

IV. Implementation and Testing

This section contains important implementation and testing plan documents for the Silhouette project. It also contains a detailed section of algorithm implementation specifics.

IV.A Implementation Plan

The figure below, Figure IV.A.1, is a diagram used to better explain the expected implementation process of the Silhouette project. Blue indicates features scheduled to be complete for the Alpha version, yellow for beta, and green for final. This diagram is only a plan reflecting a schedule which may be modified at anytime due to unseen project roadblocks. It should in no way be considered absolutely final.

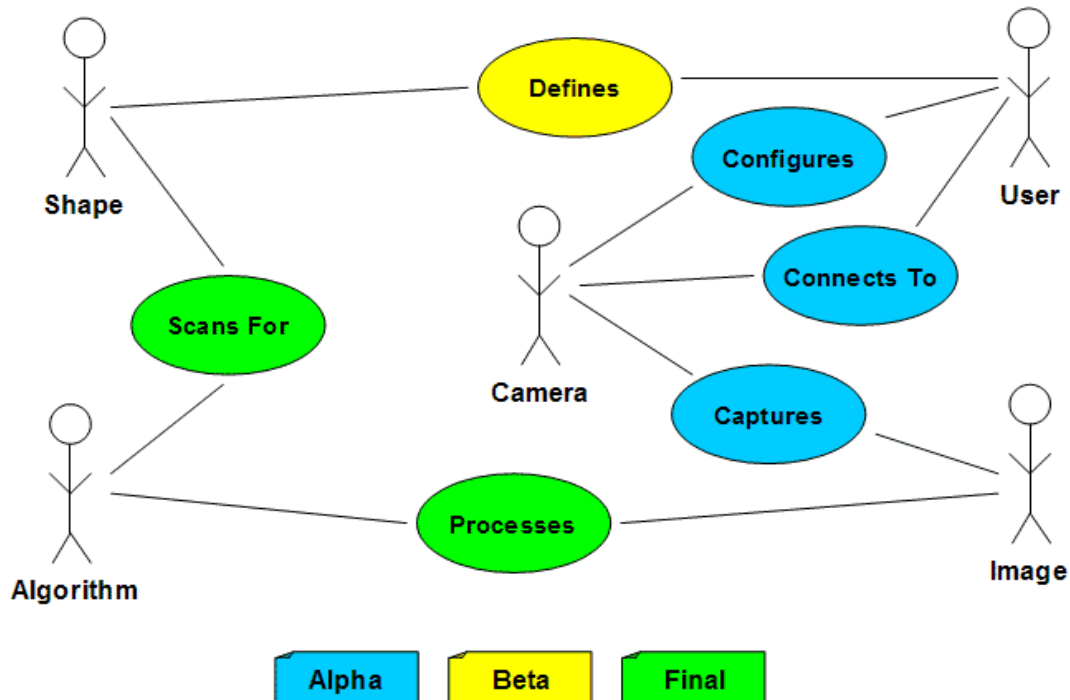


Figure IV.A.1. Implementation Plan Diagram

IV.B Implementation Plan Schedule

Table IV.B.1 shows the the project's Implementation Plan Schedule. Again, this is an approximate schedule which projects the expected time frame of component completion.

Table IV.B.1. Implementation Plan Schedule

Week Number	6-7	8-9	10-11	12	13	14-15	16
Phase Alpha Coding							
Phase Alpha Unit Testing							
Phase Alpha Integration							
Phase Alpha Integration Testing							
Phase Alpha User Testing							
Phase Beta Coding							
Phase Beta Unit Testing							
Phase Beta Integration							
Phase Beta Integration Testing							
Phase Beta User Testing							
Phase Final Coding							
Phase Final Unit Testing							
Phase Final Integration Testing							
Phase Final Integration							
Phase Final User Testing							

IV.C Test Plan

The following sections complete the Silhouette Project System Acceptance Test Plan. Each unique sub-section within this section of the SDN is designed to help the developer properly test and debug each unit/component of the application.

