“Prototype tool for design and construction of polygonal objects with implementation in MATLAB, Simulink and FPGA”

By

Jorge Guevara Escobedo

Thesis submitted in partial fulfillment of the degree of
Master of Science

Department of Electronics
National Institute of Astrophysics, Optics and Electronics
(INAOE)
Tonantzintla, Puebla, México

Advisor:
Dr. Jorge Francisco Martínez Carballido
INAOE

ABSTRACT

The aim of computer vision is to make useful decisions about physical objects and scenes based on sensed images. Nowadays, Computer Vision has almost an unlimited field of application in which several solutions are based on polygonal object identification; commercial, scientific, industrial and military applications are some examples. Although several techniques have been introduced and developed to solve the variety of computer vision challenges, there is no accepted methodology or paradigm.

This work focuses on developing an algorithm, used to develop a software tool that allows to a computer vision designer explore in the design of applications involving the recognition of polygonal objects. Usage of this work in designing applications reduces time in the development of prototype solutions using FPGAs.

The algorithm has been developed using MATLAB and I/O equivalent implementations in a Simulink block system with the aim of ease the design of hardware description entities in VHDL.

A Spartan 3e FPGA Starter kit evaluation board is used as the device for the hardware implementation of the algorithm. Binary images are loaded into the FPGA through a micro SD memory card and the resulting data from the FPGA process is visualized through the Starter Kit built-in alphanumeric LCD.

RESUMEN

El objetivo de la visión por computadora es el tomar decisiones útiles acerca de objetos físicos y escenas contenidos en imágenes captadas. Hoy en día, la visión por computadora posee un campo casi ilimitado de aplicación en la que hay varias soluciones basadas en la identificación de objetos poligonales; aplicaciones comerciales, científicas, industriales y militares son solo algunos ejemplos. Aunque varias técnicas se han desarrollado para resolver los diversos problemas de visión por computadora, no existe una metodología o paradigma establecido.

Este trabajo se centra en desarrollar un algoritmo que sea utilizado para desarrollar una herramienta software que permita a un desarrollador en visión por computadora experimentar en el diseño de aplicaciones que involucren el reconocimiento de polígonos y además proporcione facilidad en el diseño y construcción de aplicaciones con la disminución de tiempo en el desarrollo de soluciones prototipo usando FPGAs como hardware base.

El algoritmo es desarrollado en MATLAB y además un equivalente a nivel de entrada y salida es implementado en Simulink como un sistema a bloques con el objetivo de facilitar el diseño de entidades de descripción de hardware en VHDL.

La tarjeta de evaluación Starter Kit que contiene un FPGA Spartan 3E es usado para la implementación en hardware del algoritmo. Las imágenes binarias son cargadas en el FPGA mediante una tarjeta de memoria micro SD y los resultados son visualizados mediante la LCD alfanumérica incluida en el Starter Kit.

ACKNOWLEDGMENTS

Thanks to my adviser Dr. Jorge Francisco Martínez Carballido for his teaching, time, patience, support and for being a good friend.

Thanks for all my friends at INAOE. They gave me so many good moments and supported me during this process.

Thanks to CONACYT for the economical support.

This thesis work is dedicated to my parents

José Escobedo Mera
Jorge Guevara García

Rosa Aurelia Uranga Romero
María Eugenia Escobedo Uranga

And to my sister

Alejandra Guevara Escobedo

CONTENTS TABLE

Chapter 1 INTRODUCTION

1.1 Motivation

Human vision is a sophisticated system that senses and acts on visual stimuli. One can see that human vision and computer vision share objectives because both systems have the purpose to interpret spatial data. Even though both systems are functionally similar, it is impossible to expect that human eye can be replicated by a computer vision system but there may be computer vision techniques which, in some level, can replicate or even improve the human vision system. Vision science has been developed as an interdisciplinary research field by including concepts and tools from areas like computer science, image processing, artificial intelligence, pattern recognition, computer graphics and some others thus it can be thought that one of the most stimulating research areas will be those related to the main human sense: vision.

Computer vision is a rich topic for research and study; increasingly, it has a commercial future. While the goal of computer vision is to make useful decisions about physical objects and scenes based on sensed images [1], whenever necessary to automate, or improve human activities, computer vision systems become essential.

Objects within images are essentially characterized by their shape, so in a typical computer vision application, shapes from objects within images are digitalized, pre-processed, analyzed and classified. Nowadays, these techniques have been successfully applied to a wide range of practical problems like: inspection of mechanical pieces [2] [3], agricultural products quality inspection [4] [5], circuit board inspection systems [6], medical imaging [7], automotive traffic [8], forensic studies and biometrics [9], and even entertainment, multimedia, art and design [10].

There are many problems addressed in the context of analysis and recognition in which the contour extraction and interpretation has to be made by considering polygonal shapes [11] [12] [13] [14] [15] [16].

Computer vision is one of the areas where hardware-implemented algorithms can perform better than those implemented in software, moreover that it is better not to depend on a personal computer in some real applications. Nowadays reconfigurable hardware as FPGAs is increasingly being used to design powerful hardware devices combining the main advantages of ASICs and DSPs and the ease in reprogramming, simulating and testing. By now reconfigurable hardware is a very attractive option for rapid prototyping.

Developing algorithms either in software or hardware that take an image as input and produce a symbolic interpretation describing which objects are present and some other properties from each object, is not dominated by researchers and has no accepted paradigm [1]; so that, an opportunity to develop an algorithm solution and its implementation in FPGA based hardware to extract polygonal contours and its properties on images is the main objective of this work.

Figure 1-1 shows a classification of some possible areas covered by a computer vision system.

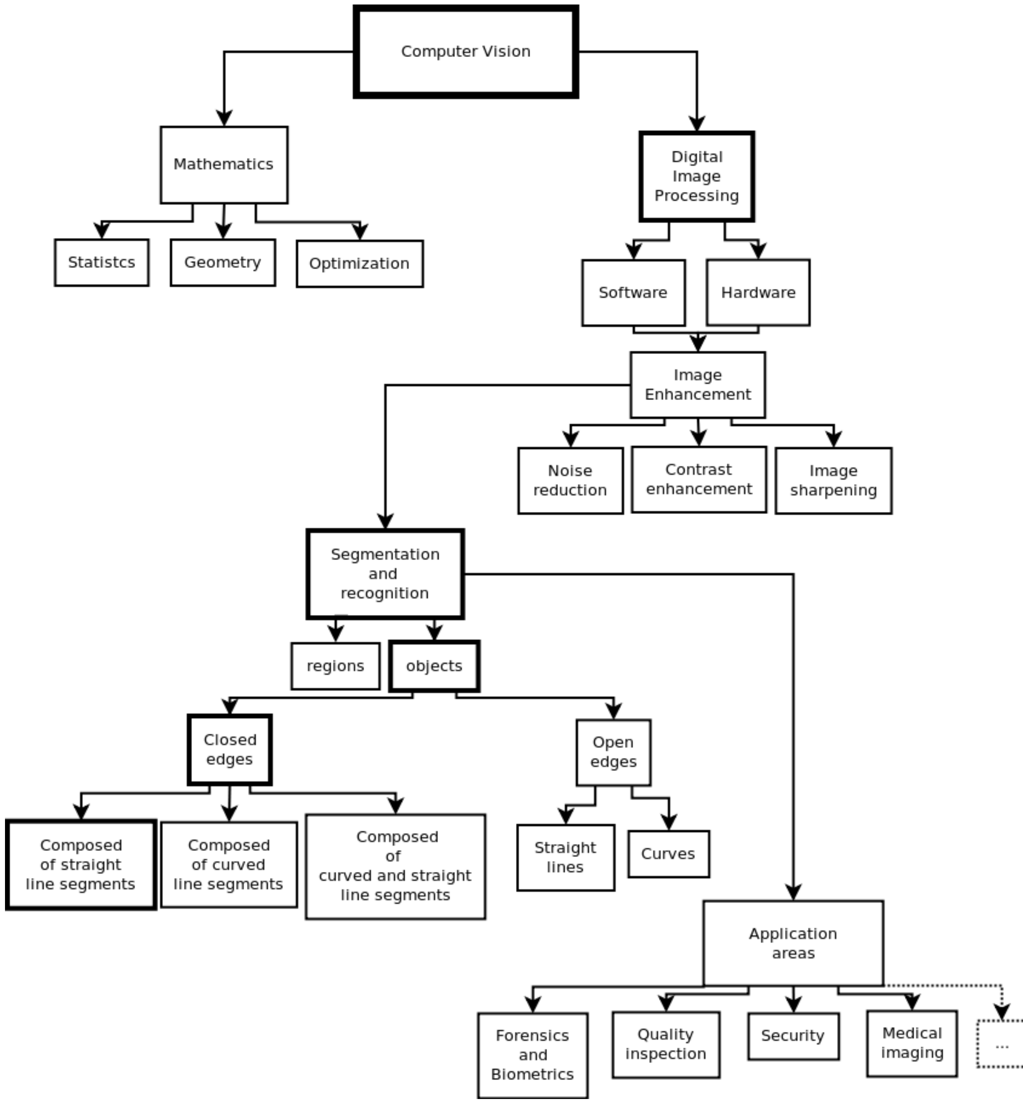


Figure 1-1 Computer Vision System [17]

1.2 Problem Description

In order to perform a computer vision application, once that an image has been acquired, segmentation has to be made with the purpose to store data representing the object of interest. A Shape can be understood as a connected set of points, so that stored data is better called shape representation. Figure 1-2 shows the computational shape representation techniques [18].

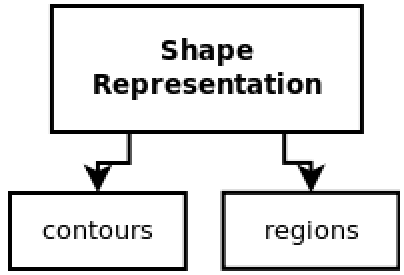


Figure 1-2 Computational Shape Representation Techniques

The contour based approach is divided into three classes as follows:

- *Parametric Contours:* the shape outline is represented by a parametric curve implying a sequential order along it.
- *Set of Contour Points:* the shape outline is simply represented as a set of points, without any special order among them;
- *Curve Approximation:* a set of geometric primitives like straight line segments are fitted to the shape outline.

The region based approach is also divided into three classes:

- *Region decomposition*: the shape region is partitioned into simpler forms and represented by the set of such primitives;
- *Bounding regions*: the shape is approximated by a special pre-defined geometric primitive fitted to it;
- *Internal features*: the shape is represented by a set of features related to its internal region.

Given a specific shape representation, it may be needed to perform a shape characterization by analyzing the shape representation. The corresponding shape characterization can be interpreted as a series of transformations taking a shape representation to a set of scalar measures and features like area, perimeter, the number of sides, the number or location of vertices or the fact of belonging to an established class or group. Figure 1-3 shows an example of a characterized shape into a set of features.

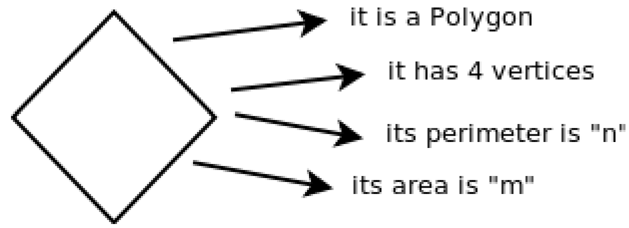


Figure 1-3 Characterized Shape

While for this work the purpose is to characterize polygonal figures on digital images, it is needed to build a system (both software and hardware), that makes it possible to extract polygon’s edges from figures, save the acquired information in a proper way for further processing of some other properties from the image, that may be needed by the application. If a polygon is defined as a plane figure that is bounded by a closed path or contour, composed of a finite sequence of straight line segments, each segment becomes a side of the figure and the intersection between two different segments is known as a corner or vertex. The problem of polygonal approximation or polygonal modeling of a given contour may be understood as locating the vertices of a polygon along the contour, in such a way, that the result is a good approximation of the given contour [18]. If vertices are extracted from the polygon’s edge, one can obtain properties such as: how many segments are involved or how many sides does the figure have; the position and orientation of each segment or the whole figures edge and even the type of polygon that has been detected.

The methods for vertex detection and polygonal approximation can be divided into two principal classes: global methods and local methods [18].

- *Global methods* are generally based on the polygonal approximation of the contour in such a way that some error function is minimized.
- *Local methods* are based on the idea of directly searching for high curvature points along the contour.

Table 1-1 shows Global and Local method examples with consulted references.

Methods of Vertex Detection and Polygonal Approximation of a Given Contour			
Global	**References**	**Local**	**References**
Approximation or Stopping Criteria	[19] Polygonal approximation of digital planar curves through break point suppression (2008).	***Coding the Object's Contour as an Ordered Sequence of Points, High-Curvature Points or as Chain-Code Histograms, Obtained by Different Techniques.***	[20] Polygon Evolution by Vertex Deletion (1999). [21] An algorithm for polygonal approximation of digitized curves (2003). [22] Polygonal shape description for recognition of partially occluded objects (2004). [23] Novel efficient two-pass algorithm for closed polygonal approximation based on LISE and curvature constraint criteria (2006). [24] Parsing Silhouettes without Boundary Curvature (2007). [25] A New Algorithm for Polygonal Approximation Based on Ant Colony Optimization (2009).
Minimization of the Mean-Squared Error between the Given Contour and the Model.	[26] Optimal polygonal approximation of digitized curves using the sum of square deviations criterion (2000).		
The Minimal Polygon Perimeter.	[27] Constrained piecewise linear approximation of digital curves (2008). [28] Minimum-Perimeter Polygons of Digitized Silhouettes (2009). [29] Two Linear-Time Algorithms for Computing the Minimum Length Polygon of a Digital Contour (2009).		
The Maximal Internal Polygon Area.	[30] Multiscale Contour Segmentation and Approximation: An Algorithm Based on the Geometry of Regular Inscribed Polygons (1996). [31] Decomposing a Simple Polygon into Trapezoids (2007).		

Table 1-1 Vertex Detection and Polygonal Approximation Methods

Processing with images on a personal computer, has many useful tools that ease experimentation to develop an algorithm; but while the main objective of the work is to design an algorithm capable of identify polygonal objects within images with its hardware implementation, the most suitable segmentation and characterization methods have to be considered.

1.3 Objectives

Design and develop a component set that as a system extracts a polygon from a boundary to serve as an element in designing a complete computer vision system that involves recognizing polygons. The work must provide a software solution either as a programming function or simulation block system to explore its usage in an application solution. Once defined the algorithm a hardware version of the work must match at input/output behavior with the designed software function and block system. To reach this work's purpose the following objectives were established:

- Design an algorithm in MATLAB to extract polygon's edge on digital binary images and save it as a list of (x y) coordinates of the pixels forming the edge.
- Define a method to extract vertices from the polygon's edge by analyzing edge's (x y) coordinates to improve the MATLAB algorithm.
- Write a MATLAB function with the designed algorithm to ease testing out as many different types of polygons on images to satisfy each potential case.
- Refactor the MATLAB algorithm into blocks to write level-2 m files of each task and build a Simulink block library.
- Design the hardware counterpart of the Simulink block in VHDL to assure input/output compatibility.
- Design, develop, and implement input/output modules for peripheral devices to interface with the FPGA evaluation board to test the VHDL system. The FPGA evaluation board will acquire the binary image from

a micro SD memory card, using FAT16 file system. The resulting data will be shown through a 16x2 alphanumeric LCD.

