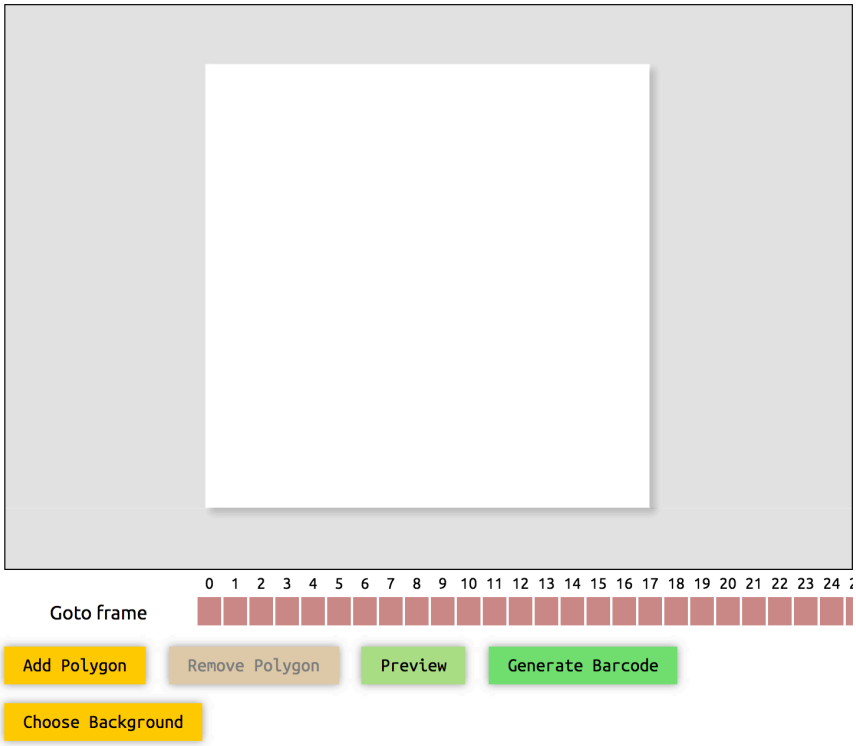


Figure VII. The initial screen for the animator

The *creator* is given initial controls to add a polygon (*Add Polygon*), navigate the 64 frames, and choose a reference image to work with (*Choose Background*). The canvas displays the stage in the middle where the animation can be previewed and manipulated.

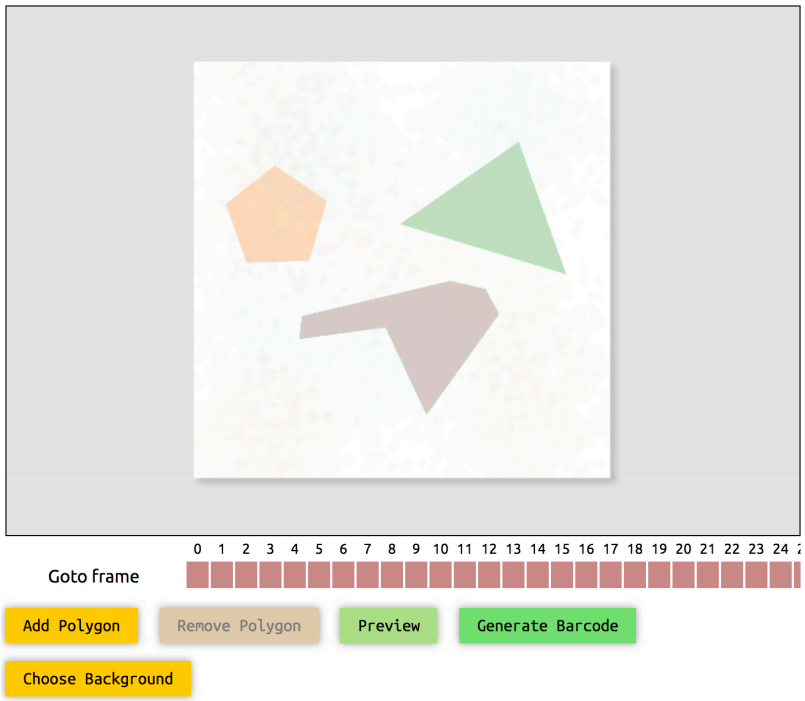


Figure VIII. A reference image is chosen to work with

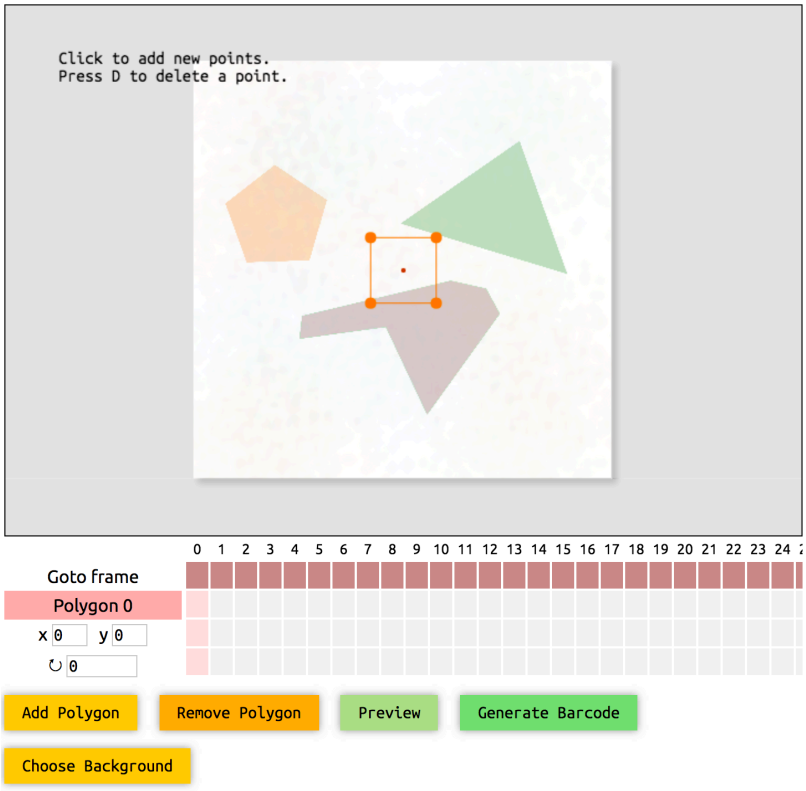


Figure IX. A polygon is added to the animation

When a polygon is added, three rows appear below the frame navigation strip (*Goto frame*). Clicking on each row gives different controls for the corresponding polygon on the stage.