IV.C.1 System Acceptance Plan

The following table, Table IV.C.1.1, outlines the System Acceptance Test Plan for the Silhouette Project. The color-coded table legend below, indicates the milestone level for each project use-case. All use-cases and project requirements have been assigned a milestone level: alpha (blue), beta (yellow), or final (green).

ALPHA	BETA	FINAL
-------	------	-------

Table IV.C.1.1. System Acceptance Test Plan

<i>Use Case</i>	<i>Test #</i>	<i>Test Name</i>	<i>Requirement(s) Tested</i>	<i>Test Purpose</i>
1 - Define and Configure a Network Camera	1	Define and Configure Camera	#1	Ensures users can successfully add and configure a web/network camera to the application framework.
2 - Adjust shape comparison algorithm tolerance.	2	Adjust Tolerance	#2 and #6	Ensures users can adjust the comparison algorithm tolerance.
3 - Adjust real-time image stream refresh rate.	3	Adjust Refresh Rate	#3	Ensures users can successfully adjust the real-time image refresh rate.
4 - Select which shapes will be used to make comparisons.	4	Shape Selection	#4	Ensures users can successfully select which shapes to use in the real-time comparison engine.
5 - Start and stop real-time comparison engine.	5	Pause and Play Comparison Engine	#5 and #6	Ensures users can start and stop the real-time shape comparison engine.

CMSI 402 SENIOR PROJECT LAB
Implementation/Testing - Silhouette Project - Mark Kolich

<i>Use Case</i>	<i>Test #</i>	<i>Test Name</i>	<i>Requirement(s) Tested</i>	<i>Test Purpose</i>
6 - Apply shape detection to each captured image from camera.	6	Shape Detection	#6	Ensures shape detection is applied to each frame from the camera.
7 - Superimpose wire-frame shape on image when match is found.	7	Wire-Frame	#7	Ensures a wire-frame shape is placed over the shape on the camera stream.
8 - Pause the real-time camera stream.	8	Pausing and Playing Camera Stream	#8	Ensures users can pause the camera stream.

IV.C.2 Requirements Test Matrix

The following table, Table IV.C.2.1, shows the System Requirements Test Matrix which assigns a specific test number to each project requirement.

Table IV.C.2.1. System Requirements Test Matrix

<i>Requirement</i>	<i>Test Number</i>
1 - Define and Configure a Network Camera	1
2 - Adjust shape comparison algorithm tolerance.	2, 6
3 - Adjust real-time image stream refresh rate.	3
4 - Select which shapes will be used to make comparisons.	4, 6
5 - Start and stop real-time comparison engine.	5, 7
6 - Apply shape detection to each captured image from camera.	6
7 - Superimpose wire-frame shape on image when match is found.	5, 7
8 - Pause the real-time camera stream.	8

IV.C.3 Test Cases

The following table, Table IV.C.3.1, shows various test cases for each test number identified in Section IV.C.1 and Section IV.C.2 of this document.

Table IV.C.3.1. Test Cases Matrix

<i>Test #</i>	<i>Test Name</i>	<i>Expected Return Value</i>	<i>Test Case</i>	<i>Test Procedure</i>
1	Define and Configure Camera	If all fields are correctly completed, the camera should be added to the Camera Manager. If not all fields are complete, a dialog alert should notify the user that all fields are required.	Camera URL = " http://www.someurl.com/camera.jpg " Camera Name = "Marks Camera" Requires Authentication? = No	The application must have opened successfully. If necessary, open the Camera Manager window. Click on the add button to open the "Add Camera" dialog. Complete the fields, and click the "OK" button.
2	Adjust Tolerance	Algorithm correctly adjusts tolerance.	Move slider from Tolerance 30 to Tolerance 90, increasing the algorithm Tolerance.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. Adjust tolerance slider as necessary to modify the algorithm tolerance.