1.4 Proposed Solution

Established objectives give way of what has to be done, so that proposed solution is based on each objective as follows:

- The contour extraction is a common operation in digital image processing, so there are several established methods to perform it. Analyzing brightness differences between pixels is suitable common approach; although further processing has to be considered. The edge must be saved as a list of (x y) coordinates with a sequence order of the pixels forming it (parametric contour).
- Locating vertices from the acquired data has to be found by characterizing the ongoing list of the pixels forming the polygon's edge (local vertex extraction method).
- By writing a MATLAB function m-file, testing can be more interactively while input and output parameters are introduced all in a single instruction.
- Simulink blocks will be designed and developed from Level-2 m-files which allows the MATLAB m language, multiple input/output ports and supports a matrix as data type.
- The hardware counterpart of each block will be designed as a VHDL description language entity to assure the both systems match. While it is not useful to test each single VHDL entity in the FPGA, simulations in Modelsim will be performed to test correct function of each entity.
- Once all VHDL entities are tested and connected as a system, it is possible to test it in the FPGA evaluating board. The FPGA will acquire the test image from a micro SD memory card and will show resulting vertices in the built-in 16x2 alphanumeric LCD. To interface the SD memory card, communication protocol and FAT16 file system are

needed to be understood. Drivers for the SD memory card and LCD communication will be designed and developed in VHDL.

- Each mode of the system will be tested with the same image and required input parameters to assure the compatibility between each mode at input/output level.

Figure 1-4 shows a diagram of the proposed solution process.

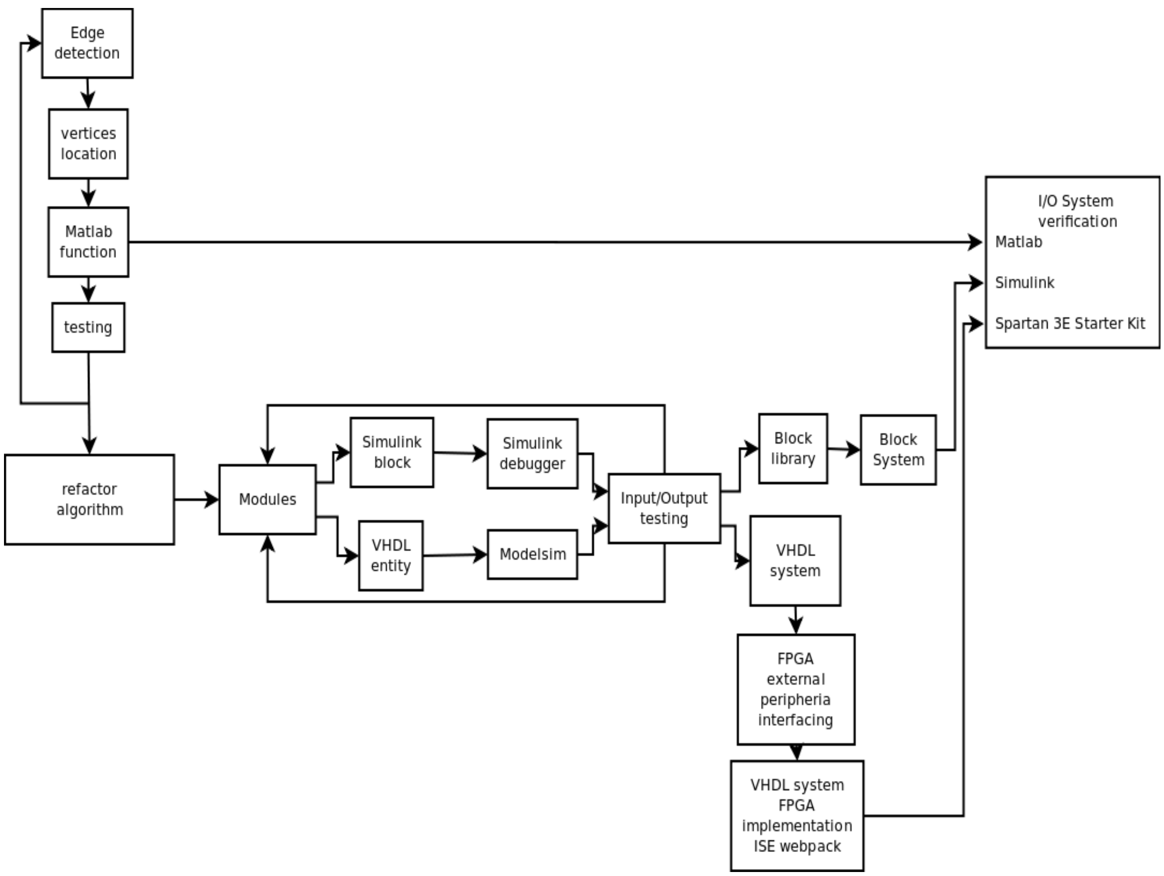


Figure 1-4 Solution Process

1.5 Organization of this Thesis Report

Chapter 2 explains how the algorithm was designed and gives details of the criteria taken to reach the objectives.

Chapter 3 shows how the algorithm was refactored and implemented as a Simulink block system.

Chapter 4 shows the VHDL implementation of the refactored algorithm and how needed peripherals were interfaced to the FPGA evaluation board.

Chapter 5 shows the obtained results for the MATLAB function, the Simulink block system and the VHDL system implementations.

Chapter 6 concludes the presented work and comments on the future work.

Chapter 2 ALGORITHM DEVELOPMENT USING MATLAB

2.1 Introduction

While the objective of this work is to extract vertices from polygonal figures in images, it is important at first to try out several ways to find out those vertices. It is important to choose a tool that gives us opportunity to readily visualize results. MATLAB does that, so in fact, the entire algorithm was first designed in MATLAB.

2.2 Corners Extraction

To extract vertices from a polygonal figure, we needed information about the edge of that figure, so the first big issue of this work is the edge extraction of the polygonal figure.

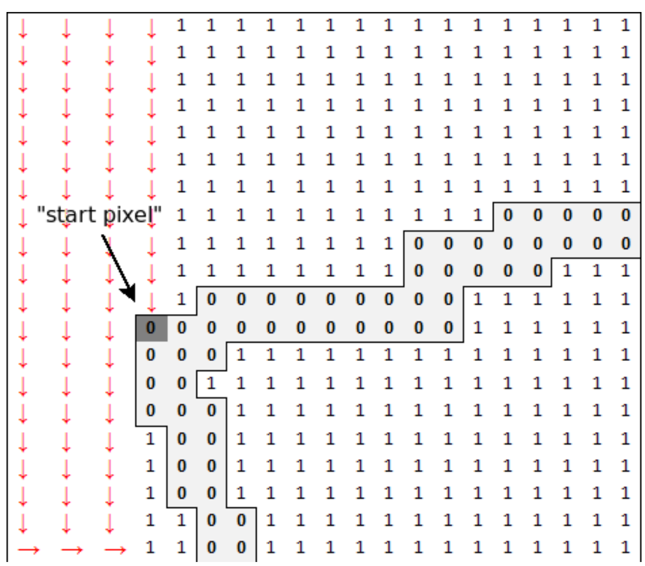


Figure 2-1 Start up pixel location

Information given by edge extraction can be known by applying some operations at pixel levels; so at first, it is important to find out the first pixel that belongs to the edge. In this case, the most “left-up” pixel is considered as the first edge pixel. This first or better called, “start pixel”, is found by analyzing each pixel, from up to down and left to right, until it is known as a “0” or “black” pixel as shown in Figure 2-1.

The “start pixel” can be found by implementing the sequence shown by Figure 2-2.

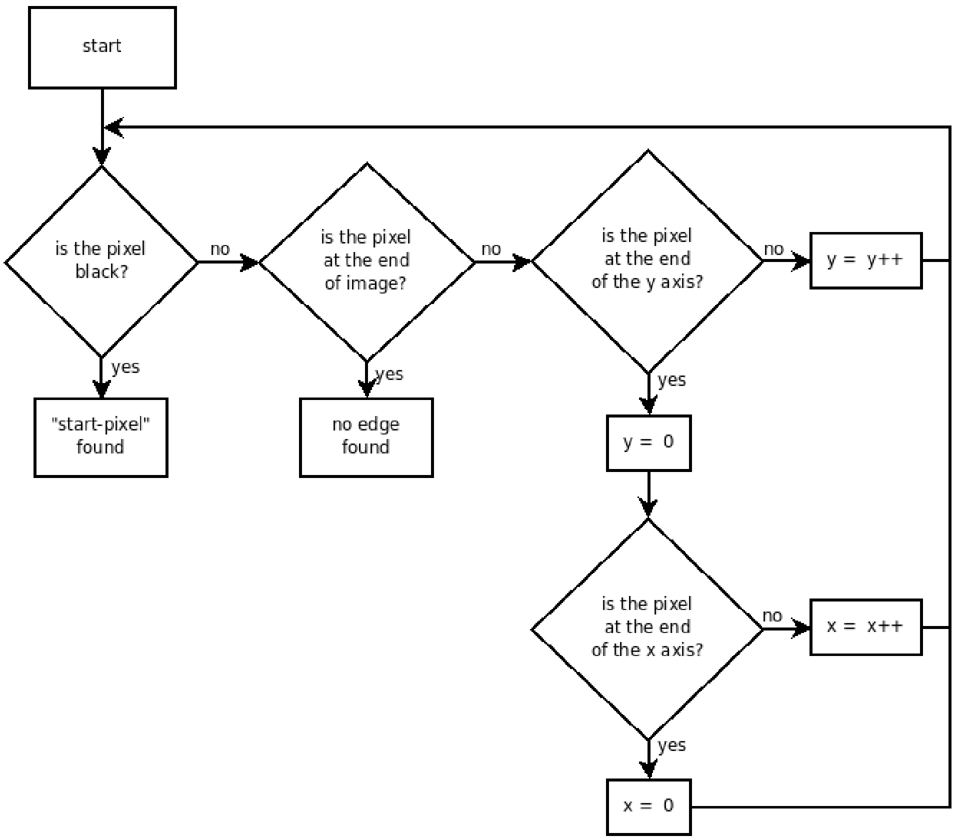


Figure 2-2 Start up pixel algorithm

Once the “start pixel” has been found, it is possible to extract the edge by analyzing the pixel’s value between the current pixel and its adjacent pixels to decide if that pixel belongs to the edge or not and save its (x y) coordinates to define a list containing the entire (x y) pixel’s coordinates conforming the edge as proposed in [32].

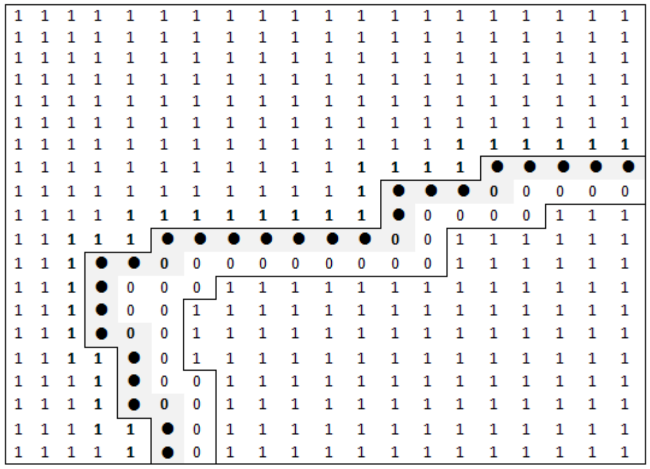


Figure 2-3 Edge extraction from adjacent pixels value differences [32]

While the objective to manage edge information is to extract vertices, it can be inferred that a vertex in a polygon is a point where two lines or segments with different slope meet, so that; a vertex has to be found in the edge's direction change. Direction changes can be interpreted as corners, so after several options were tried to find a way to extract useful information from the whole edge's (x y) coordinates list, it can be seen that just pixels that take the shape of corners are sufficient. In fact each corner becomes a vertex candidate, so one only needs to extract those corners as shown on Figure 2-4.

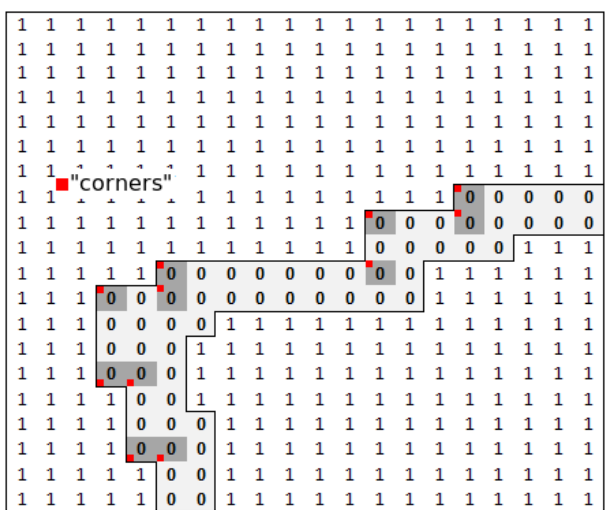


Figure 2-4 Edge's corners

Extracting corners instead of the whole edge, makes the process more complicated than just finding differences in pixel's value, but it can also give us a sense about some other properties like a defined quadrant and type of corner (external or internal). Thus, to extract corners, it is needed to define each edge quadrant and take sense of how pixels form a corner, so then it is possible to create a pattern for each kind of corner as can be seen on the Figure 2-5.

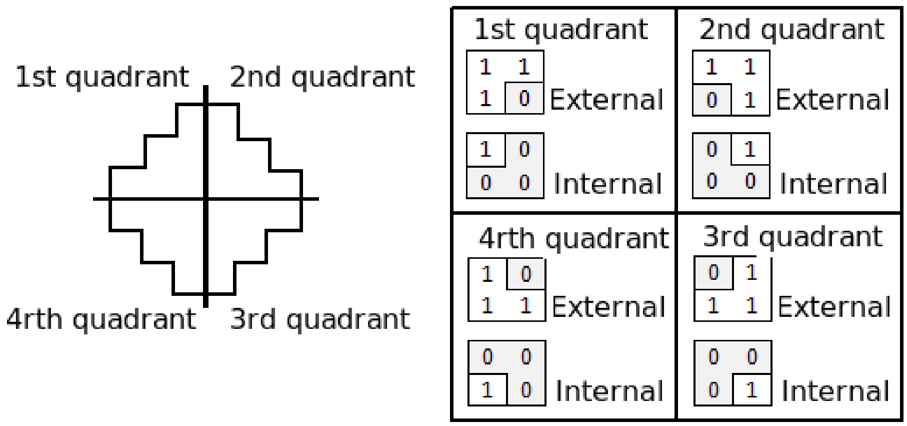


Figure 2-5 Corner patterns

Corners extraction can be then achieved by comparing the current pixel and its neighbors with the defined patterns and form a new list containing the corners: (x y) coordinates, the type of corner (external or internal) and, quadrant where the corner is situated. From the matched pattern, knowledge as where to move and obtain a new pixel group, those by repetition, the whole edge can then be scanned.