Clicking on the polygon name (*Polygon 0*) gives knobs to drag the vertices of the polygon to define its shape. Clicking on the stage during this state adds new vertices to the polygon. Pressing *D* while hovering over a vertex knob deletes that vertex. A dot in the middle of the polygon represents the center of mass, which acts as the origin point for any rotational transformation. In this state, clicking *Remove Polygon* removes the polygon.

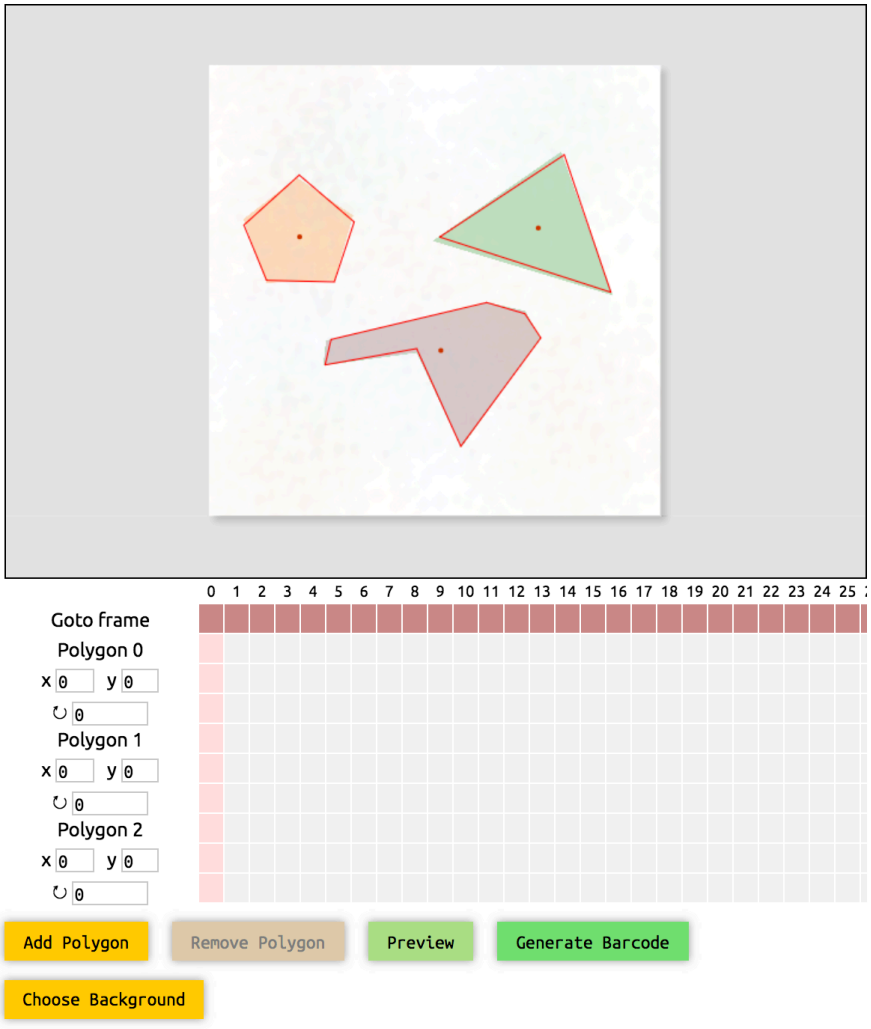


Figure X. Three polygons are defined to match the three shapes in the reference image.

Clicking the translation row (*x [0] y [0]*) puts the stage in translation state. Dragging during this state changes the position of the polygon for the current frame. A new keyframe is added for the current frame if there is none; otherwise, the existing keyframe is changed. The translational values (x and y) can also be changed by typing new values into the textboxes.

Similarly, clicking the rotation row (↻ *[0]*) puts the stage in rotation state. Dragging during this state rotates the polygon and manipulates the keyframe for the current frame. The rotational angle can also be changed by typing new values (in °) into the textbox.