CMSI 402 SENIOR PROJECT LAB
Implementation/Testing - Silhouette Project - Mark Kolich

<i>Test #</i>	<i>Test Name</i>	<i>Expected Return Value</i>	<i>Test Case</i>	<i>Test Procedure</i>
3	Adjust Refresh Rate	Application adjusts camera stream refresh rate.	Move refresh rate slider from 2000ns to 3000ns.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. Adjust the refresh slider to adjust the applications refresh rate.
4	Shape Selection	Selected shapes are recognized and configured within the real-time shape recognition algorithm.	User selects a triangle, and a circle from the shape selection window.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. The users selects one or more shapes from within the shape selection window.
5	Pause and Play Comparison Engine	The real-time comparison engine is temporarily paused, or un-paused.	User selects the "real-time shape recognition" check box to enable shape recognition.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. The user has selected one or more shapes from the shape selection window. The user selects or un-selects the "real-time shape recognition" check box.

CMSI 402 SENIOR PROJECT LAB
Implementation/Testing - Silhouette Project - Mark Kolich

<i>Test #</i>	<i>Test Name</i>	<i>Expected Return Value</i>	<i>Test Case</i>	<i>Test Procedure</i>
6	Shape Detection	The algorithm successfully identifies shapes on the real-time camera stream and returns a series of points for each shape.	The user selects the "real-time shape recognition" check box to enable shape recognition. Application recognizes a square in the real-time camera stream.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. The user has selected one or more shapes from the shape selection window. The user selects the "real-time shape recognition" check box to enable shape recognition.
7	Wire-Frame	The algorithm successfully identified shape(s) on the real-time camera stream and superimposes a wire-frame outline over the recognized shape(s).	The algorithm scans the camera stream and identifies a square. A square wire-frame is superimposed over the shape on the real-time stream.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera. The user has selected one or more shapes from the shape selection window. The user selects the "real-time shape recognition" check box to enable shape recognition.
8	Pausing and Playing Camera Stream	The application successfully pauses or un-pauses the camera stream.	The user clicks the "pause button" to pause/play the stream.	The application must have opened successfully and the user is currently viewing a live camera stream from a configured network/web-camera.

IV.C.4 System Test Report - ALPHA

As of February 15, 2005 (Milestone #1), the ALPHA System Test Report is complete. All ALPHA based tests performed flawlessly as expected.

IV.C.4.1 Test Outcomes - ALPHA

Table IV.C.4.1.1. ALPHA Test Matrix

<i>Test #</i>	<i>Test Name</i>	<i>Outcome</i>	<i>Notes</i>
1	Define and Configure Camera	PASS	The application supports adding and configuring a web or network camera. Users can also modify camera information for cameras previously saved in the application framework.
3	Adjust Camera Refresh Rate	PASS	Once a camera has been configured and the user is successfully viewing the camera-stream, the user can successfully adjust the camera's refresh rate. The refresh rate determines how often the image from the camera will be refreshed and updated in the application.
8	Pausing and Playing Camera Stream	PASS	When a user is viewing a camera stream, they can now successfully pause and resume the camera feed as necessary.

IV.C.4.2 Related Test Problem Reports (TPR) - ALPHA

No problems with ALPHA testing have been identified at this time.

IV.C.5 System Test Report - BETA

As of March 29, 2005 (Milestone #2), the BETA System Test Report is complete. The results of all BETA tests are included in the BETA Test Matrix below.

IV.C.5.1 Test Outcomes - BETA

Table IV.C.5.1.1. BETA Test Matrix

<i>Test #</i>	<i>Test Name</i>	<i>Outcome</i>	<i>Notes</i>
4	Shape Selection	PASS	Users can successfully select shapes from the shape selection window. The users' selections are then passed to the shape comparison engine which scans for shapes.
6	Shape Detection	FAIL	At this time, shape detection is only able to detect circles using the Hough Algorithm. Additional shapes are to be added soon.

A TPR, Test Problem Report, has been filed for failed Test #6. The details of the TPR can be found in Section IV.C.5.2 of this document.

IV.C.5.2 Related Test Problem Reports (TPR) - BETA

At this time, Test #6 has failed due to limited shape recognition functionality. Additional shape recognition functionality will be added by the next milestone.

Table IV.C.5.2.1. TPR for System Test #6

<i>Test #</i>	<i>Test Name</i>	<i>Outcome</i>	<i>Notes</i>
6	Shape Detection	FAIL	At this time, shape detection is only able to detect circles using the Hough Algorithm. Additional shapes are to be added soon.
Suggested Action:			
I am currently in the process of implementing new algorithms to detect squares and other polygons. Additionally, triangles will also be available by the final milestone.			