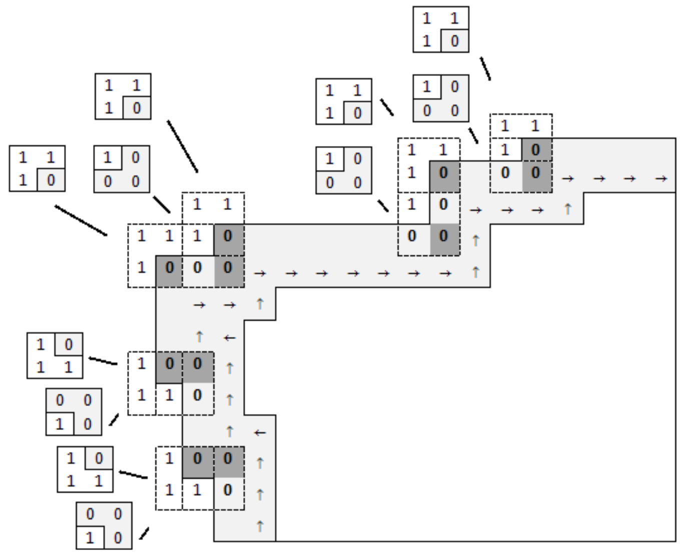


Figure 2-6 Corners extraction example

A corners list can be obtained by implementing the next sequence.

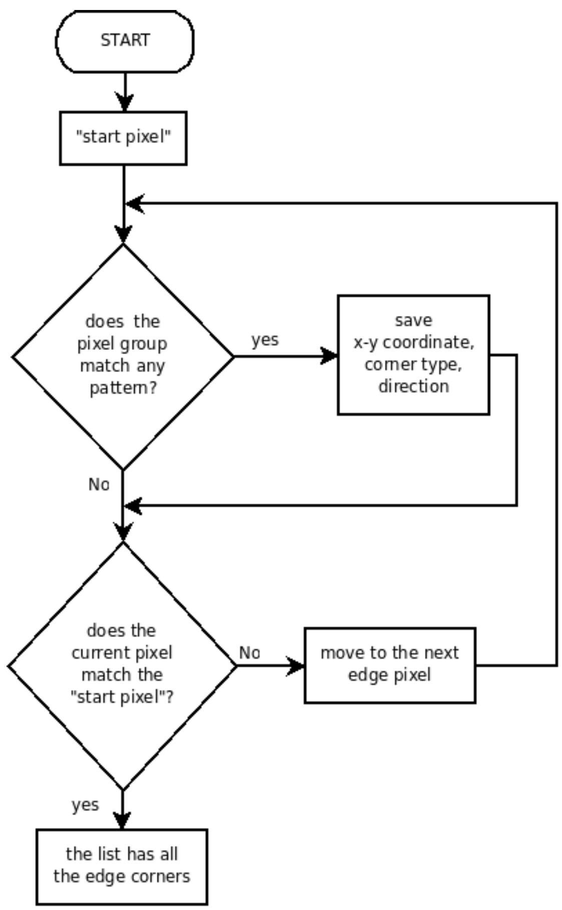


Figure 2-7 Sequence to find corners list

This way, all information needed directly from image’s polygon, has been saved in a list and is ready for further process.

2.3 Vertices Location

Locating vertices of a polygonal figure from binary images with edges detected is the main objective of this work, so by extracting vertices, it is possible to know important properties from each polygon such as the number of sides, the distance of each side, the type of polygon, its position and rotation.

Once the corners list has been obtained, useful information from the polygon’s edge is available, so vertices extraction can be performed by analyzing this list. With this list, is possible to know the (x y) coordinates of pixels that represent direction changes in the polygon’s edge, the type of corner and the quadrant where each pixel is located.

It can be seen that just few elements from this list will be considered as vertices, meaning that there is a need to analyze what kind of changes are better candidates to being a vertex.

The first and most intuitive kind of vertex is that located in quadrant changes, meaning significant slope changes and no difference in corner type as it can be observed from the list on Figure 2-8 that shows an example of vertices indentified by quadrant changes.

(x y) coordinates
Type of corner
Quadrant

x	y	t	q
6	19	1	1
7	19	0	1
7	18	1	1
8	18	0	1
8	17	1	1
9	17	0	1
9	16	1	1
10	16	0	1
10	15	1	1
11	15	0	1
11	14	1	1
12	14	0	1
12	13	1	1
13	13	0	1
13	12	1	1
14	12	0	1
14	11	1	1
15	11	0	1
15	10	1	1
16	10	0	1
16	9	1	1
17	9	0	1
17	8	1	1
18	8	0	1
18	7	1	1
19	7	0	1
19	6	1	1
20	6	0	1
20	5	1	1
21	5	0	1
21	4	1	1
22	4	0	1
22	3	1	1
34	3	1	2
34	6	1	3
33	6	0	3
33	10	1	3
32	10	0	3
32	15	1	3
31	15	0	3
31	19	1	3
30	19	0	3

30	23	1	3
29	23	0	3
29	24	1	3
27	24	0	3
27	25	1	3
26	25	0	3
26	26	1	3
25	26	0	3
25	27	1	3
24	27	0	3
24	28	1	3
23	28	0	3
23	29	1	3
22	29	0	3
22	30	1	3
21	30	0	3
21	31	1	3
20	31	0	3
20	32	1	3
19	32	0	3
19	33	1	3
18	33	0	3
18	34	1	3
17	34	0	3
17	35	1	3
16	35	0	3
16	36	1	3
13	36	1	4
13	34	0	4
12	34	1	4
12	32	0	4
11	32	1	4
11	30	0	4
10	30	1	4
10	28	0	4
9	28	1	4
9	26	0	4
8	26	1	4
8	24	0	4
7	24	1	4
7	22	0	4
6	22	1	4
6	19	1	1

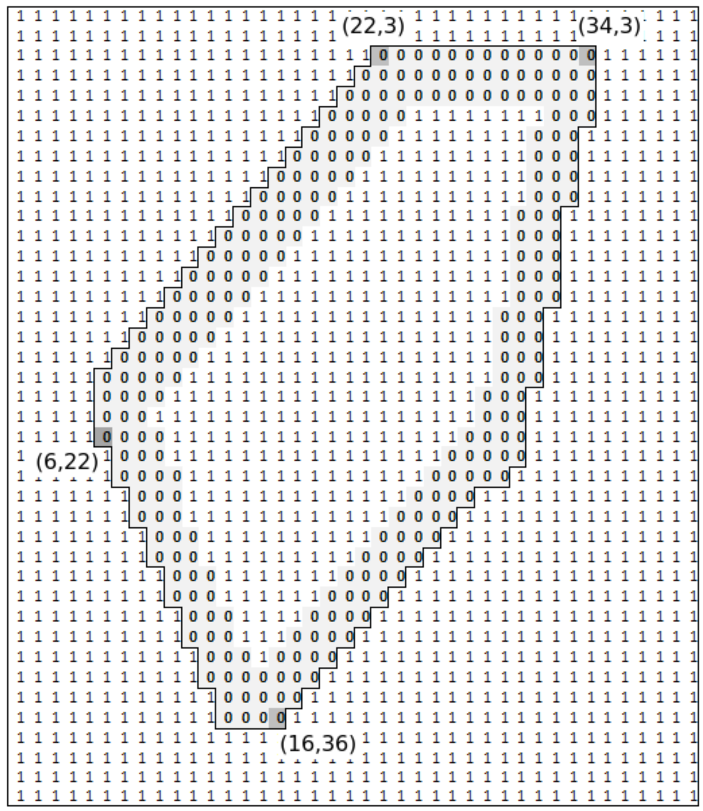


Figure 2-8 Vertices on quadrant change

From Figure 2-8, it can be seen that there is still an unidentified vertex located in a slope difference between two sides of the polygon; both are on the same quadrant, so the quadrant change condition is useless for this kind of vertex. This case is important to find out that the corners list is not enough to extract all kind of vertices from the polygon’s edge, so there is a need to find a new way to extract properties that can be used to find this type of vertices.

The only way that a vertex is located between two lines in the same quadrant, is that both lines have different slope and intersect, so to find this kind of vertex a way to find differences in slope is needed. Further FPGA implementation has to be considered; meaning that, it is important to use basic arithmetic operations as the elements to find this change in slope. Computing a line slope is not the case, so this is not an option to find this kind of vertex.

By observing on Figure 2-9, it can be seen that the number of pixels between corners determine the small changes in direction of pixel's segments of the edge, so it is possible to take the distance between corners as an indicator for the slope of that line, so if distances between corners are computed, changes in slope can then be identified and a list capable to provide additional properties can be obtained from the corners list of (x y) coordinates.

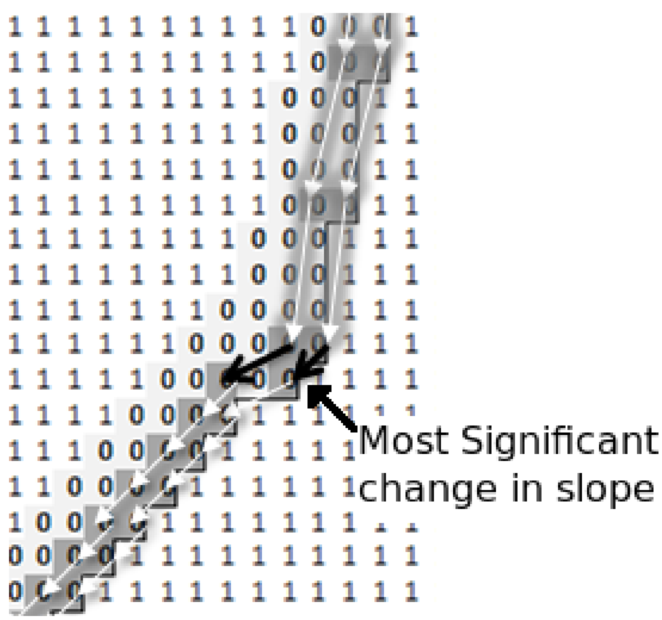


Figure 2-9 Slope Change

Distances between corners are obtained by performing an absolute subtraction between the current evaluating corners list of (x y) coordinates and its nearest last evaluated corners list of (x y) coordinates of the same type, as shown in Figure 2-10.

(x y) coordinates
Type of corner
Quadrant
Distances between corners

x	y	t	q	Δx	Δy
6	19	1	1	0	0
7	19	0	1	0	0
7	18	1	1	1	1
8	18	0	1	1	1
8	17	1	1	1	1
9	17	0	1	1	1
9	16	1	1	1	1
10	16	0	1	1	1
10	15	1	1	1	1
11	15	0	1	1	1
11	14	1	1	1	1
12	14	0	1	1	1
12	13	1	1	1	1
13	13	0	1	1	1
13	12	1	1	1	1
14	12	0	1	1	1
14	11	1	1	1	1
15	11	0	1	1	1
15	10	1	1	1	1
16	10	0	1	1	1
16	9	1	1	1	1
17	9	0	1	1	1
17	8	1	1	1	1
18	8	0	1	1	1
18	7	1	1	1	1
19	7	0	1	1	1
19	6	1	1	1	1
20	6	0	1	1	1
20	5	1	1	1	1
21	5	0	1	1	1
21	4	1	1	1	1
22	4	0	1	1	1
22	3	1	1	1	1
34	3	1	2	12	0
34	6	1	3	0	3
33	6	0	3	11	2
33	10	1	3	1	4
32	10	0	3	1	4
32	15	1	3	1	5
31	15	0	3	1	5
31	19	1	3	1	4
30	19	0	3	1	4

x	y	t	q	Δx	Δy
30	23	1	3	1	4
29	23	0	3	1	4
29	24	1	3	1	1
27	24	0	3	2	1
27	25	1	3	2	1
26	25	0	3	1	1
26	26	1	3	1	1
25	26	0	3	1	1
25	27	1	3	1	1
24	27	0	3	1	1
24	28	1	3	1	1
23	28	0	3	1	1
23	29	1	3	1	1
22	29	0	3	1	1
22	30	1	3	1	1
21	30	0	3	1	1
21	31	1	3	1	1
20	31	0	3	1	1
20	32	1	3	1	1
19	32	0	3	1	1
19	33	1	3	1	1
18	33	0	3	1	1
18	34	1	3	1	1
17	34	0	3	1	1
17	35	1	3	1	1
16	35	0	3	1	1
16	36	1	3	1	1
13	36	1	4	3	0
13	34	0	4	3	1
12	34	1	4	1	2
12	32	0	4	1	2
11	32	1	4	1	2
11	30	0	4	1	2
10	30	1	4	1	2
10	28	0	4	1	2
9	28	1	4	1	2
9	26	0	4	1	2
8	26	1	4	1	2
8	24	0	4	1	2
7	24	1	4	1	2
7	22	0	4	1	2
6	22	1	4	1	2
6	19	1	1	0	3

Figure 2-10 Corners list and Distances between corners list

Now it can be seen that changes in slope are identified, and also from the same list, how changes in quadrant can still be met by watching that when a change in quadrant happens, an axis remains in the same position, so the distance in either axis x or y, is zero. This property is also useful to identify vertical or horizontal lines represented by a zero distance in one coordinate and its pixels' length on the other. On the other side having non-zero distance in both axes means that between both corners, there is a sloped line. Figure 2-11 shows types of vertices identified in a polygon.

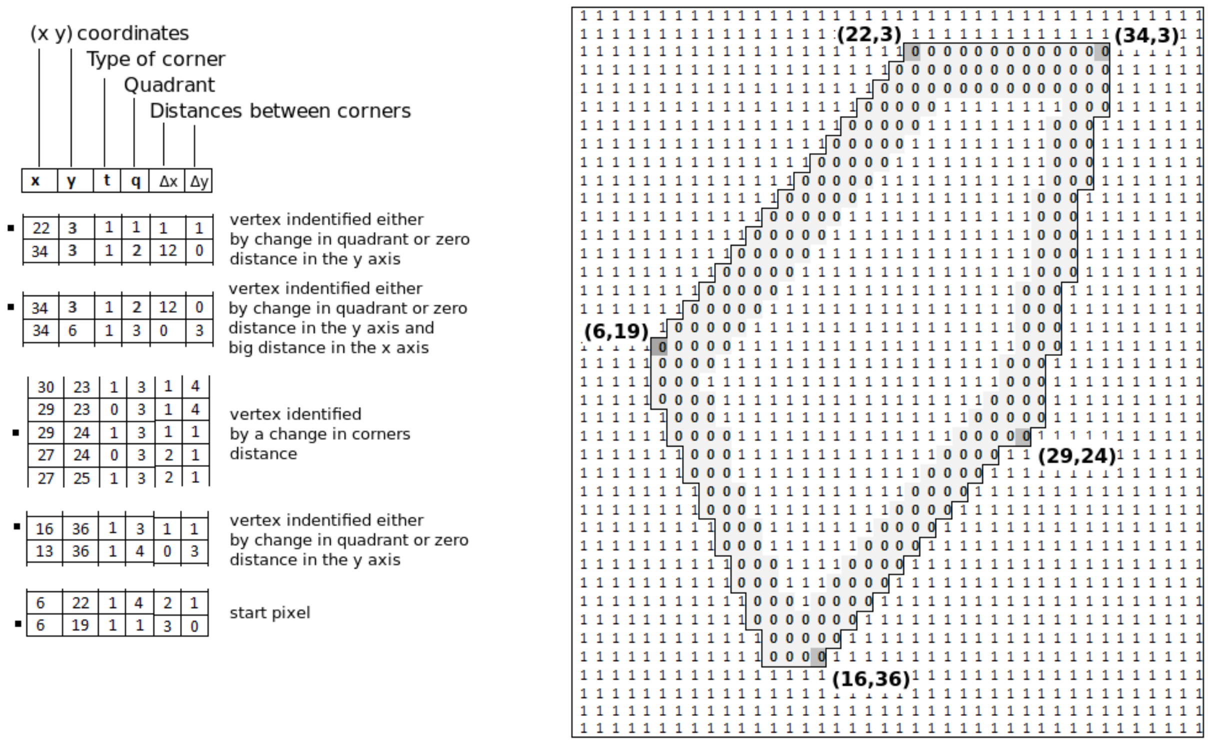


Figure 2-11 Identified Vertices

A new list containing distances between corners of the same type can be obtained by implementing the sequence shown in Figure 2-12.