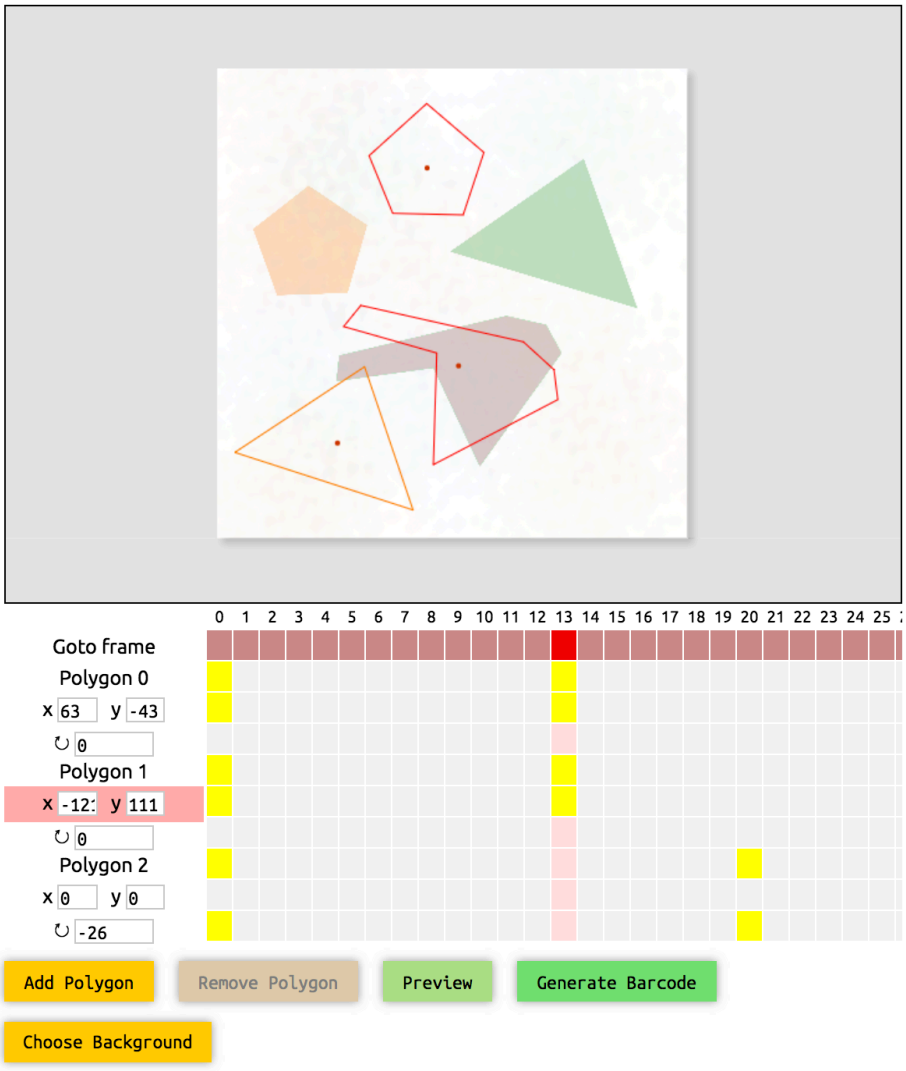


Figure XI. Multiple keyframes are added for multiple transformations for multiple polygons.

In the *Figure XI*, multiple keyframes are added to define an animation. The current frame is 13. The stage displays the view of the animation at the current frame.

Clicking and dragging along the frame navigator strip (*Goto frame*) scrolls through the frames dragged over. The animation can also be previewed by clicking *Preview*. Keyframes can also be added by clicking any of the cells in the keyframe table. Clicking a keyframe in the polygon definition row (*Polygon #*) adds keyframes for both translational and rotational transformations. Double-clicking any keyframe removes that keyframe.

Clicking *Generate Barcode* encodes the animation into a QR code and renders it to the webpage (for the *creator* to save or copy).

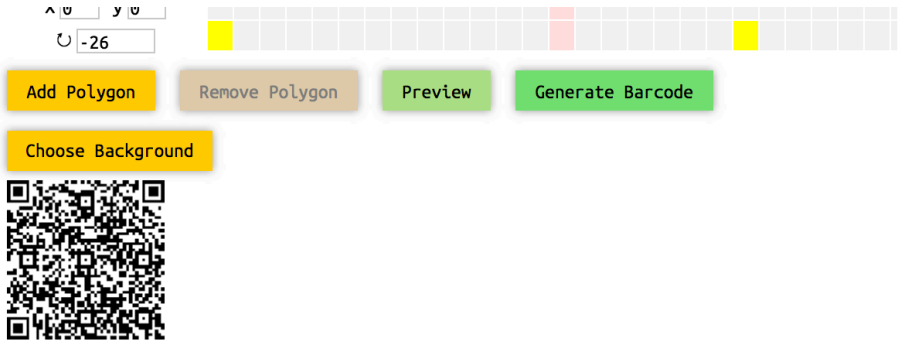


Figure XII. The generated QR code.

Technical Details

The browser application is written in *JavaScript,* in an object-oriented manner. It works mainly on the *Google Chrome* browser, which supports the necessary latest *ECMAScript* functionality. The programming paradigm is event-based. Controller-to-model communication is done with direct function calls. All other communication is done with events. This helps keep the relational set as small as possible.

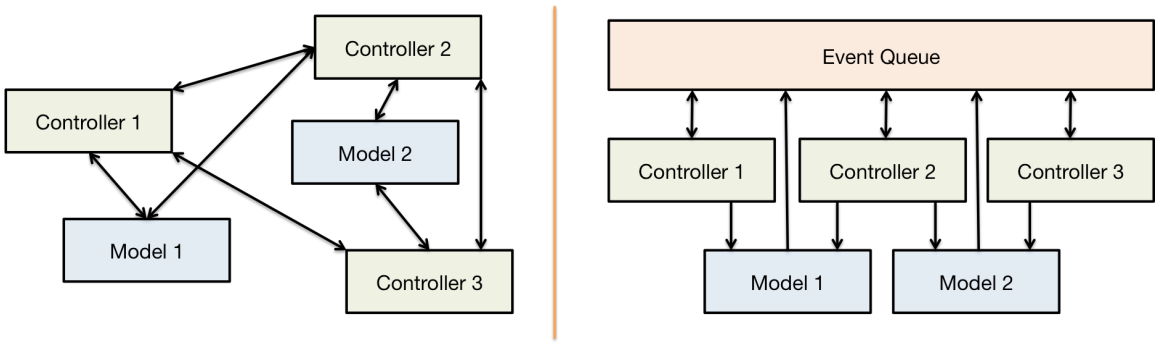


Figure XIII. Non-event-based model-controller paradigm (left) vs. event-based model-controller paradigm (right). The event-based paradigm is more organized and easier to reason about.

The models organize the underlying data for the animations, such as keyframe data for the transformations and polygon data for the vertex definitions of the polygons. The controllers access and manipulate data contained in the models (via interface functions) to provide the user interface controls and manage the graphical displays.

Events are defined under an event identifier. Objects that listen to an event (event owners) attaches a callback function and a priority to the event identifier in the event queue. Objects that fire an event dispatches an event with the corresponding identifier and data to the event queue. When an event is fired, the callback functions attached to the event are called in order of decreasing priority for each event owner.

Controllers listen to and fire events that can be responded to by other controllers. Controllers manipulate data contained in the models. Changes in the model fire events that other controllers can listen to to respond to the changed data.

See comments in the code for more detailed technical descriptions.

Viewer-Side

In this prototype, the following stages are written in separate programs. In a production version, these stages would be combined into one program that guides the *viewer* to take a photo of the object and outputs the rendered video for the *viewer* to watch.

Aligner

After scanning the QR code, the aligner displays the polygonal outline on a camera view to guide the viewer to manually align to the objects in the scene. After alignment, the photo taken will be passed into the next stage. The aligner was not developed as part of this prototype.