IV.C.6 System Test Report - FINAL

As of April 26, 2005 (Final Delivery), the FINAL System Test Report is complete. The results of all FINAL tests are included in the FINAL Test Matrix below.

IV.C.6.1 Test Outcomes - FINAL

Table IV.C.6.1.1. FINAL Test Matrix

<i>Test #</i>	<i>Test Name</i>	<i>Outcome</i>	<i>Notes</i>
2	Adjust Tolerance	PASS	Users can successfully adjust the shape recognition algorithm tolerance.
5	Start and Stop Real Time Comparison Engine	PASS	Users can successfully start and stop the real-time comparison engine using the Silhouette user-interface.
7	Superimpose Wire-Frame	PASS	The application successfully superimposes a wire-frame shape onto the real-time camera stream when a shape is recognized.

IV.C.6.2 Related Test Problem Reports (TPR) - FINAL

No problems with FINAL testing have been identified at this time.

IV.D Description of Shape Recognition Algorithm

Any machine vision expert will tell you shape or pattern recognition is not a simple task. In fact, the algorithms and image processing involved is often extremely complex, computationally intensive, and requires multiple processing phases. The Silhouette project is no exception. My project's powerful application framework processes images using a complex four-step process:

1. **Camera Capture** - First, the framework must successfully authenticate to a specific camera and capture the video/image as a JPEG or GIF file. This captured image is processed and readied for the second phase, edge detection.
2. **Edge Detection** - After the image is captured from the camera, Silhouette uses a powerful edge detection algorithm to "strip out the meat of the image and leave the edges." The Sobel Edge Detection algorithm is a critical component of the shape recognition process. In a nutshell, edge detection is the process of defining edges in an image by scanning for differences in color intensity across the horizontal and vertical axes. The Sobel Algorithm used in the Silhouette Project was custom implemented for my application.
3. **Vector Extraction** - Once the image is successfully processed through the Sobel edge detection algorithm, the system must extract vectors, or lines, from the defined edges. In this case, a vector is defined as a line in space connecting two (x,y) points. For the Silhouette Project, I designed a custom vector extraction algorithm which follows extracted edges and produces a set of lines which can then be processed for shape recognition.
4. **Shape Detection** - Following the vector extraction process, the application now holds a stack, or list, of vectors. The next step, shape detection, uses the list of extracted vectors to compute possible shapes using pixel differences or

angles between the vectors.

IV.D.1 Sobel Edge Detection

Code Implementation: Section V.C

Package: edu.lmu.cs.kolich.shape

Source File: Sobel.java

One of the most eloquent and powerful edge detection algorithms is known as the Sobel Edge Detection algorithm. Because of its power and simplicity, it is used in a wide variety of image processing applications across the technology industry. In essence, “the Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image”¹. In other words, the algorithm picks a point and uses the color values of neighboring pixels to determine if an edge is present. The algorithm then combines these horizontal and vertical values to obtain an average. If an edge is present, then the algorithm sets the pixel value to white, otherwise it is set to black. Ultimately, this eliminates the “meat” of the image and only leaves the most noticeable edges as bold, white lines.

The Sobel Edge Detection algorithm is implemented as follows:

1. Assume we are looking at pixel-X ($p[i,j]$) in a binary image. To determine if pixel-X falls on an edge, we must also look at the neighboring pixels. For example, in Figure IV.D.1.1 below, pixels P_0 through P_8 are used to compute the horizontal and vertical intensities around pixel-X.

¹ Robert Fisher, Simon Perkins, Ashley Walker and Erik Wolfart. *Hyper Media Image Processing Reference*. 2003, Online Publication: <http://homepages.inf.ed.ac.uk/rbf/HIPR2/>

P₀	P₁	P₃
P₈	X	P₄
P₇	P₆	P₅

Figure IV.D.1.1. Sobel input matrix.