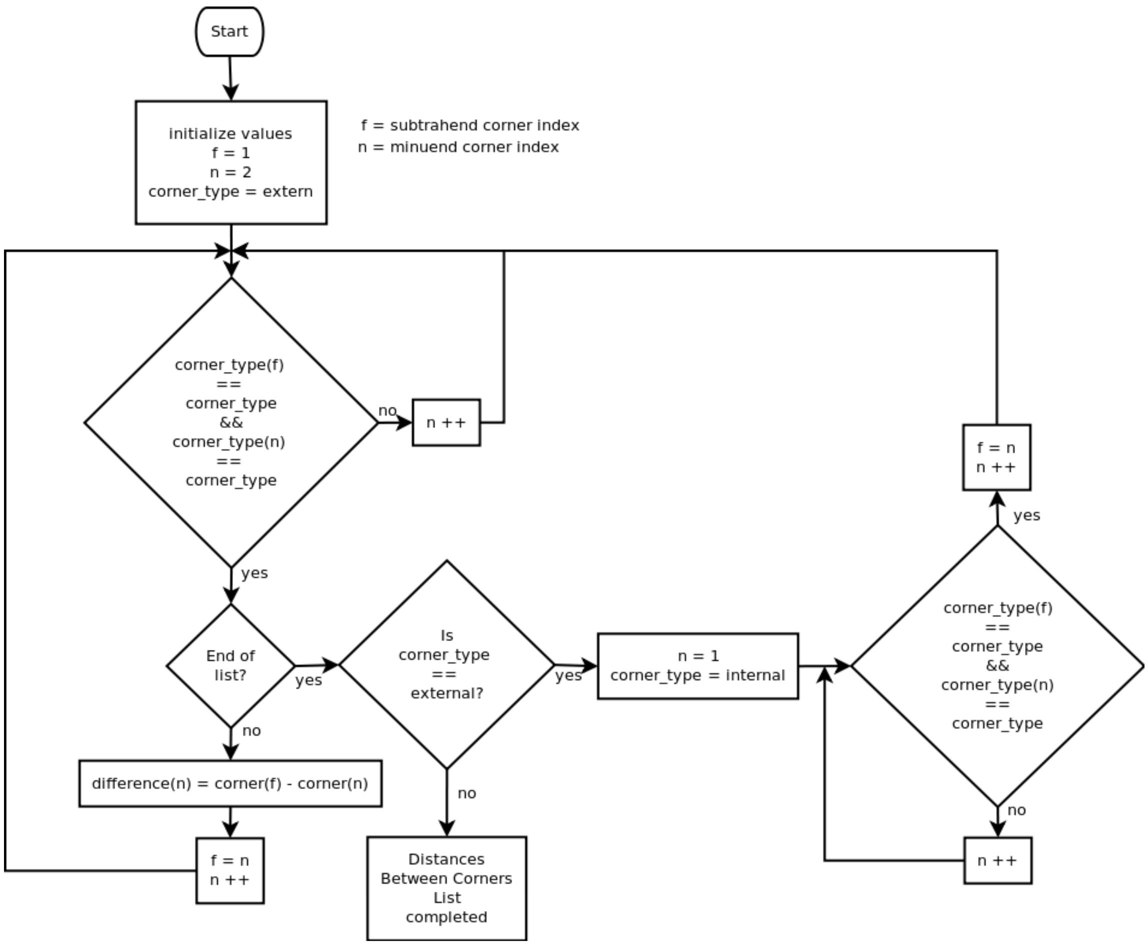
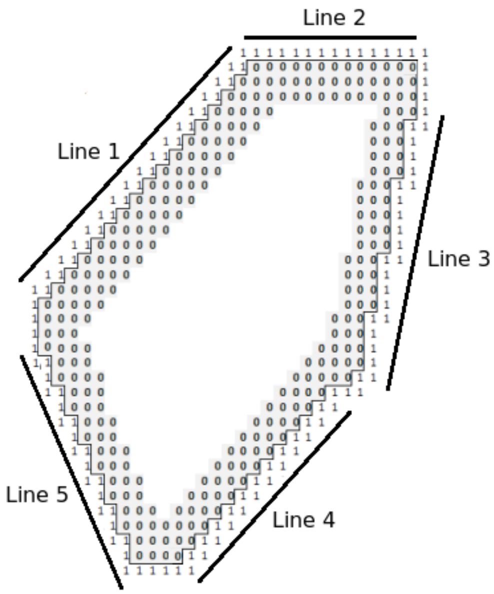


Figure 2-12 Distances Between Corners Algorithm

By now, it seems that all necessary information from the image has been saved in the distances between corners list, so the next step is to manage all the information in a manner that all kind of vertices can be identified. From the list, it can be seen that the most difficult vertex to be computationally identified is the one located in a slope change in the same quadrant.

By analyzing the distances between corners list, it can be seen that coordinates absolute differences keep a pattern that identifies the line they included; using this characteristic, one can observe that changes in that pattern means that there is a slope change, regardless of quadrant change or not. The distances between corners show that a line is formed by a combination of no more than two different distances; so by identifying two different distances to form a pattern and differentiate one line from another. Figure 2-13 shows how distances between corners form a pattern.



Line 1

Δx	Δy
0	0
0	0
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1

start pixel

45° sloped line pattern = [1 1]

Line 2

Δx	Δy
1	1
12	0

Horizontal line zero distance in the y axis

Line 3

Δx	Δy
1	4
1	4
1	5
1	5
1	4
1	4
1	4
1	4
1	1
2	1

pattern = [1 4] [1 5]

Line 4

Δx	Δy
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1

45° sloped line pattern = [1 1]

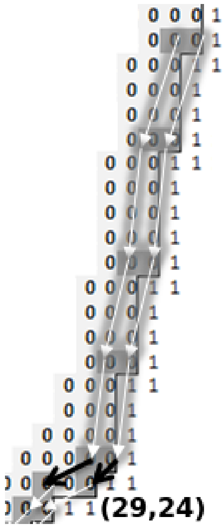
Line 5

Δx	Δy
1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2
1	2

pattern = [1 2]

Figure 2-13 Patterns of distances between corners

This is possible by getting a pattern at the beginning of the line segment and searching for a mismatch down the list that identify a slope change as in Figure 2-14.



(x y) coordinates
Type of corner
Quadrant
Distances between corners

x	y	t	q	Δx	Δy
33	10	1	3	1	4
32	10	0	3	1	4
32	15	1	3	1	5
31	15	0	3	1	5
31	19	1	3	1	4
30	19	0	3	1	4
30	23	1	3	1	4
29	23	0	3	1	4
29	24	1	3	1	1
27	24	0	3	2	1

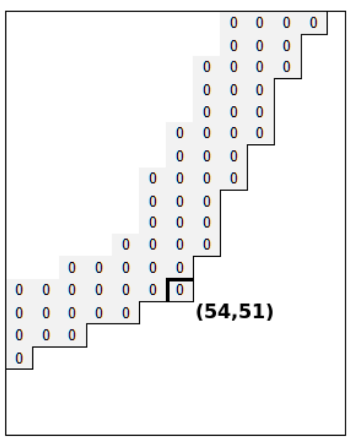
← Pattern mismatch (row 29, 24)

Pattern

1	5
1	4

Figure 2-14 Slope's Change

If the type of corner is considered, a subsequent decision can be taken for a better location of the vertex, Figure 2-15 and Figure 2-16 shows each case.



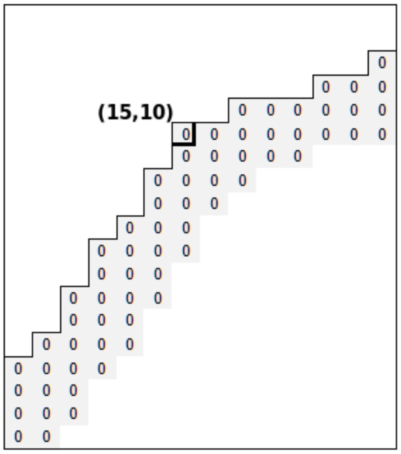
x	y	corner type	quadrant	Δx	Δy
54	48	0	3	1	3
54	51	1	3	1	3
51	51	0	3	3	3
51	52	1	3	3	1
49	52	0	3	2	1
49	53	1	3	2	1
47	53	0	3	2	1

pattern mismatch (row 51, 52)

pattern	
1	3
3	3

for corner_type = external
vertex = (x y)coordinate (n-2)

Figure 2-15 External Corner Mismatch



x	y	corner type	quadrant	Δx	Δy
13	14	1	1	1	1
14	14	0	1	1	1
14	12	1	1	1	2
15	12	0	1	1	2
15	10	1	1	1	2
17	10	0	1	2	2 — pattern mismatch
17	9	1	1	2	1

pattern	
1	1
1	2

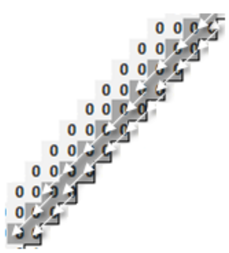
for corner_type = internal
vertex = (x y)coordinate (n-1)

Figure 2-16 Internal Corner Mismatch

From Figure 2-13, it can be seen that distances between corners in lines 1, 4 and 5 have a single distance pattern until there is a slope change; this leads for the need to identify the number of contiguous corner distances that belong to a single line. This can be done with a single parameter as input that gives the minimum equal contiguous distances to consider it as a single line. If distances are equal in both axes, we know that it is a 45° line. Figure 2-17 shows an example of a 45° line and its distances' list between corners.

Distances between corners
(x y) coordinates

x	y	Δx	Δy
26	25	1	1
26	26	1	1
25	26	1	1
25	27	1	1
24	27	1	1
24	28	1	1
23	28	1	1
23	29	1	1
22	29	1	1
22	30	1	1
21	30	1	1
21	31	1	1
20	31	1	1
20	32	1	1
19	32	1	1
19	33	1	1
18	33	1	1
18	34	1	1
17	34	1	1
17	35	1	1
16	35	1	1
16	36	1	1



45° Sloped Line

An input parameter <= 24 of equal contiguous distances is needed in this case to consider just one distance as a pattern

Figure 2-17 45° Line

The next and less complex type of vertex is the one located in quadrant changes, as explained before, it can also be taken from the distances between corners list by identifying at least one zero distance between any corner, that means that a corner has just changed in one axis, so that the line is taking a new direction in a new quadrant. As in finding a better location for the vertex between a slope's change in the same quadrant, by considering the type of corner, a better location of the vertex can also be met in the quadrant change vertex by considering the quadrant information. Figure 2-21 to Figure 2-21 shows the cases that can happen in deciding a better location of the vertex for each quadrant.

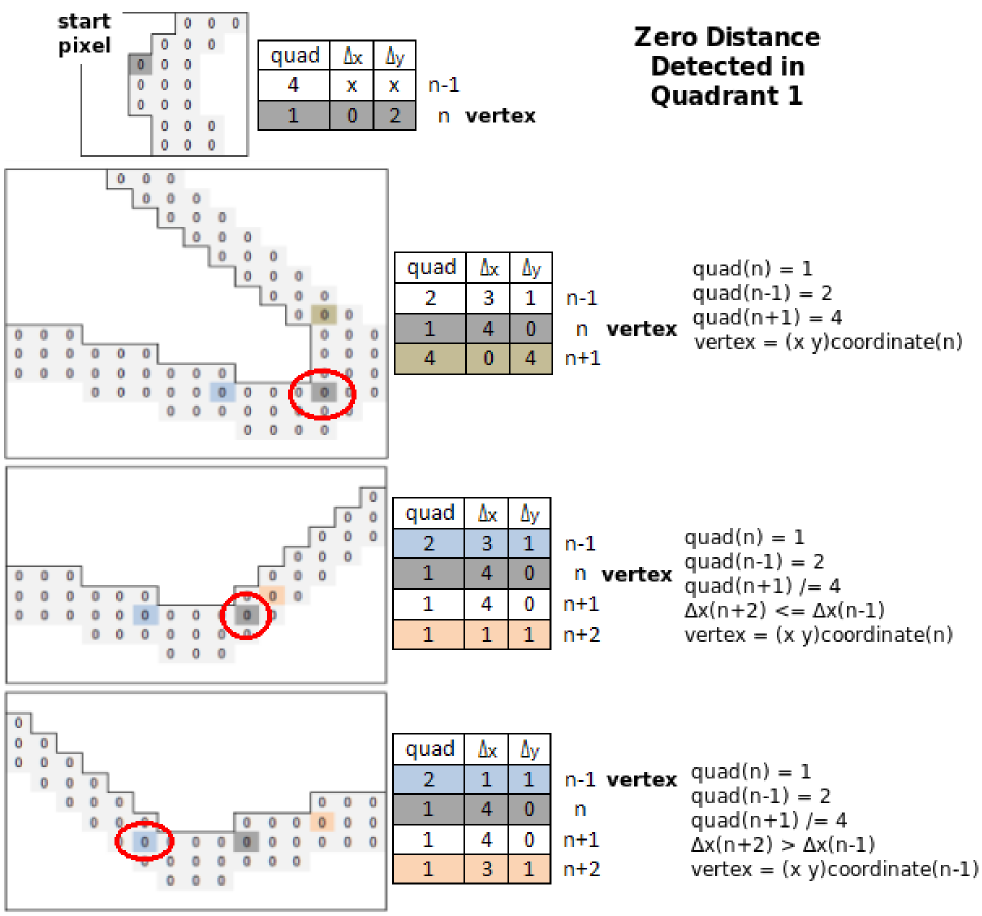
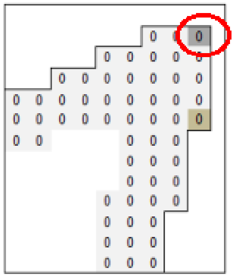


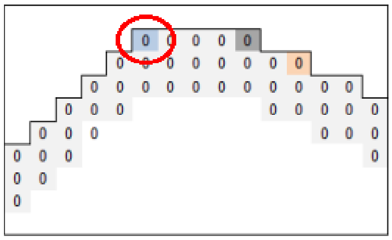
Figure 2-18 Quadrant 1 Vertex Better Locations

Zero Distance Detected in Quadrant 2



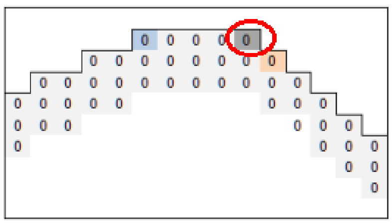
quad	Δx	Δy	
1	2	1	n-1
2	2	0	n **vertex**
3	0	4	n+1

quad(n) = 2
quad(n-1) = 1
quad(n+1) = 3
vertex = (x y)coordinate(n)



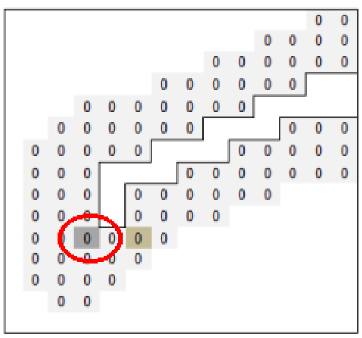
quad	Δx	Δy	
1	1	1	n-1 **vertex**
2	4	0	n
2	4	0	n+1
2	2	1	n+2

quad(n) = 2
quad(n-1) = 1
quad(n+1) = 2
Δx(n+2) > Δx(n-1)
vertex = (x y)coordinate(n-1)