Decoder

Overview

In this prototype, the decoder takes an input image (taken by the *viewer*) that contains the QR code and decodes the QR code. It outputs two comma-separated values (CSV) files containing the numerical representation for the polygon definitions and the animation transformations. This makes it easier for the next stage to work with the numerical values in the CSV files rather than the encoded bits.

Instructions

Run: /decoder/decode.py *<name>*

This script takes an image file named *qr<name>.png* and outputs the CSV files *poly<name>.csv* and *anim<name>.csv* containing the polygon definitions and the animation transformations, respectively.

Technical Details

The decoder is written in *Python* and uses the *qrtools* library [7] to decode the QR codes. This library uses the *ZBar* bar code reader library [8].

Matcher

Overview

In this prototype, the matcher takes the input image (taken by the *viewer*) and the polygon definition file from the *Decoder* to match the defined polygons to the shapes

in the input image. The matcher outputs images for the extracted parts and the inpainted background.

Instructions

Run with MatLab: /matcher.m

Make sure to set *IMAGE_FILE* and *POLYGON_FILE* to the input image file and the polygon definition file, respectively.

Technical Details

The MatLab script uses various image processing functions to perform the matching algorithm. *hough* performs the Hough transform, *houghpeaks* filters the peaks of the Hough transform, and *houghlines* extracts the line segments from the Hough transform.

Watcher

Overview

In this prototype, the watcher takes the polygon definitions and animation transformations CSV files from the *Decoder* as well as the background and extracted parts image files from the *Matcher* to display the rendered animation to the *viewer*.

Instructions

Run on a *Google Chrome* browser: /watcher/

Figure XIV. An example animation being played on the Watcher.

Once the input files are chosen, clicking *Play Animation* will playback the animation at 60 fps with an animation frame interval of 0.2.

6 Examples

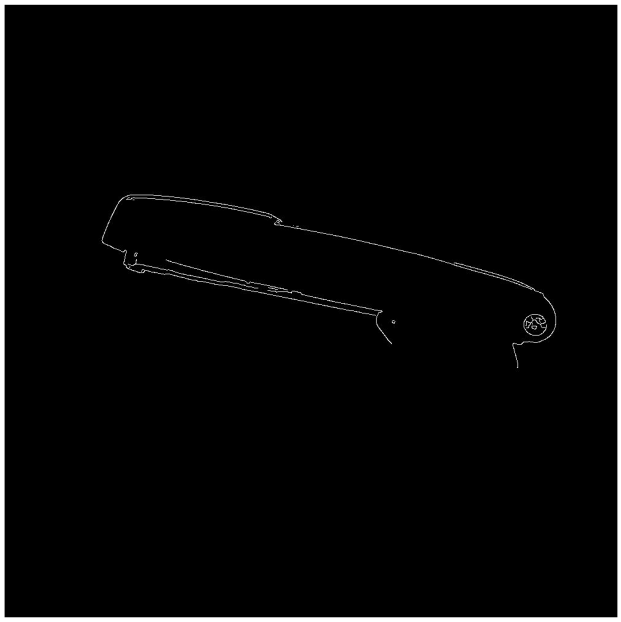
Stapler

A polygon matching the top part of the stapler is animated to rotate down towards the base of the stapler.

Original

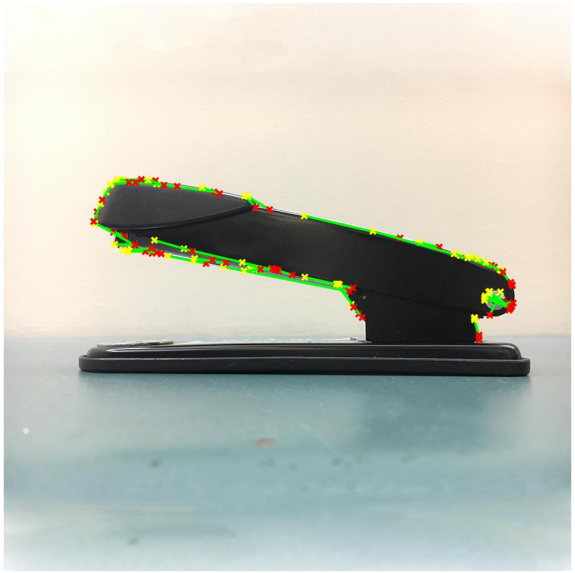
Defined Polygon

Edge Detection



Since the stapler has sharp edges, the edge detection perfectly traced the outline of the stapler.

Hough Lines



Matched Lines

The smaller edges of the polygon near the front-bottom part and back part (where the screw is) did not match as well since there were many lines that could similarly have matched those edges.