2. Next, we extract the values of P₀ through P₈ and compute the horizontal and vertical intensities, I_H and I_V, around pixel-X.

$$I_H = (P_3 + P_1 + P_0) - (P_5 + P_6 + P_7)$$

$$I_V = (P_3 + P_4 + P_5) - (P_0 + P_8 + P_7)$$

3. Finally, sum the horizontal and vertical intensities, and multiply by a brightness constant, c, if needed. The brightness constant can be used to increase or decrease the intensity of the final detected edge output.

$$X = [c * (I_H + I_V)]$$

In sum, the Sobel algorithm produces extremely vivid results and does not require much CPU overhead. As a result, it is a great algorithm for applications which require fast, yet efficient edge detection.

IV.D.2 *Vector Extraction, Vectorization*

Code Implementation: Section V.C

Package: edu.lmu.cs.kolich.algorithm

Source File: VectorExtractor.java

Vectorization is a difficult, yet critical component to the Silhouette shape recognition system. In essence, vectorization is the process of extracting vectors from a series of lines within a binary image. The extracted vectors are usually stored in a list, which can then be analyzed and manipulated based on the needs of the application, or system. The vector extraction algorithm used in the Silhouette Project is a basic, yet fairly effective vectorization tool which uses the slope of a line to trace and extract vectors in the image. For example, assume we have a series of pixels in a binary which clearly represents a line to the viewer. As shown below in Figure IV.D.2.1, a human can clearly distinguish lines from a group of pixels on a grid. However, it is much more complicated to “instruct” or “teach” a computer to extract vectors/lines from a series of pixels.

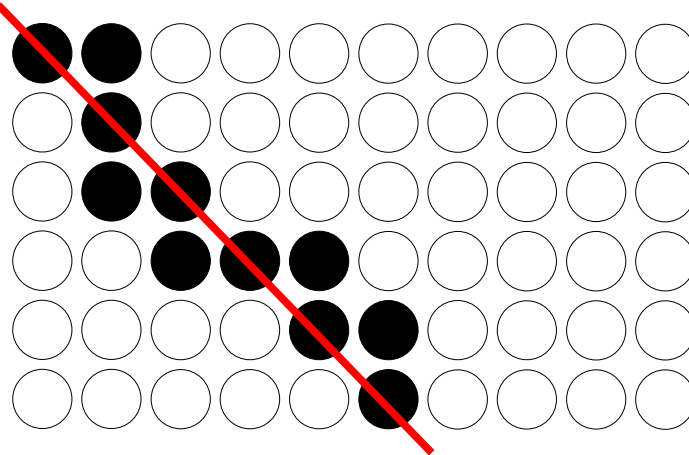


Figure IV.D.2.1. A series of pixels with an extracted vector.

As aforementioned, the Silhouette vector extraction algorithm is based on

computing the slope of a line as it traces a series of pixels. If the slope changes dramatically at any given point, then it is safe to assume the line ends, and we have extracted a vector. Computing the slope of a line is critical to the success of the algorithm due to the fact that we must gracefully handle various types of line intersections. The Silhouette vector extraction algorithm can be implemented as follows:

1. Receive and parse edge detected binary image into a 2-D image processing array. Most often, pixel values are stored as integers within an array.
2. Scan each row of pixels in the pixel array, left to right and top to bottom, looking for a starting point. For the sake of this example, assume our starting point is pixel (9,0) as shown in Figure IV.D.2.2 below.

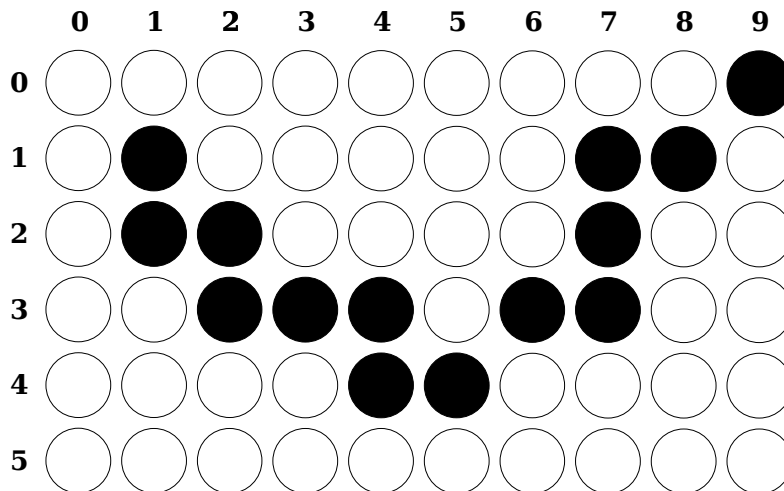


Figure IV.D.2.2. A set of sample pixels on a binary image.