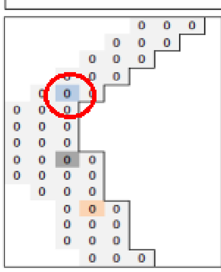
quad	Δx	Δy	
1	2	1	n-1
2	4	0	n **vertex**
2	4	0	n+1
2	1	1	n+2

quad(n) = 2
quad(n-1) = 1
quad(n+1) = 2
Δx(n+2) <= Δx(n-1)
vertex = (x y)coordinate(n)



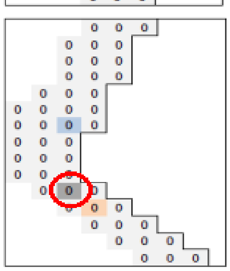
quad	Δx	Δy	
3	2	1	n-1
2	0	4	n **vertex**
1	2	0	n+1

quad(n) = 3
quad(n-1) = 2
quad(n+1) = 1
vertex = (x y)coordinate(n)



quad	Δx	Δy	
3	1	1	n-1 **vertex**
2	0	4	n
2	0	4	n+1
2	1	3	n+2

quad(n) = 2
quad(n-1) = 3
quad(n+1) = 2
Δy(n+2) > Δy(n-1)
vertex = (x y)coordinate(n-1)

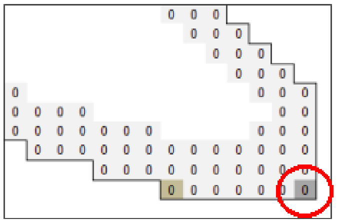


quad	Δx	Δy	
3	1	3	n-1
2	0	4	n **vertex**
2	0	4	n+1
2	1	1	n+2

quad(n) = 2
quad(n-1) = 3
quad(n+1) = 2
Δy(n+2) <= Δy(n-1)
vertex = (x y)coordinate(n)

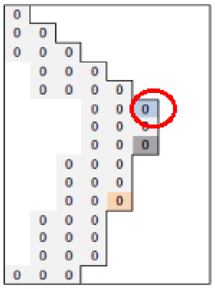
Figure 2-19 Quadrant 2 Vertex Better Locations

Zero Distance Detected in Quadrant 3



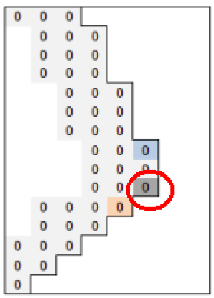
quad	Δx	Δy	
2	1	1	n-1
3	0	5	n **vertex**
4	6	0	n+1

quad(n) = 3
quad(n-1) = 2
quad(n+1) = 4
vertex = (x y)coordinate(n)



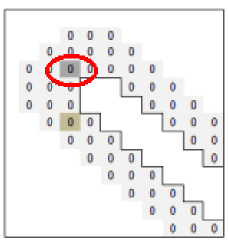
quad	Δx	Δy	
2	1	1	n-1 **vertex**
3	0	2	n
3	0	2	n+1
3	1	3	n+2

quad(n) = 3
quad(n-1) = 2
quad(n+1) = 3
Δy(n+2) > Δy(n-1)
vertex = (x y)coordinate(n-1)



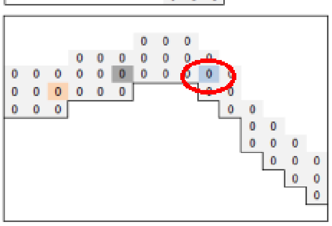
quad	Δx	Δy	
2	1	3	n-1
3	0	2	n **vertex**
3	0	2	n+1
3	1	1	n+2

quad(n) = 3
quad(n-1) = 2
quad(n+1) = 3
Δy(n+2) <= Δy(n-1)
vertex = (x y)coordinate(n)



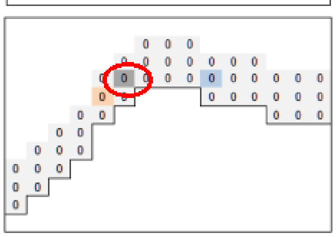
quad	Δx	Δy	
4	1	1	n-1
3	3	0	n **vertex**
2	0	3	n+1

quad(n) = 4
quad(n-1) = 3
quad(n+1) = 2
vertex = (x y)coordinate(n)



quad	Δx	Δy	
4	1	1	n-1 **vertex**
3	4	0	n
3	4	0	n+1
3	3	1	n+2

quad(n) = 3
quad(n-1) = 4
quad(n+1) = 3
Δx(n+2) > Δx(n-1)
vertex = (x y)coordinate(n-1)

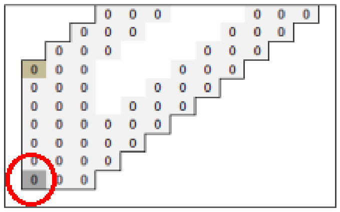


quad	Δx	Δy	
4	3	1	n-1
3	4	0	n **vertex**
3	4	0	n+1
3	1	1	n+2

quad(n) = 3
quad(n-1) = 4
quad(n+1) = 3
Δx(n+2) <= Δx(n-1)
vertex = (x y)coordinate(n)

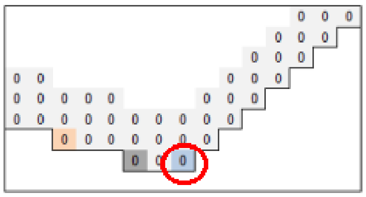
Figure 2-20 Quadrant 3 Vertex Better Locations

Zero Distance Detected in Quadrant 4



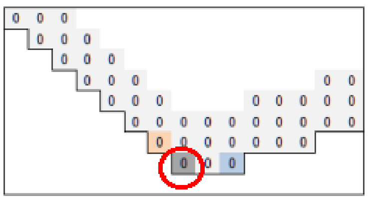
quad	Δx	Δy	
3	1	1	n-1
4	2	0	n **vertex**
1	0	6	n+1

quad(n) = 4
quad(n-1) = 3
quad(n+1) = 1
vertex = (x y)coordinate(n)



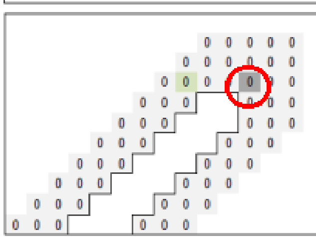
quad	Δx	Δy	
3	1	1	n-1 **vertex**
4	2	0	n
4	2	0	n+1
4	3	1	n+2

quad(n) = 4
quad(n-1) = 3
quad(n+1) = 4
Δx(n+2) > Δx(n-1)
vertex = (x y)coordinate(n-1)



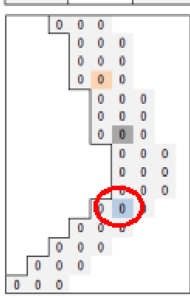
quad	Δx	Δy	
3	3	1	n-1
4	2	0	n **vertex**
4	2	0	n+1
4	1	1	n+2

quad(n) = 4
quad(n-1) = 3
quad(n+1) = 4
Δx(n+2) <= Δx(n-1)
vertex = (x y)coordinate(n)



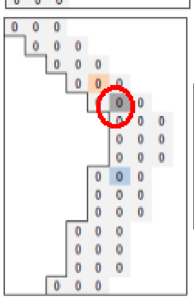
quad	Δx	Δy	
1	1	1	n-1
4	0	3	n **vertex**
3	3	0	n+1

quad(n) = 4
quad(n-1) = 1
quad(n+1) = 3
vertex = (x y)coordinate(n)



quad	Δx	Δy	
1	1	1	n-1 **vertex**
4	0	4	n
4	0	4	n+1
4	1	3	n+2

quad(n) = 4
quad(n-1) = 1
quad(n+1) = 4
Δy(n+2) > Δy(n-1)
vertex = (x y)coordinate(n-1)



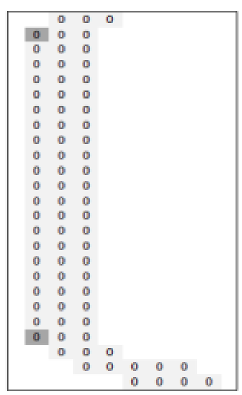
quad	Δx	Δy	
1	1	3	n-1
4	0	4	n **vertex**
4	0	4	n+1
4	1	1	n+2

quad(n) = 4
quad(n-1) = 1
quad(n+1) = 4
Δy(n+2) <= Δy(n-1)
vertex = (x y)coordinate(n)

Figure 2-21 Quadrant 4 vertex better locations

The missing type of vertices are those located at the beginning and end of an horizontal/vertical line, if the corner of the beginning and end of the line are of the same type, vertices can be extracted by implementing the same condition explained before for a quadrant change vertex, and considering that the distance different from zero, be longer than an introduced parameter. If corners are not of the same type, there will not be a zero distance, so it is just needed to identify a long distance either in axis x or in axis y. By applying those conditions, vertices can be found and the last evaluated and current evaluating distances between corners (x y) coordinates need to be saved. The long distance criteria have to be taken from comparing the current distance between corners and an input parameter called: “the minimum length to consider the distance between corners as a horizontal/vertical line”. Figure 2-22 shows both cases.

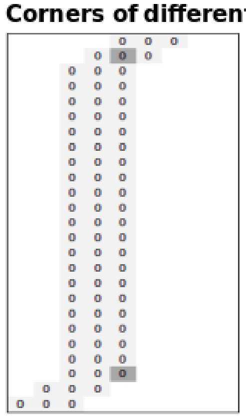
Corners of the same type



For a minum length input parameter < 20

Corner type	Δx	Δy	
1	1	1	Vertex at the beginning
1	0	20	Vertex at the end

Corners of different type



For a minum length input parameter < 20

Corner type	Δx	Δy	
1	1	1	Vertex at the beginning
0	1	20	Vertex at the end

Figure 2-22 Horizontal/Vertical line vertices locations

Now we can see that using the distance information from the corners is possible to obtain vertices for a polygonal line. Vertices can be classified into the following four types:

1. Quadrant change.
2. At the beginning and end of a vertical/horizontal line.
3. Change in slope for lines in the same quadrant.
4. Change in slope for lines of minimum length segment of single distance element.

Table 2-1 shows the kind of vertices and its conditions that have to be considered while evaluating the distances between corners list.

Vertex Type	Conditions
Quadrant change	This can be identified by a zero-distance between corners either in axis X or Y
Beginning and end of a horizontal/vertical line	This can be identified by a combination of a zero-distance between corners in an axis and a big distance between corners in the other axis (the big distance judgment is taken from a minimum length of an horizontal/vertical line input parameter)
Different slope lines in the same quadrant identified by a two-element pattern	Contiguous segment of two-elements pattern
Different slope in the same quadrant identified by a one-element pattern segment of minimum length	Consider a line of one-element pattern when the contiguous corners with the same distance is at least some specified minimum of corners

Table 2-1 Possible identifiable vertices

From Table 2-1 we observe that algorithm’s decisions are based on two input parameters and corners features like: *distance, type, and quadrant*. Figure 2-23 shows a general diagram of how to locate vertices.

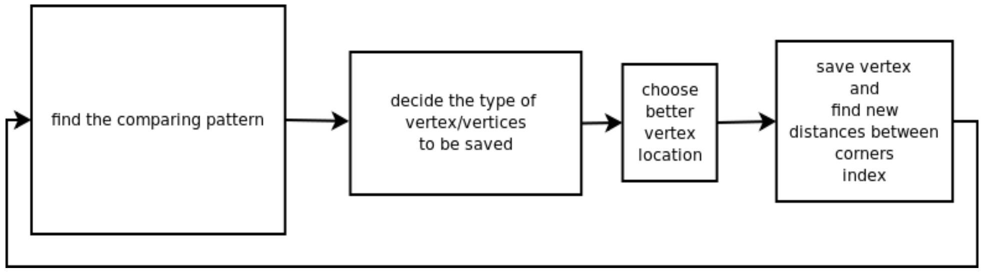


Figure 2-23 Vertices Location Process

Figure 2-24 shows a more detailed diagram of how the pattern can be found.

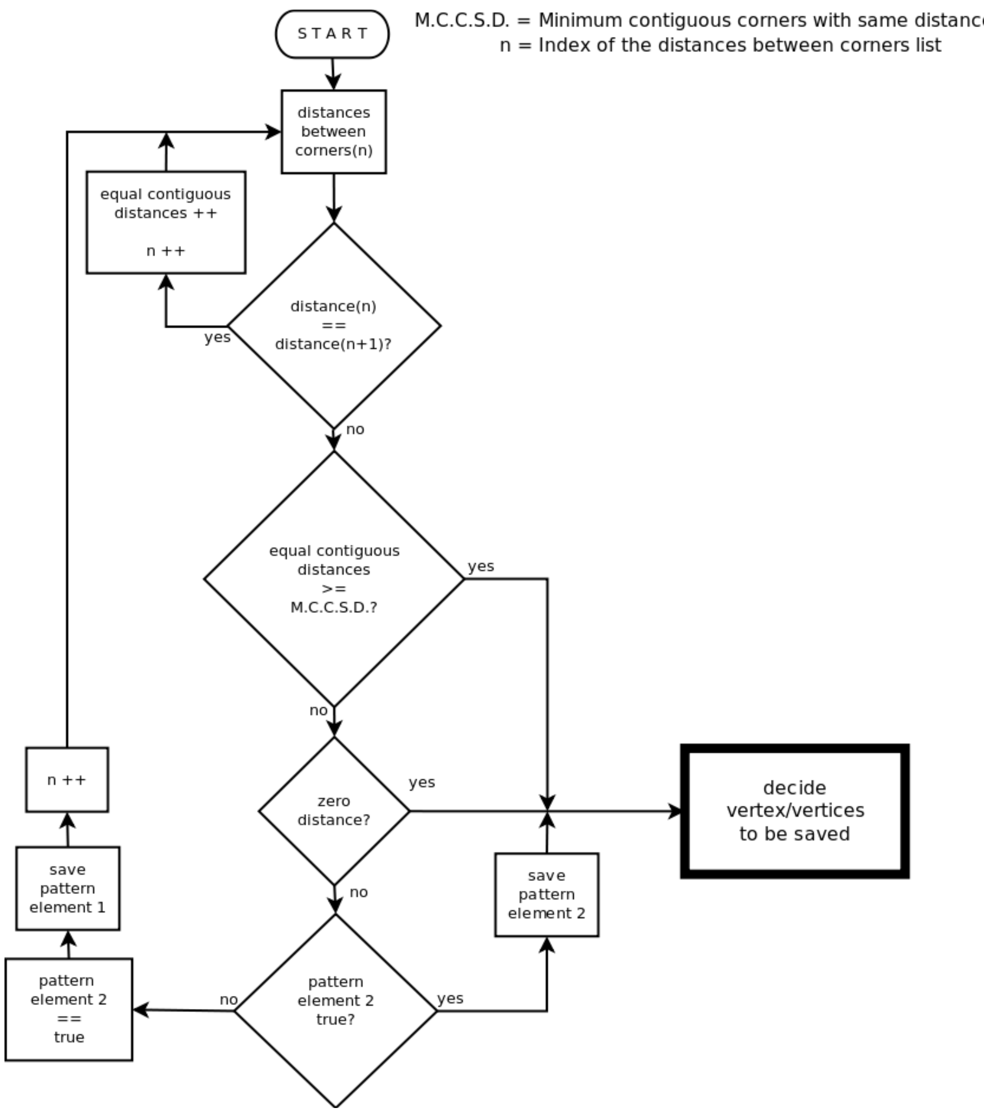


Figure 2-24 Pattern Extraction Algorithm

Figure 2-25 details how to decide which of the candidate corners becomes a vertex.