Extended Matched Lines

Masked Out Object

Inpainted Background

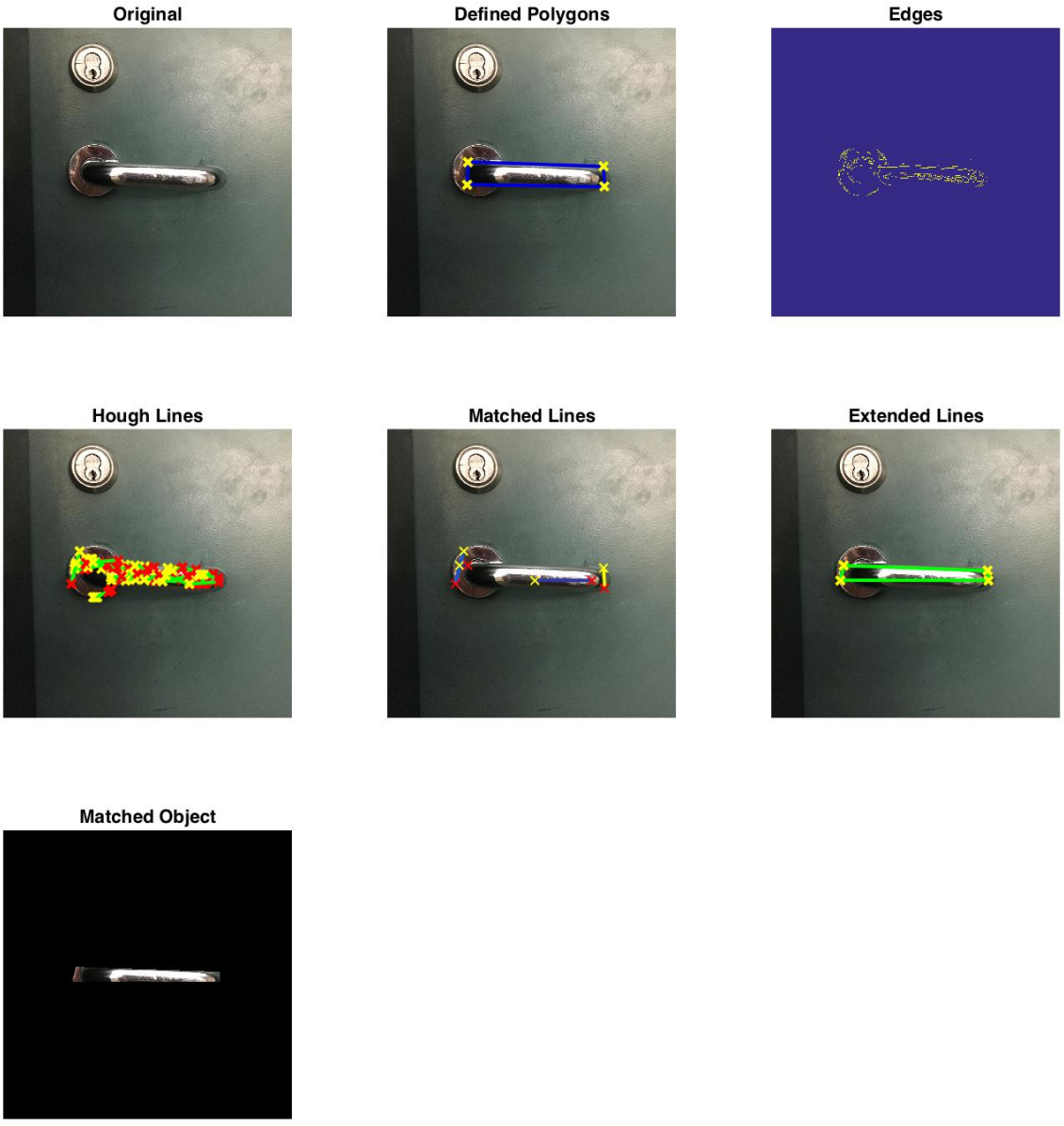
Resulting Animation

Even with the minor inaccuracies, the resulting animation does not have any significant mistakes.

Door Handle

A polygon matching the door is animated to rotate down as if the door handle was being turned.

Matcher Output



The edge detection mostly detected the bright part of the door handle and did not catch the true border of the door handle. However, the overall shape of the handle was matched well.

Resulting Animation

Even though only the top bright part of the door handle was matched well, the animation mostly worked, with the minor remnants of the bottom of the door handle still left in the inpainted background portions.

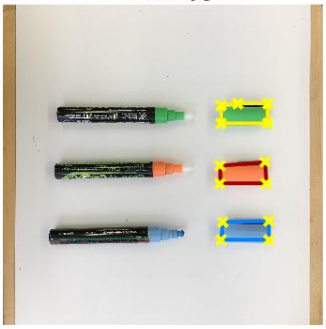
Markers

Three polygons matching the marker caps are animated to cap the markers.