3. Because pixel (9,0) contains a color value, it must be a good starting point. Next, we must find the “next good pixel” for which to visit. So, push the point (9,0) onto the top of “last good point” stack, set (9,0) to zero which removes any color, and look for the next good pixel. By setting the pixel value to zero,

our algorithm is able to remember that the pixel has already been visited. To look for the next good pixel, we simply check neighboring pixels for valid non-black (zero) color values. In this case, the pixel at (8,1) is the next pixel to visit.

4. At this point, we must now compute the slope between (9,0) and (8,1). If it is less than our desired tolerance range, then we can safely assume it is part of the same line. Repeat the process by pushing point (8,1) onto the "last good point" stack, and setting the value of (8,1) to zero to indicate we've visited the pixel. If backtracking is necessary, we can use the "last good point" stack to pop the last good point and continue the extraction process.
5. At this point, we can safely repeat steps 3 and 4 for the remaining pixels on the same slope. However, when the algorithm reaches (4,4), the next good pixel is found at (4,3). If we included the pixel (4,3) on our line, the average slope of the line we've extracted from pixel (9,0) to (4,4) will dramatically change. As a result, the algorithm must stop and store the vector on the vector stack. Figure IV.D.2.3 shows the current pixel grid after the first vector extraction pass.

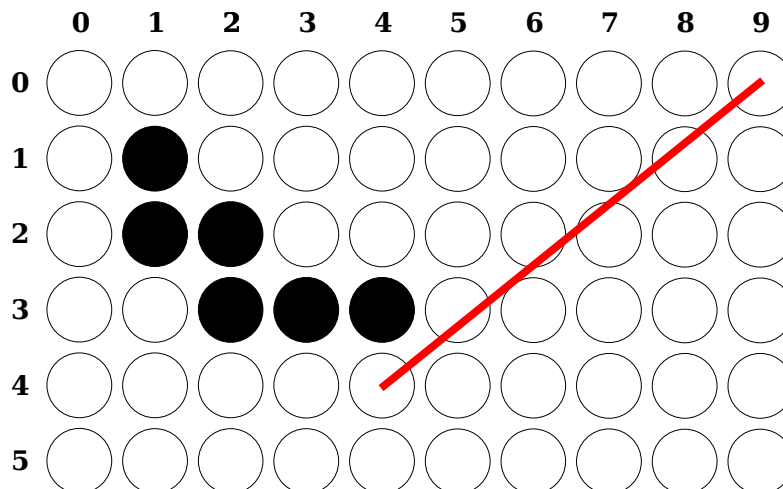


Figure IV.D.2.3. Pixel array after the first extraction pass. We've

extracted a vector from (9,0) to (4,4).

6. Now that the algorithm has successfully extracted a vector, we must start over scanning for a new starting point. In this case, the next starting point is on the next row found at pixel (1,1).
7. By repeating steps 3 and 4 for the slope of this line, we will have two vectors extracted a shown in Figure IV.D.2.4 below. The second extracted vector is shown in blue.