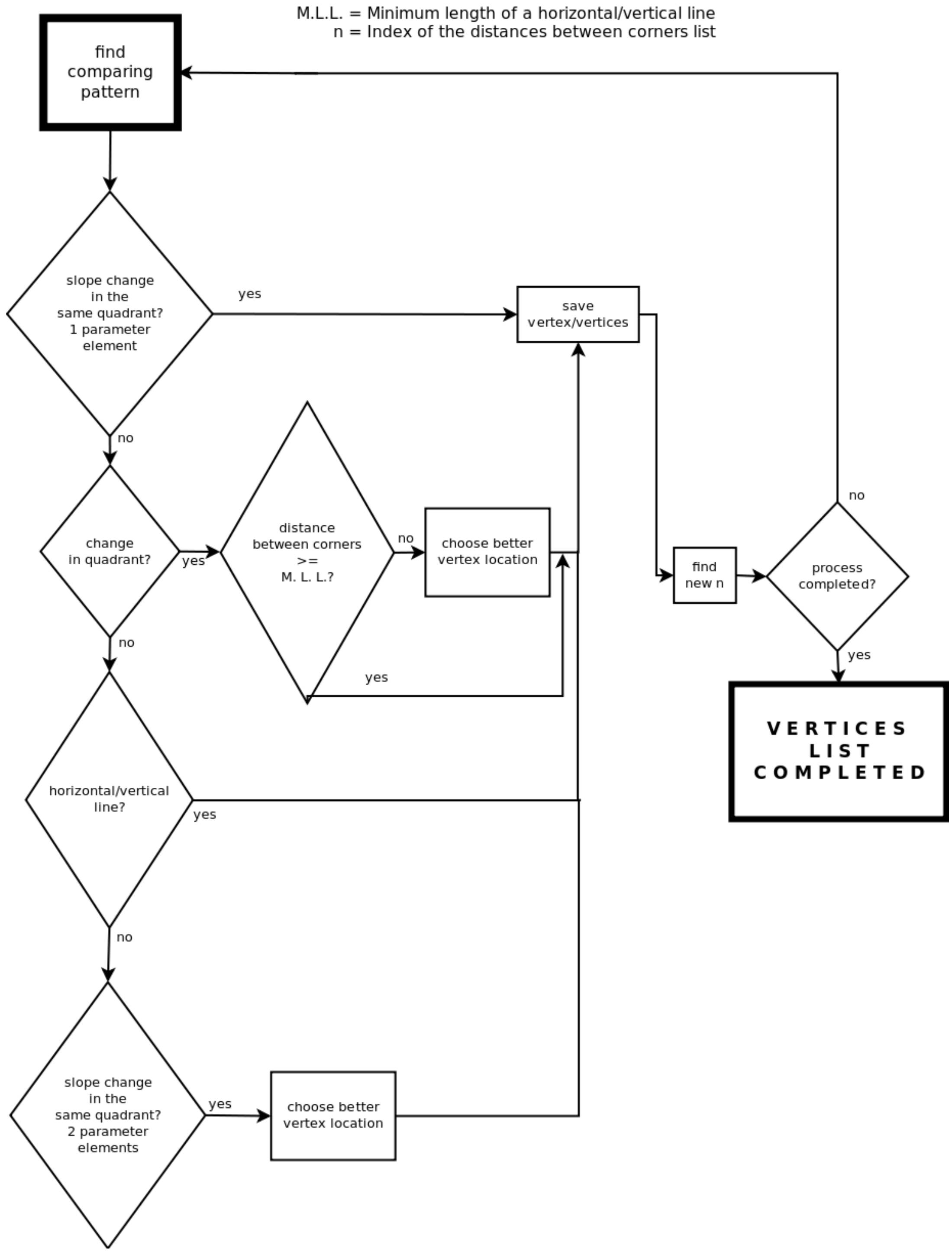


Figure 2-25 Vertices Location Algorithm

The algorithm is represented by Figure 2-24 and Figure 2-25; pseudocode is also presented for a better understanding of the vertices location process.

n *= Index of the distances between corners list and corners (x y)coordinates list.*

distances *= Distances between corners list.*

corner *= Corner (x y)coordinates*

better_vertex_location *= Taken from the criteria explained in for each case.*

M.C.C.S.D. *= Minimum contiguous corners with same distance input parameter.*

M.L.L. *= Minimum length of a horizontal/vertical line input parameter.*

```
while (n < length(distances))
      while (true)
            / **search for pattern elements**/

            if distances(n) == distances(n+1)
               n++
               contiguous_distances ++
            elsif pattern_element_1_saved == false
               save pattern_element_1
               pattern_element_1_saved = true
               if contiguous_distances >= M.C.C.S.D.
                   slope_change_in_same_quadrant_1_pattern_element = true
                  break
               end if
            elsif  distances(n,x) == 0 or distances(n,y) == 0
               change_in_quadrant = true
               break
            else
               save pattern_element_2
               contiguous_distances = 0
               n ++
               /**pattern mismatch**/

               while(pattern(1)==distances(n)or pattern(2)==distances(n))
                   n ++
                   if  distances(n,x) == 0 or distances(n,y) == 0
                       change_in_quadrant = true
                       break
                   end if
              end while
            end if
      end while
     /*a change in slope in the same quadrant with one pattern element*/

      if slope_change_in_same_quadrant_1_pattern_element == true
         save corner(n)
      /*a change in quadrant*/

      elsif change_in_quadrant == true
```

```
            /*a horizontal/vertical line is in the change in quadrant*/
            if distances(n,x) >= M.L.L. or distances(n,y) >= M.L.L.
               save corner(n-1)
               save corner(n)
            else
               save corner(better_vertex_location)
            end
           /*there is a horizontal/vertical line*/

         elsif distances(n,x) >= M.L.L. or distances(n,y) >= M.L.L.
            save corner(n-1)
            save corner(n)
/*a slope change in the same quadrant with 2 pattern elements located by a mismatch*/

         else
            save corner(better_vertex_location)
         end if
   end while
```

2.4 Summary

This chapter explained how the MATLAB algorithm was designed. The chapter focused on laying out criteria taken to develop each part of the algorithm and the procedure followed to reach the established objectives. As first step it presented why and how the start pixel is obtained; second, the reason for just extracting corners instead of the whole set of pixels for the polygon's edge (as in [ref]). Figure 2-26 and Figure-2-27 shows a comparison on the proposed contour representation and the one proposed in [32].

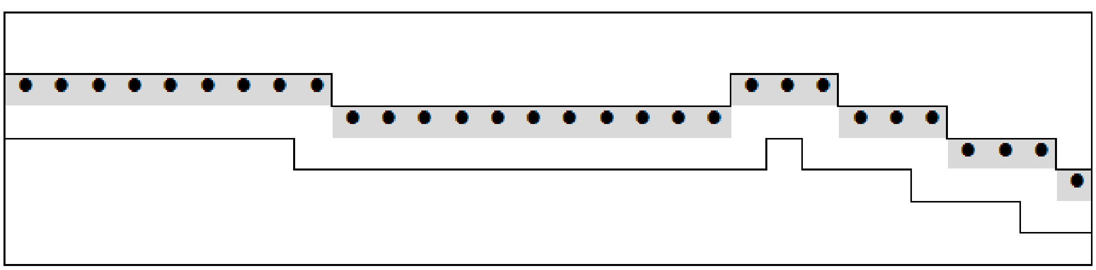


Figure 2-26 Contour Representation in [32]

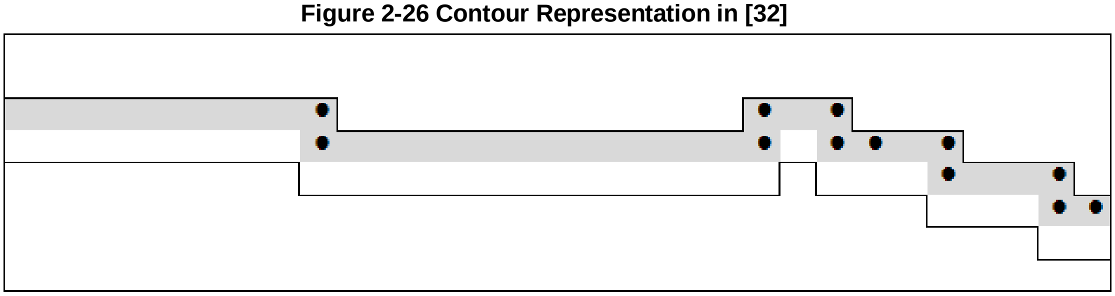


Figure-2-27 Proposed Contour Representation using corners

Finally the reason for locating vertices and the process to locate them was also explained.

Once the algorithm was developed in MATLAB, it was refactored for further implementation as a Simulink block system and a VHDL system. The following chapter shows how the algorithm was implemented as a Simulink block system.

Chapter 3 SIMULINK BLOCK SYSTEM

3.1 Introduction

Once that the algorithm has been designed in MATLAB it is possible to build an input/output level equivalent block system. If the MATLAB algorithm is decomposed into several elements, a block can be built representing each element.

3.2 Corners Extraction

The first required task, as explained before, is the corners edge extraction, so the algorithm in MATLAB was implemented as a block system using Level-2 S Functions in m. Figure 3-1 shows the Simulink block diagram for the corners extraction system.

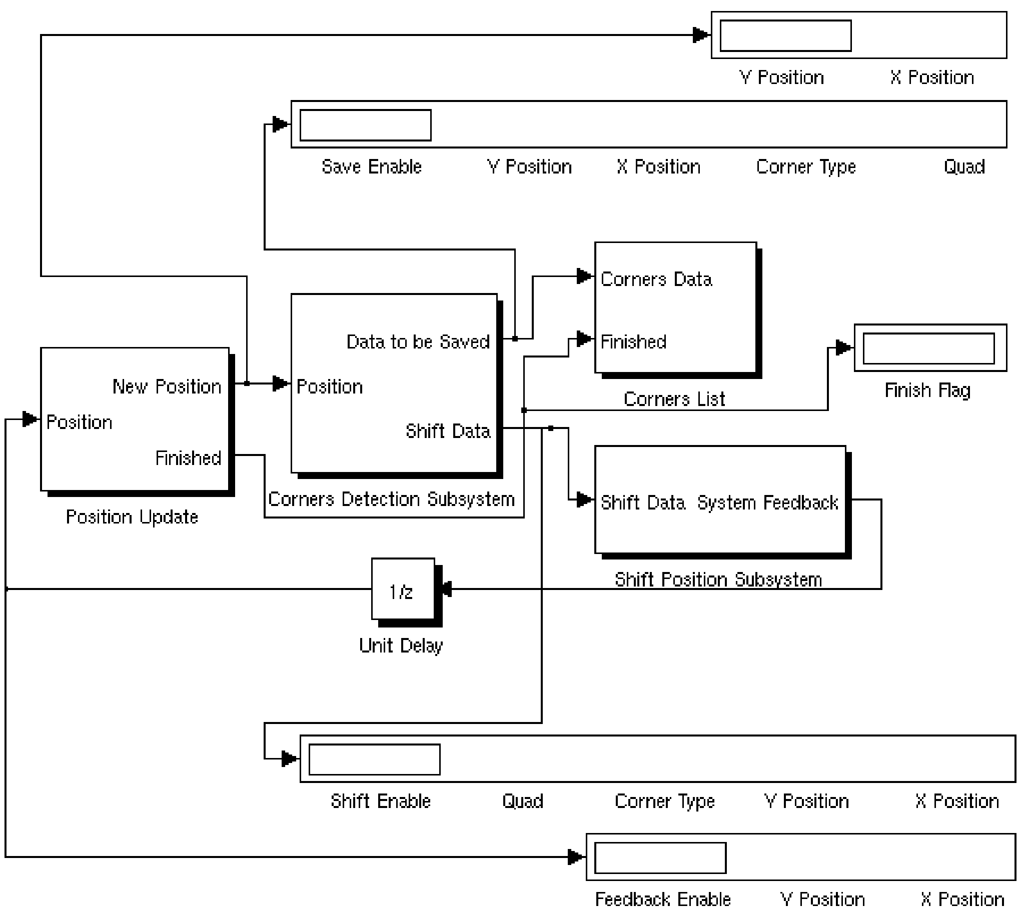


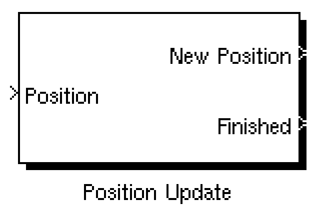
Figure 3-1. Corners Extraction System

From Figure 3-1 can be seen that the general operation of the corners extraction system has four stages:

- First stage - Finds the start pixel at the beginning of the process or updates the position that from which to start next cycle.
- Second stage - Decides if the current pixel is a corner.
- Third stage - If a corner has been detected, save its position in the corners list.
- Fourth stage - Move to a new pixel position in the polygon's edge and decide if the whole edge has been covered, the corners list has been completed or a new cycle needs to start.

The following section gives a more detailed explanation of each block.

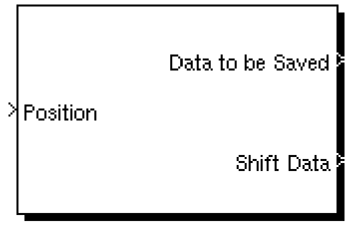
3.2.1 Position Update Block



ELEMENT	DESCRIPTION
Input Parameter	Binary Image
Position[3]	Feedback Enable(1/0), Y Position, X Position
New Position[2]	Y Position, X Position
Finished	Finish Flag (1/0)

At the first system iteration, this block extracts the start pixel's position from the polygon's edge, transfers it to the next block trough the *New Position* port, maintains the *Finished* port in the low start-up value and waits for a *Feedback Enable* high data in the *Position* port for a position update. When the pixel's position coming from the feedback is equal to the start pixel's position, the *Finished* port will turn high and no position update will be passed through the *New Position* port to the next block.

3.2.2 Corners Detection Subsystem

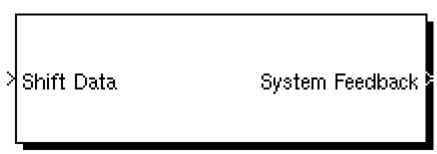


Corners Detection Subsystem

ELEMENT	DESCRIPTION
Input Parameter	Binary Image
Position[2]	Y Position, X Position
Data to be Saved[5]	Save Enable, Y Position, X Position, Corner Type, Quadrant
Shift Data[5]	Shift Enable, Corner Type, Quadrant, Y Position, X Position

This block represents a subsystem that decides if the received pixel's position in the *Position* input port is a corner or not, indentifies its type and quadrant, and then it turns the *Save Enable* to high and sends the required data to be saved to the *Corners List block* through the *Data to be Saved* output port; after that, the block verifies if the next pixel's position represents a corner or not; in the case it does, a feedback is made with the new position and repeats the process until the next pixel's position is not a corner. Finally if the pixel is not a corner, the *Shift Enable* turns high and the required data to find a new corner is sent to the *Shift Position* block.