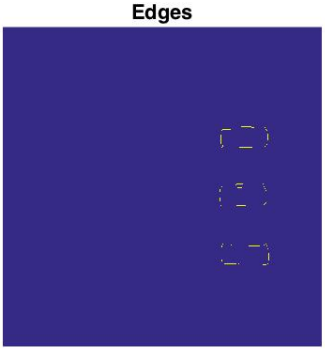
Matcher Output

Original

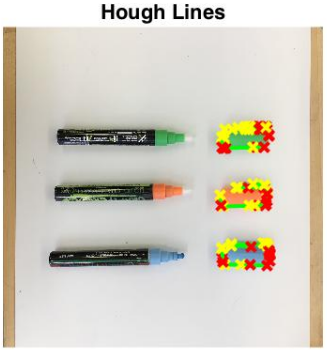
Defined Polygons



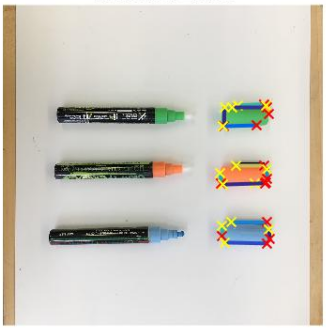
Edges



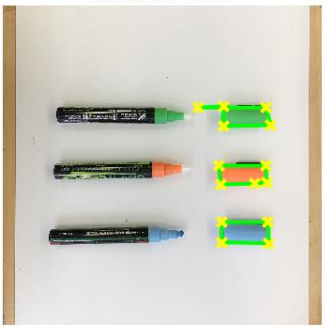
Hough Lines



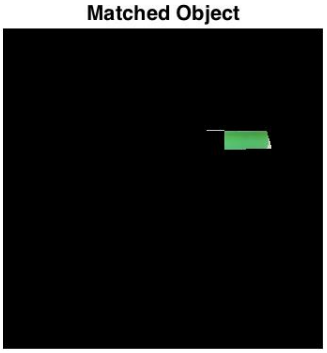
Matched Lines



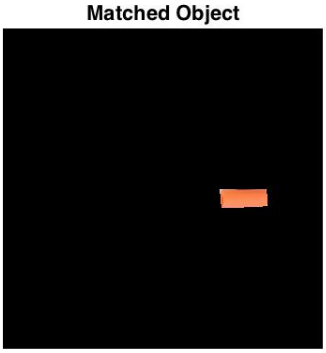
Extended Lines



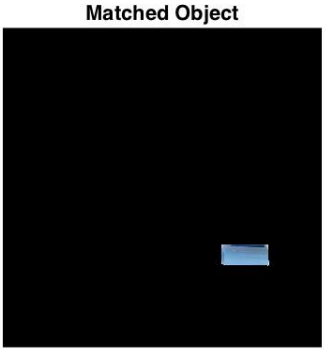
Matched Object



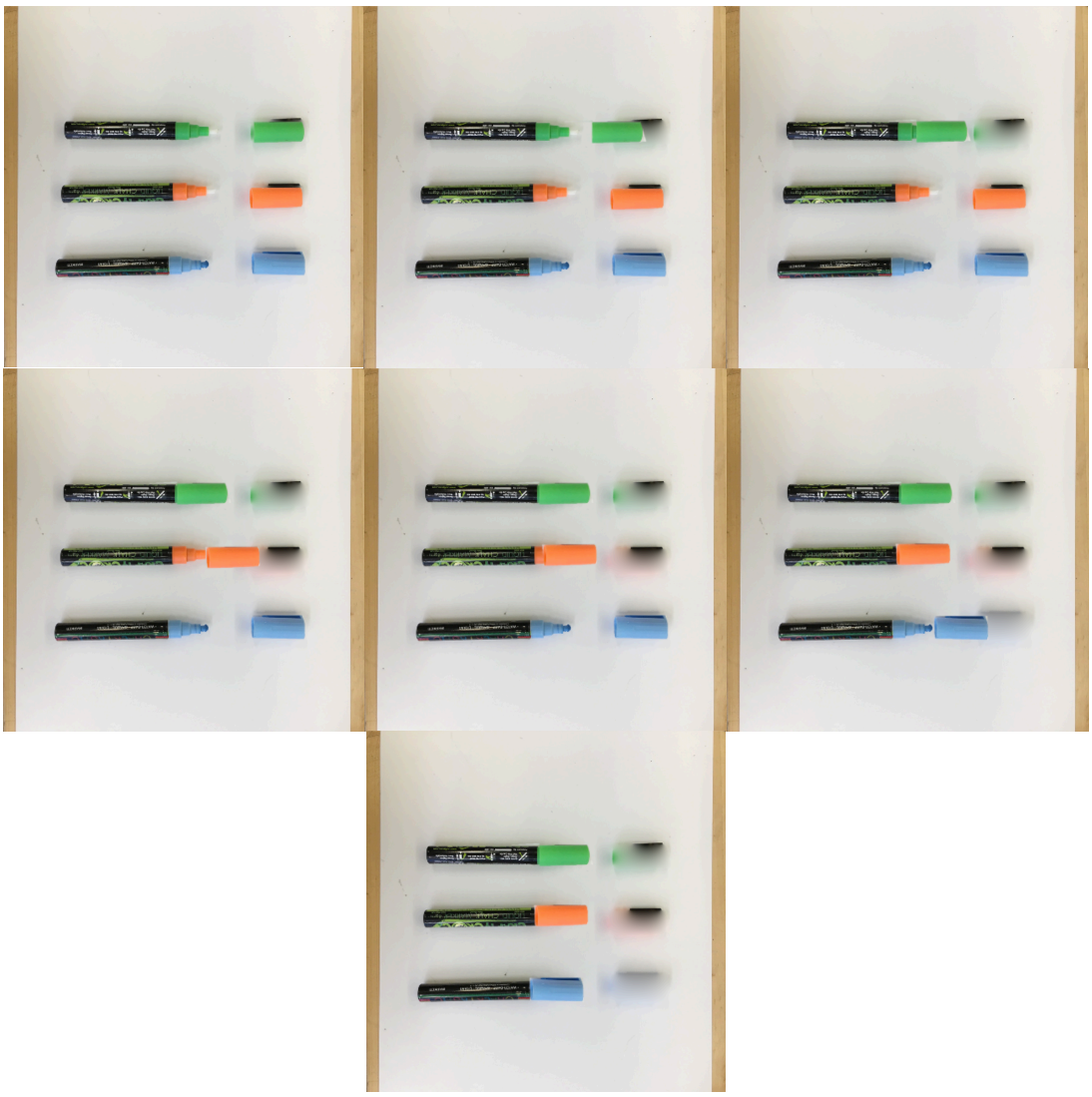
Matched Object



Matched Object



Resulting Animation



The animation worked well but there seems to still be noticeable remnants of the caps in the inpainted background, as the matched regions did not capture the black clip of the cap and minor edge artifacts.

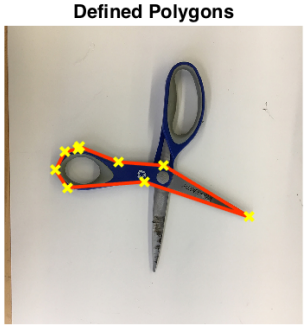
Scissors

A polygon matching the top arm of the scissor is rotated to close the scissors.