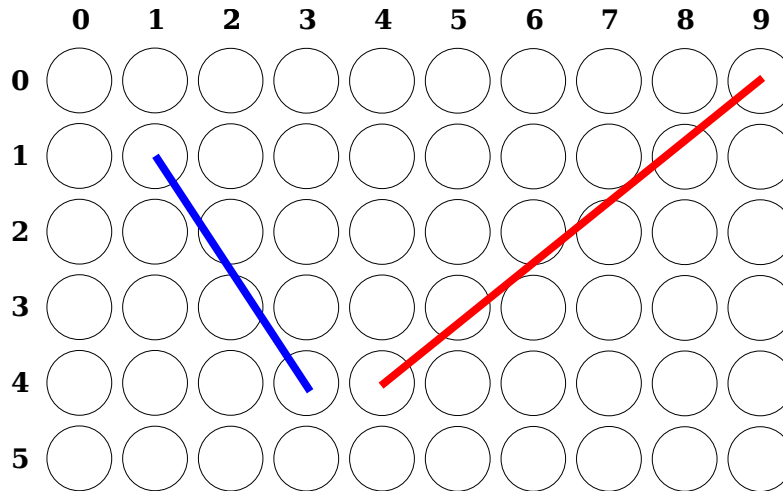


Figure IV.D.2.4. Two extracted vectors using the Silhouette vectorization algorithm.

In sum, the Silhouette vectorization algorithm is an effective, yet primitive vector extraction algorithm. It is a core component of the Silhouette Project and the shape recognition core of the application. By using the stack of extracted vectors, shape detection algorithms can then manipulate and scan the stack looking for possible shapes.

IV.D.3 Shape Recognition

Code Implementation: Section V.C

Package: edu.lmu.cs.kolich.shape

Source File: Square.java

The Silhouette project uses a basic form of vector comparisons to scan for shapes based on the extracted vector stack. The vector stack is primarily used for detecting squares and other rectangle-like variants. Circles use the popular Hough algorithm which translates an edge detected image into a series of points which is then analyzed. The Silhouette shape detection algorithm is implemented as follows:

1. Pick a starting point on the vector stack, say V_S . The first vector is often the vector on the top of the vector stack.
2. Next, we must compare the first vector, V_S , to all other vectors on the stack looking for other vectors which may fall within our tolerance level. First, we must pop off the next vector, say V_N . To compare these vectors for square like properties, we must extract both (x,y) pairs of V_N and V_S .

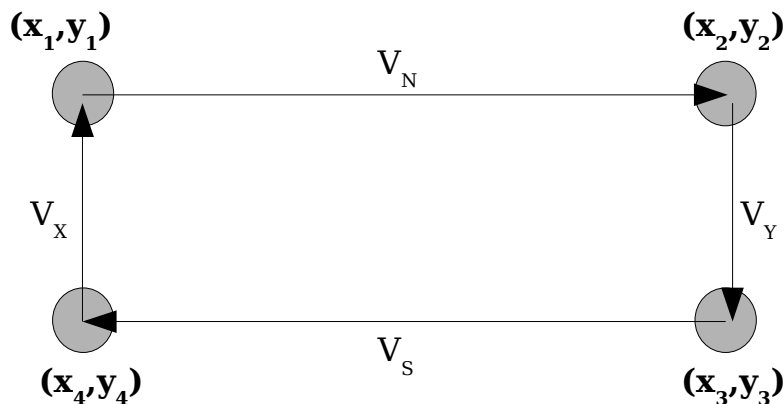


Figure IV.D.3.1. Current visual structure of vector V_N and V_S .

3. After extracting both (x,y) pairs of the vectors, we can compare their values and the values of other vectors using the following pseudo code:

```
if ( |y1 - y2| <= tolerance AND
    |x2 - x3| <= tolerance AND
    |y4 - y3| <= tolerance AND
    |x1 - x4| <= tolerance ) {

    Search for vectors in the vector stack which
        are close to  $V_x=(x1,y1) \rightarrow (x4,y4)$  AND
         $V_y=(x2,y2) \rightarrow (x3,y3)$ 

    if ( points from  $V_x$  AND  $V_y$  share common (x,y) points with
         $V_N$  and  $V_S$  ) {
        SQUARE DETECTED!
    }

} // end outer if
```

Interestingly, this algorithm is fairly efficient and tends to work for a wide variety of cases. It is important to note that our algorithm tolerance must be adjusted if we are to detect squares and other shapes which are not oriented on a flat horizontal plane. Our tolerance variable in the pseudo code-fragment above can be adjusted for any type of square variant, regardless of an angled orientation.

In sum, this algorithm is extremely primitive and could use a significant amount of overall optimization. It does, however, serve the purpose of this project application framework.