3.2.3 Shift Position Block

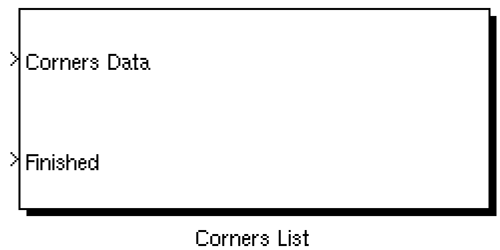


Shift Position Subsystem

ELEMENT	DESCRIPTION
Input Parameter	Binary Image
Shift Data[5]	Shift Enable, Corner Type, Quadrant, Y Position, X Position
System Feedback	Y Position, X Position

This block represents a subsystem that goes over the pixel's edge until a new corner's candidate is found. When a high *Shift Enable* data is received from the Corners Detection Subsystem block, the block evaluates the type of corner and quadrant of the last saved corner, where the next edge's pixel is and repeats the process until a corner's candidate is detected. Then it sends the new pixel's position to the Position Update block for the next process iteration.

3.2.4 Corners List Block



ELEMENT	DESCRIPTION
Input Parameter	Binary Image
Corners Data[5]	Save Enable, Y Position, X Position, Corner Type, Quadrant
Finished	Finish Flag

This block stores the detected corners information as received from the Corners Detection Subsystem block. If a high Save Enable data is received, the block saves the (x y) coordinates of the detected corner, its type and its quadrant. If a high Finish Flag data is received, the block shows the evaluated polygon and plots the saved corners. When the input *Finished* is activated, the whole corner's list is saved to a text file.

3.3 Vertices Location

The vertices location module in MATLAB has the purpose of obtaining a list of vertices of the polygon from the list of corners. Figure 3-2 shows the block diagram for the vertices location subsystem having: *Distances between corners, Distances data, Enable control, Comparison, Vertices location, New index, Vertex selection, Vertices list, and Feedback shift* blocks.

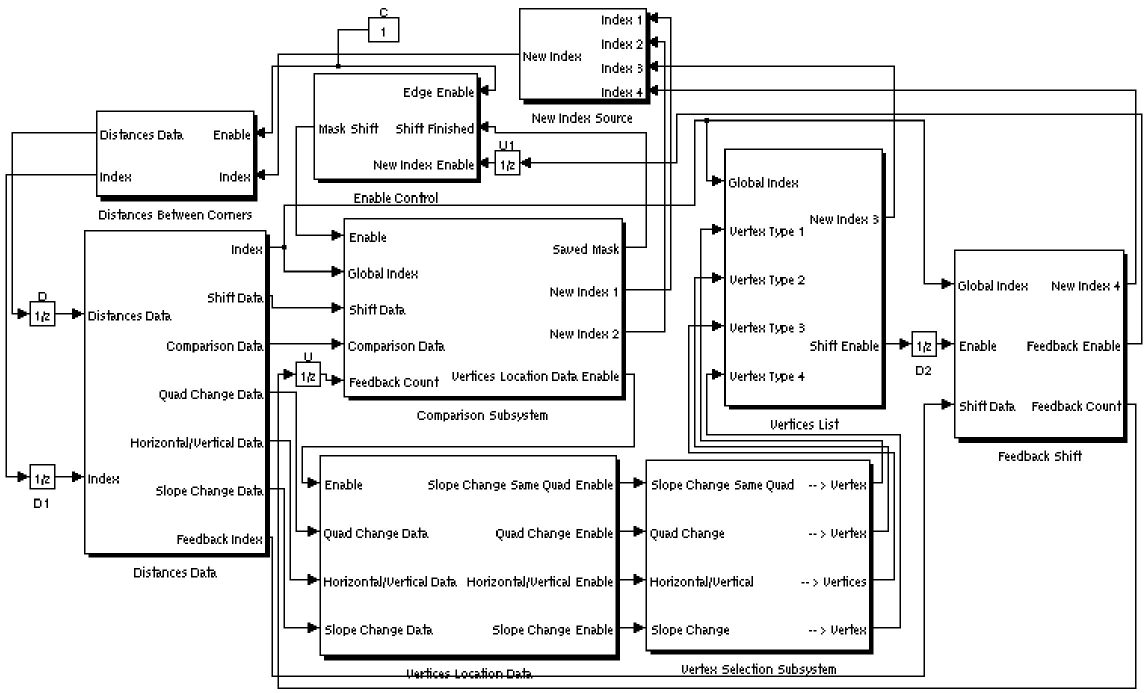
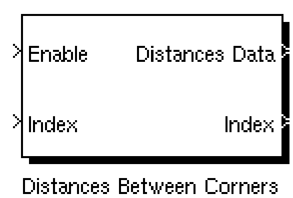


Figure 3-2 Vertices Location System

3.3.1 Distances Between Corners Block

This *Distances Between Corners* block has the purpose of compute the distances between corners from the corners list.

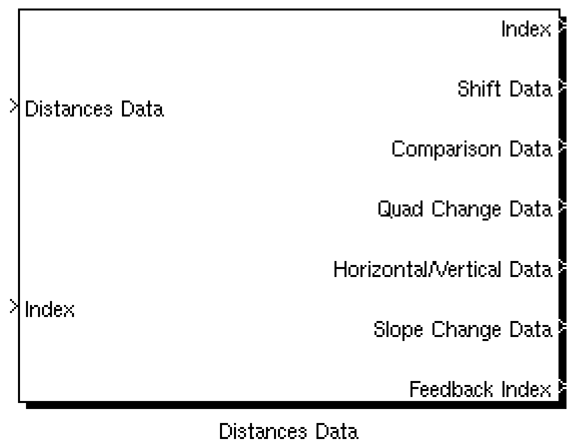


ELEMENT	DESCRIPTION
Enable	System Start Enable
Index	Distances Index input
Distances Data[12]	Y Distance(n-1), X Distance(n-1), Corner Type(n-1), Quadrant(n-1), Y Distance(n), X Distance (n), Corner Type(n), Quadrant(n), Y Distance(n+1), X Distance(n+1), Corner Type(n+1), Quadrant(n+1)
Index	Global Index Output

This block receives the text file generated in the corners extraction system and computes the distances between corners from the corners list. Once the distances between corners list has been generated, the data needed from the Distances Data block is sent by the *Distances Data* output port, later from the second system iteration and on, the data is requested by the Index input port, the new distances data is loaded and sent, and the index is updated and sent through the *Index* output port.

3.3.2 Distances Data Block

The *Distances Data* block has the purpose of manage the received data in a way that all requested data be ready at each output.

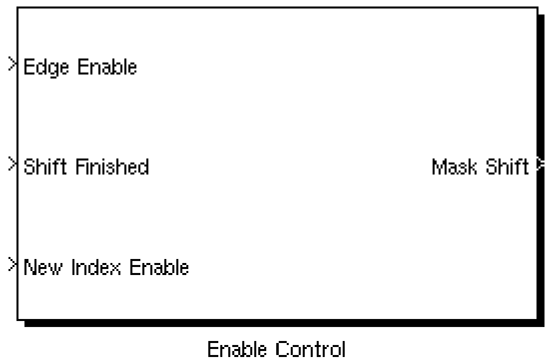


ELEMENT	DESCRIPTION
Distances Data[12]	Y Distance(n-1),X Distance(n-1),Corner Type(n-1),Quadrant(n-1), Y Distance(n),X Distance(n),Corner Type(n),Quadrant(n), Y Distance(n+1),X Distance(n+1),Corner Type(n+1),Quadrant(n+1)
Index	Global Index Input
Index	Global Index Output
Shift Data[4]	Y Distance(n),X Distance(n),Y Distance(n+1),X Distance(n+1)
Comparison Data[2]	Y Distance(n),X Distance(n)
Quad Change Data[7]	Y Distance(n-1),X Distance(n-1), Y Distance(n),X Distance(n), Quadrant(n-1), Quadrant(n), Quadrant(n+1)
Horizontal/Vertical Data[2]	Y Distance(n),X Distance(n)
Slope Change Data	Corner Type
Feedback Index[4]	Y Distance(n),X Distance(n),Y Distance(n+1),X Distance(n+1)

This block receives all the information from the *Distances Between Corners* block through the *Distances Data* and *Index* input ports and distributes it according to how is it required in each output port.

3.3.3 Enable Control Block

The *Enable Control* block has the purpose of coordinate the behavior of the system for a proper operation of its elements.

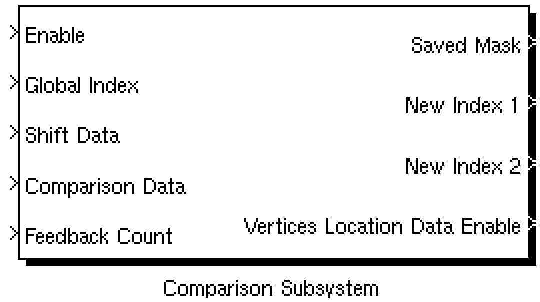


ELEMENT	DESCRIPTION
Edge Enable	System Enable from the Corners Extraction System
Shift Finished	Mask Comparison Flag
New Index Enable	Feedback Enable
Mask Shift	Comparison Enable

When a system high *Edge Enable* data is received from the Corners Extraction System, it indicates that system can start, and then it enables the Comparison Subsystem block through a high *Mask Shift* enable data until high *Shift Finished* flag data is received. When high *New Index Enable* data is received, the block resets all enable outputs, thus setting signals for other system iteration.

3.3.4 Comparison Subsystem

The *Comparison subsystem* has the purpose of find a line pattern, save it and look for a pattern mismatch with the left distances between corners of the list. While finding the line pattern, if a zero-distance between corners is evaluated, a quadrant change vertex is identified, and also if the count of contiguous corners with same distance reaches the maximum, the first mismatch will meant that a slope change in the same quadrant vertex is met.

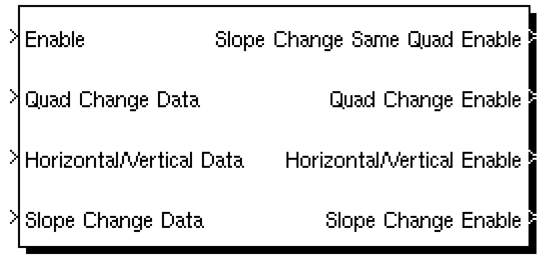


ELEMENT	DESCRIPTION
Input Parameter	Minimum Contiguous Corners with Equal Distance
Enable	Comparison Enable
Global Index	Global Index Input
Shift Data[4]	Y Distance(n),X Distance(n),Y Distance(n+1),X Distance(n+1)
Comparison Data[2]	Y Distance(n),X Distance(n)
Feedback Count	Contiguous Corners with Same Distance Count
Saved Mask	Line Pattern Saved Flag
New Index 1	New Index if comparison not needed
New Index 2	New Index if comparison needed
Vertices Location Data Enable[2]	Enable, Type of Vertex Candidate

In general, this subsystem evaluates distances between corners to find a line pattern, if a zero-distance is found or the contiguous vertices with same distance count is smaller than the input parameter, a line pattern comparison with the subsequent distances between corners is not needed, so the index is updated and sent through the *New Index 1* output port and the *Vertices Location Data Enable* is set to high to locate de vertices candidates just indentified. In the other hand, if none of the previously mentioned conditions are met, the line pattern is saved, the *Saved Mas*k flag is set to high and a comparison is enabled by the *Enable* input port. When a mismatch is found by the pattern comparison, the index is updated and sent through the *New Index 2* output port and the *Vertices Location Data Enable* is set to high.

3.3.5 Vertices Location Block

The *Vertices location block* has the purpose of evaluate which of the vertex candidates is the identified vertex and enables the corresponding output.



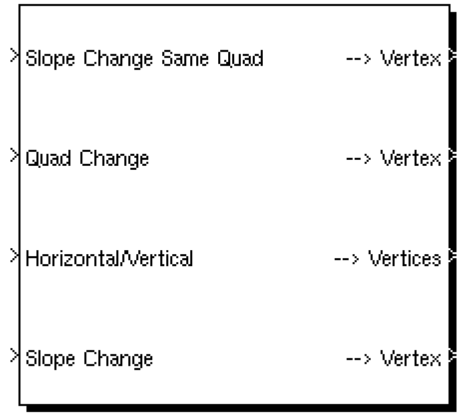
Vertices Location Data

ELEMENT	DESCRIPTION
Enable[2]	Enable, Type of Vertex Candidate
Quad Change Data[7]	Y Distance(n-1),X Distance(n-1), Y Distance(n),X Distance(n), Quadrant(n-1), Quadrant(n), Quadrant(n+1)
Horizontal/Vertical Data	Y Distance(n),X Distance(n)
Slope Change Data	Corner Type
Slope Change Same Quad Enable	Slope Change Same Quadrant Vertex Found
Quad Change Enable	Quadrant Change Vertex Found
Horizontal/Vertical Enable	Horizontal/Vertical Vertices Found
Slope Change Enable	Slope Change Vertex Found

This block decides from the type of vertex candidate information in the *Enable* input port of what kind of vertex has been detected and set to high the corresponding enable output port of the vertex found . Only one output port enable can be set at a time and will be sent to the Vertex Selection Subsystem.

3.3.6 Vertex Selection Subsystem Block

The *Vertex selection subsystem* has the purpose of find the better location of the identified vertex or vertices in the Horizontal/Vertical case.



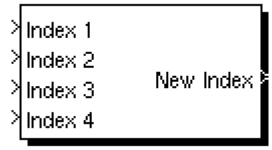
Vertex Selection Subsystem

ELEMENT	DESCRIPTION
Input Parameter	Minimum Length of a Horizontal/Vertical Line
Slope Change Same Quad[2]	Enable
Quad Change[7]	Enable, Y Distance(n-1),X Distance(n-1), Y Distance(n),X Distance(n), Quadrant(n-1), Quadrant(n), Quadrant(n+1)
Horizontal/Vertical[3]	Enable, Y Distance(n),X Distance(n)
Slope Change[2]	Enable, Corner Type
→ Vertex	Slope Change Same Quadrant Vertex
→ Vertex	Quadrant Change Vertex
→ Vertices	Horizontal/Vertical Vertices
→ Vertex	Slope Change Vertex

This subsystem gives the final location of the detected vertex or vertices; this final location is obtained from the enabled input port that matches with the type of vertex indentified and the criteria described in the MATLAB algorithm section.

3.3.7 New Index Source Block

The *New index* block has the purpose of identify the last updated index and send it to the output for new system iteration.



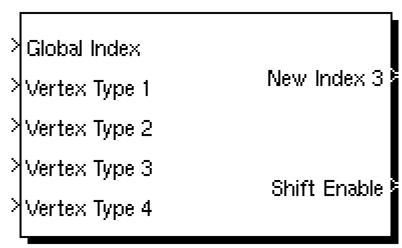
New Index Source

ELEMENT	DESCRIPTION
Index 1	Updated Index
Index 2	Updated Index
Index 3	Updated Index
Index 4	Updated Index
New Index	New Index for new Distances Data

This block manages the indices updates during each stage of the system process and decides which is the last updated index and sends it for a new *Distances Between Corners* block read.

3.3.8 Vertices List Block

The *Vertices list* has the purpose of store the index of each distance between corners identified as a vertex.