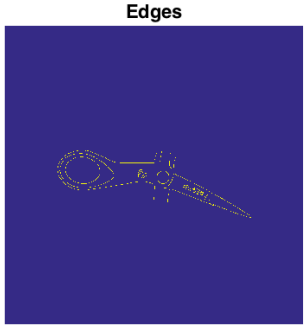
Matcher Output

Original

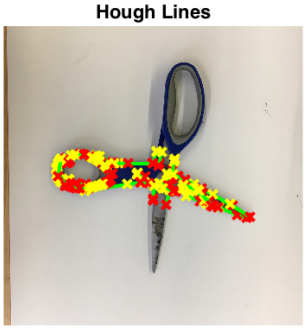
Defined Polygons



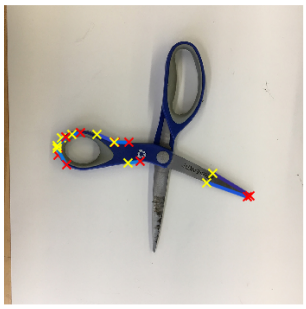
Edges



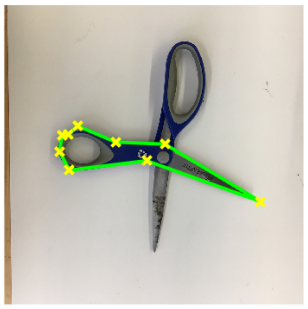
Hough Lines



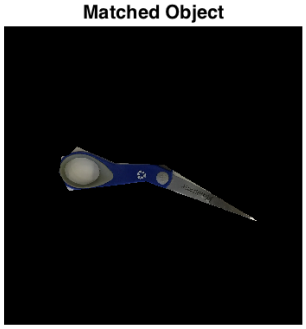
Matched Lines



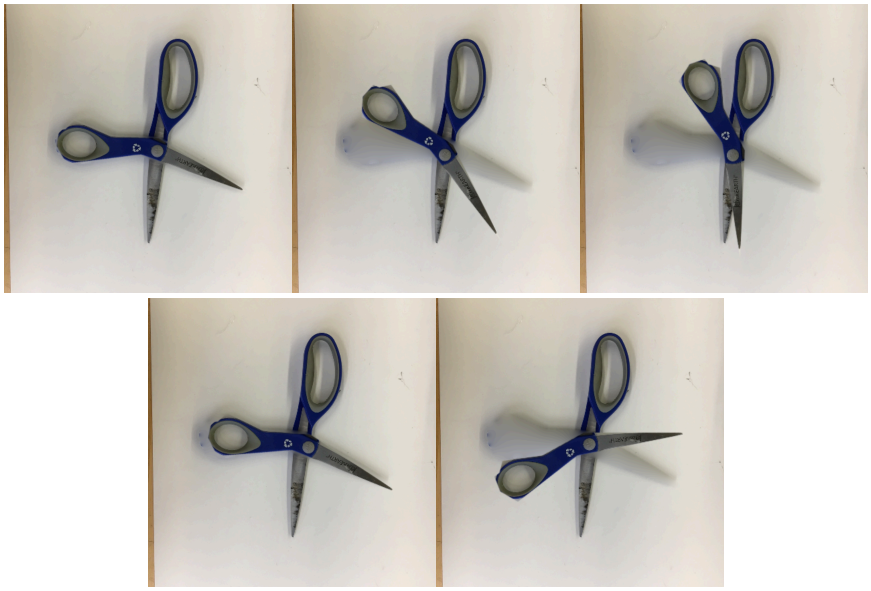
Extended Lines



Matched Object



Resulting Animation



6 Examples – Printed

Some tests involved actually printing out a poster with shapes to be animated. The QR code was also printed on the poster. A photograph was taken and used as the input to the system.

Peace Poster

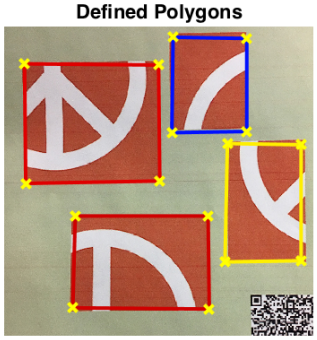
Four polygons matching parts of a peace sign are animated to align up and form a complete peace sign.

Original Photo

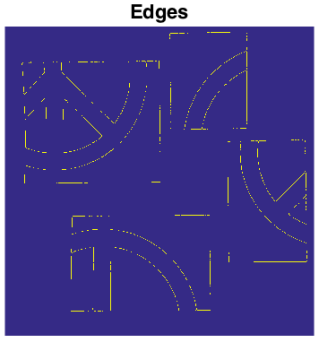
Matcher Output

Original

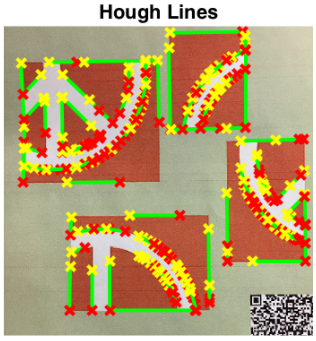
Defined Polygons



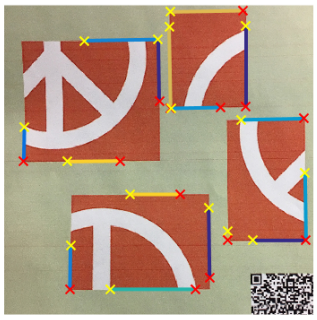
Edges



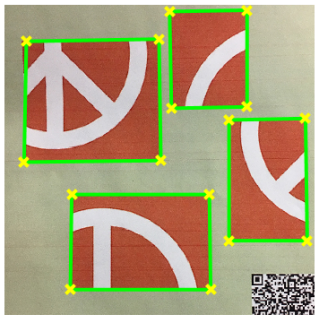
Hough Lines



Matched Lines

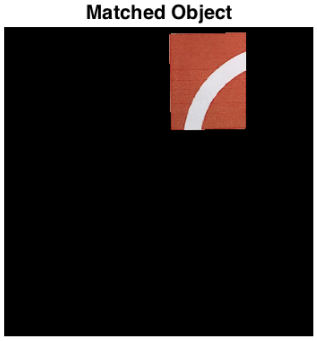


Extended Lines

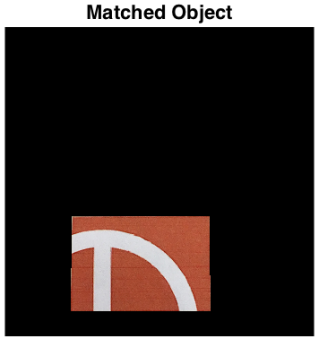


Matched Object

Matched Object



Matched Object



Resulting Animation

There is a small remnant of the top-left corner, but the overall shape of the peace sign was achieved with minor misalignments.

Lux Et Veritas Pyramid

Three polygons matching parts of a pyramid are animated to line up and spell out “Lux Et Veritas”.

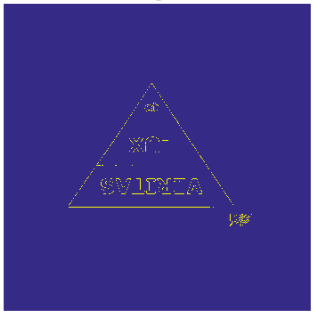
Original Photo

Matcher Output

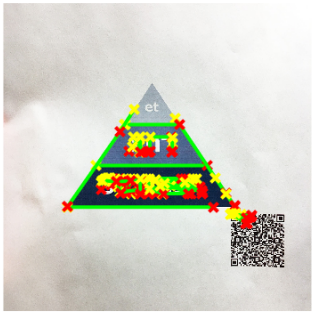
Original

Defined Polygons

Edges

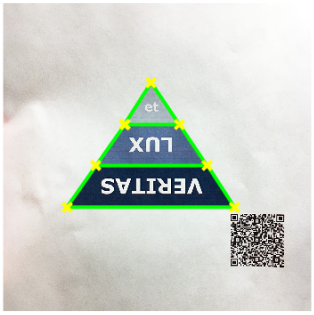


Hough Lines

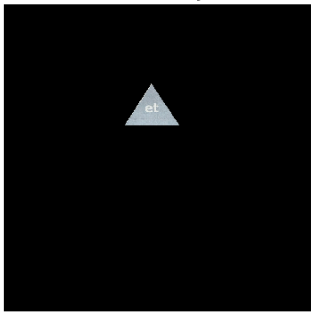


Matched Lines

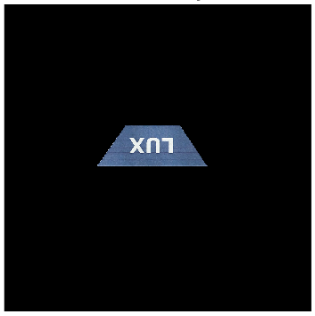
Extended Lines



Matched Object



Matched Object



Matched Object

Resulting Animation

The alignment was shifted slightly to account for the displacement of the polygons.

The background inpainting worked perfectly here as the background looked just like the white paper the graphics were printed on. The shapes matched well and animated to their correct positions.

7 Future Work

The system described can perform some basic polygon animation to a satisfactory extent for a prototype. However, much future work could be done to improve and extend the system.

The animation tool can be improved by providing more robust features and allow for more dynamic animations. For example, scale can be added as another transformation. A direction can also be associated with rotation so that rotations that wrap around between positive and negative go in the desired direction. Also, another neat feature would be to allow for the animation of non-polygons. Instead of just being able to define polygons and animate them, the creator can also be allowed to define circles, curves, and other non-shapes. The non-shapes can be other graphics that can be overlaid onto the scene and animated. For example, a graphic that represents a hand can be animated to “push” a button in the scene. These

graphics would not require any extra space in the QR code, as the same bits used to represent the number of polygonal vertices can include the graphics. For example, 4 bits used to represent polygonal vertices can support up to 10 vertices or 6 different graphics in the 16 total values.

Higher resolution and more complex animations can be achieved by using higher capacity bar codes such as the High Capacity Color Barcodes [9].

On the viewer side, for production, a mobile application can be developed that incorporates all of the steps of the viewer-side algorithm. The viewer would hold a camera view up to the object to be photographed, including the QR code in the view. The app would scan the QR code and display the polygon outlines on the camera view for the viewer to align with the object. Once aligned, the app would take a photo and perform the algorithm to generate the resulting animation. This resulting animation would be played back to the viewer.

The matching algorithm itself also has room for improvement. Currently, it only works for polygons. The algorithm would be need to be changed to support curves and other shapes that are not easily match-able to Hough lines. Also, segmenting the objects beforehand could help with matching the exact edges of the objects rather than sometimes cutting out pieces of the background or the object itself.

8 Conclusion

The prototype for turning real-life still images into video animations worked sufficiently well for the examples that were tested. The major challenge was to design a way to encode a video into the small capacity afforded by QR codes. The main insight that solved this challenge was that much of the information needed to encode the video did not need to be encoded in the QR code. Rather, much of that information can be captured through a photo, and the QR code simply needed to encode the *changes* to that photo. Then, all the work is done by large amounts of code that can interpret these changes and apply them to the photo. This concept can be applied to other problems that may require the storage of large amounts of data into small capacities.

The technology of animating objects in real life also has exciting applications in the future of augmented reality. Currently, augmented reality headsets such as

the Microsoft Hololens [10] can only *project* holograms onto a real-life environment. With further development of this technology, in the future, augmented reality headsets could *transform* and *manipulate* the real-life environment as well.

9 References

[1] "Information Capacity and Versions of the QR Code." *Information Capacity and Versions of QR Code*. N.p., n.d. Web. 02 May 2017. <http://www.qrcode.com/en/about/version.html>.

[2] "Bit Rate." *Wikipedia*. Wikimedia Foundation, 25 Apr. 2017. Web. 02 May 2017. <https://en.wikipedia.org/wiki/Bit_rate#Video>.

[3] "Canny Edge Detector." *Wikipedia*. Wikimedia Foundation, 25 Apr. 2017. Web. 02 May 2017. <https://en.wikipedia.org/wiki/Canny_edge_detector>.

[4] "Hough." *Extract Line Segments Based on Hough Transform - MATLAB Houghlines*. N.p., n.d. Web. 02 May 2017. <https://www.mathworks.com/help/images/ref/houghlines.html>.

[5] "Imfill." *Fill in Specified Regions in Image Using Inward Interpolation - MATLAB Regionfill*. N.p., n.d. Web. 02 May 2017. <https://www.mathworks.com/help/images/ref/regionfill.html>.

[6] "Inpainting." *Inpainting — OpenCV 2.4.13.2 Documentation*. N.p., n.d. Web. 02 May 2017. <http://docs.opencv.org/2.4/modules/photo/doc/inpainting.html>.

[7] Primetang. "Primetang/qrtools." *GitHub*. N.p., 06 Feb. 2017. Web. 03 May 2017. <https://github.com/primetang/qrtools>.

[8] *ZBar Bar Code Reader*. N.p., n.d. Web. 03 May 2017. <http://zbar.sourceforge.net/>.

[9] "High Capacity Color Barcodes (HCCB)." *Microsoft Research*. N.p., n.d. Web. 03 May 2017. <https://www.microsoft.com/en-us/research/project/high-capacity-color-barcodes-hccb/>.

[10] Microsoft. "Detail of Light Relfecting on Lens of HoloLens." *Microsoft HoloLens*. N.p., n.d. Web. 03 May 2017. <https://www.microsoft.com/en-us/hololens>.