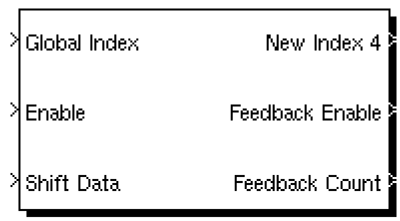
Vertices List

ELEMENT	DESCRIPTION
Global Index	Last Updated Index
Vertex Type 1	Slope Change Same Quad Vertex
Vertex Type 2	Quadrant Change Vertex
Vertex Type 3	Horizontal/Vertical Vertices
Vertex Type 4	Slope Change Vertex
New Index 3	New Updated Index
Shift Enable[2]	Feedback Shift Enable, Type of Shift

This block stores the index of the located vertex, updates the index and sets to high the *Shift Enable* output port and sends the type of shift needed to be performed. When all the distances between corners list has been analyzed, the block loads the corners list text file and plots the (x y) coordinates that match with the saved indexes.

3.3.9 Feedback Shift Block

The *Feedback shift block* has the purpose of shift to a new distance between corners index for a new system iteration.



Feedback Shift

ELEMENT	DESCRIPTION
Input Parameter	Minimum Length of a Horizontal/Vertical Line
Global Index	Last Updated Index
Enable[2]	Feedback Shift Enable, Shift Type
Shift Data[4]	Y Distance(n),X Distance(n),Y Distance(n+1),X Distance(n+1)
New Index 4	New Updated Index
Feedback Enable	Reset Enable Control Outputs
Feedback Count	Contiguous Corners with equal Distance Count

This is the last block in the system and performs a corners' shift to determine the new index in which the system will start a new iteration. There are two types of shifts to be performed and this depends on the Shift Type indicated by the *Enable* input port. The Contiguous Corners with equal Distance Count is sent through the *Feedback Count* output-port to determine if a slope change in the same vertex's quadrant has to be considered on the shift performed. If while a shift operation, a distance longer or equal than the Minimum Length of a Horizontal/Vertical Line input parameter, vertices at the beginning and end of the line need to be located.

3.4 Summary

Once the MATLAB algorithm was designed and explained in the last chapter, a Simulink block system version was implemented. The MATLAB algorithm was decomposed into small elements and from each element a level-2 S function was written in m code so that an equivalent Simulink block could be developed. This chapter explained the function of each Simulink block and how it was built.

The next chapter shows how VHDL entities were built from the decomposed MATLAB algorithm and also explains how external peripherals were interfaced with the FPGA evaluation board.

Chapter 4 VHDL SYSTEM

4.1 Introduction

The VHDL system was implemented concurrently with the Simulink system to ensure system I/O equivalence between the three phases of this work (the MATLAB algorithm, the Simulink block system and the VHDL system). Figure 4-1 shows a simplified diagram of the VHDL system.

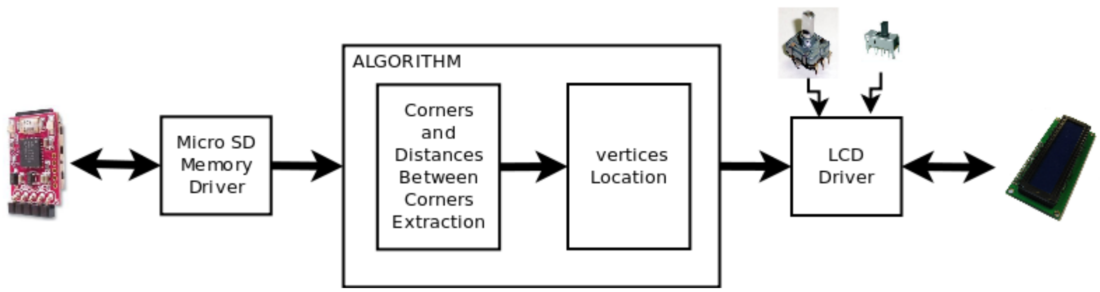


Figure 4-1 VHDL System

From Figure 4-1 we can see that input and output peripherals are added to the Simulink implemented system. The binary image input is loaded into the FPGA from a micro digital secure memory device, and the vertices (x y) coordinates can be viewed in a LCD; with these added peripherals a complete physical system is implemented.

As in the MATLAB algorithm and in the Simulink system, the first task to be performed is the corners extraction from the binary image, so the whole system was also divided into two main parts: the corners and distances between corners extraction and the vertices location.

4.2 Corners and Distances Between Corners Extraction

Figure 4-2 shows a diagram for the corners and distances between corners extraction section of the VHDL system.

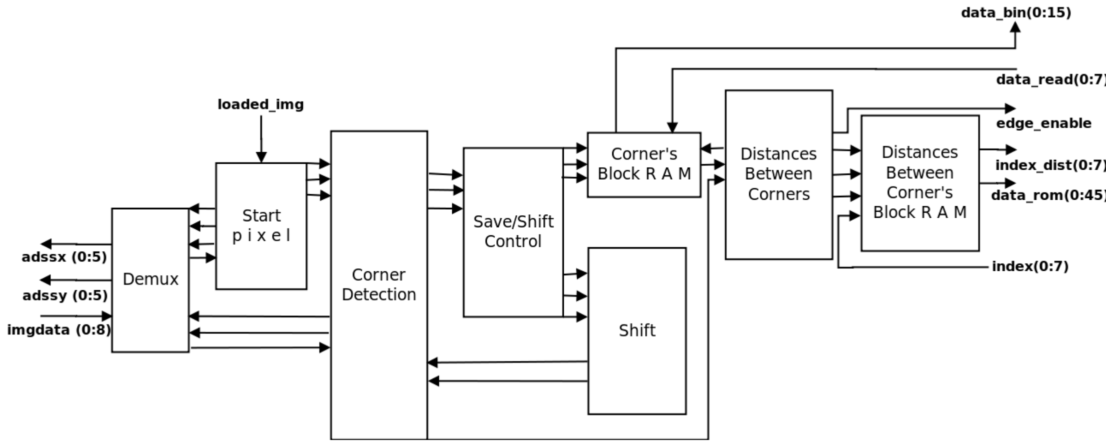


Figure 4-2 Corners and Distances Between Corners Extraction System

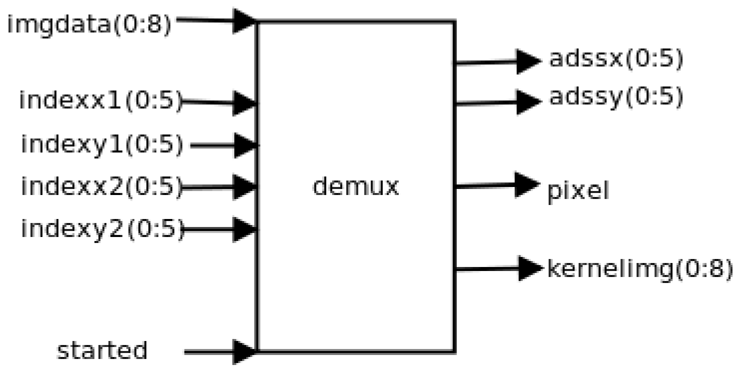
This system operates to extract the information needed from the binary image, as follows:

- First stage. Find the Start Pixel by evaluating image's pixels.
- Second stage. From start pixel, evaluate the image's edge until a candidate-vertex is found.
- Third stage. Define the vertex and its type, save it and do a corners' shift to a new position in the image for a new pixel evaluation.
- Fourth stage. Decide if the new position is the same as the start pixel; if true, enable the distances between corners extraction; else, feedback to a new corner extraction.
- Fifth stage. If corners extraction is finished, extract the distances between corners from the saved corners.

Each element is fully described above.

4.2.1 Demux Entity

The *Demux* entity has the purpose of requesting data from the *Image's Block RAM* and transfers it to the corresponding operating process entity.

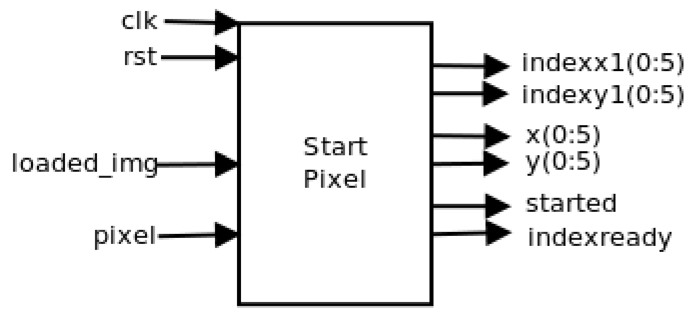


ELEMENT	DESCRIPTION
Imgdata[9]	Values of the requested pixel and its surrounding pixels
Indexx1	X Updated index for start pixel process
Indexy1	Y Updated index for start pixel process
Indexx2	X Updated index for the corner extraction process
Indexy2	Y Updated index for the corner extraction process
started	The start pixel has been found
adssx	X Position of the requested pixel from the loaded binary image
adssy	Y Position of the requested pixel from the loaded binary image
pixel	Requested pixel value for the start pixel process
kernelimg[9]	Requested pixel values for the corners extraction process

This entity serves as a bridge between the *Image's Block RAM* and the current process that requests data from the binary image. The *imgdata* input port receives the value of the requested pixel by the *adssx* and *adssy* position output ports and its eight surrounding pixels. If *started* low data is received, just the current pixel data is sent through the *pixel* output port; else, the whole submatrix pixels values are sent through the *kernelimg* output port.

4.2.2 Start Pixel Entity

The *Start Pixel* entity has the purpose of find the first zero or black valued pixel in the image.

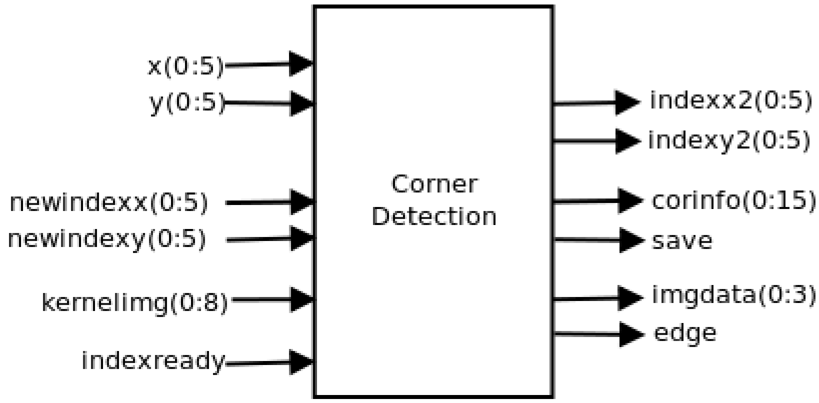


ELEMENT	DESCRIPTION
clk	Clock signal
rst	Reset
loaded_img	The image has been loaded in the block RAM
pixel	Requested pixel value
Indexx1	X Updated index
Indexy1	Y Updated index
x	Start pixel’s X coordinate
y	Start pixel’s Y coordinate
started	Start pixel found
indexready	Start pixel found

This entity performs the first task of the whole system that is to find the start pixel. It evaluates the pixel’s value requested by the *indexx1* and *indexy1* output ports until a zero or black value is found. Once the start pixel is found, the *started* and *indexready* output ports turn high and the start pixel’s (x y) coordinates are sent through the *x* and *y* output ports.

4.2.3 Corner Detection Entity

The *Corner Detection* entity has the purpose of evaluate each polygon’s edge pixel and verify if it is a corner or not, extract the corner data and finally, decide if the whole edge has been computed.

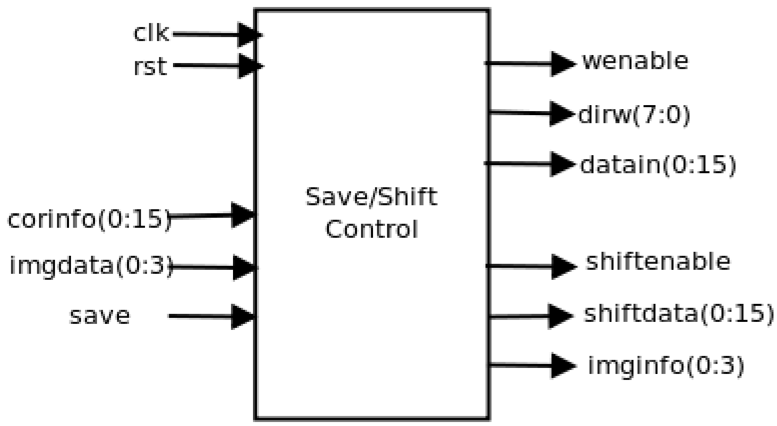


ELEMENT	DESCRIPTION
x	Start pixel's X coordinate
y	Start pixel's Y coordinate
newindexx	New shifted index for new image data request
newindexy	New shifted index for new image data request
kernelimg[9]	Requested pixel values for the corners extraction process
indexready	Start pixel found
indexx2	X Updated index
indexy2	Y Updated index
corinfo[2]	Corner information (type and quadrant)
save	Save detected corner enable
imgdata[4]	Necessary pixel values to define shift direction
edge	Corners extracted from the whole edge flag signal

When the *indexready* turns to high, the entity receives the (x y) coordinates of the start pixel and the *kernelimg* data from the image, then it compares with the corner patterns defined in the algorithm section and decides if the current pixel is a corner or not. If a corner is detected, also its type and quadrant are known, so the information is sent through the *imgdata* output port. Now there is needed to define the direction that the shift has to take from the detected corner; so that the necessary pixels to find that direction are sent through the *imgdata* output port. When a corner detection has been performed, the entity waits for the shifted position of the new image's data request from the *newindexx* and *newindexy* input ports and verifies if that position matches with the start pixel's position, so this way we can find if the whole edge has been covered and the *edge* output turns to high for the *Distances Between Corners* entity.

4.2.4 Save/Shift Control Entity

The *Save/Shift Control* entity has the purpose of deciding if the data just analyzed by the *Corner Detection* entity needs to be saved and shifted or just shifted.

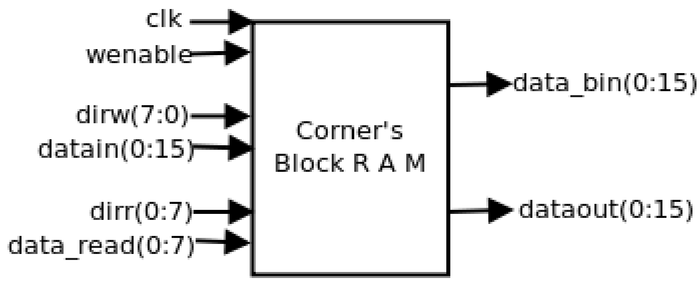


ELEMENT	**DESCRIPTION**
clk	Clock Signal
rst	Reset
corinfo[2]	Corner information (type and quadrant)
imgdata[4]	Necessary pixel values to define shift direction
save	Save detected corner enable
wenable	Enables the Corner's Block RAM
dirw	Corner's Block RAM index
datain[4]	Data to be saved(corner's x coordinate, corner's y coordinate, corner type, quadrant)
shiftenable	Enables the Shift entity
shiftdata[4]	Data from the last saved corner to define the shift direction
imginfo[4]	Necessary pixel values to define shift direction

If the *save* input port is high, the data received in the *wenable* output port turns high, the *Corner's Block RAM* index is updated, sent through the *dirw* output port and the data received in the *imgdata* input port is sent to the *Corner's Block RAM* through the *datain* output port. The shift entity is enabled by turning to high the *shiftenable* output port and the required data is sent through the *shiftdata* and *imginfo* output ports.

4.2.5 Corner's Block RAM

The *Corner's Block RAM* has the purpose of store all the corners extracted from the polygon's edge.

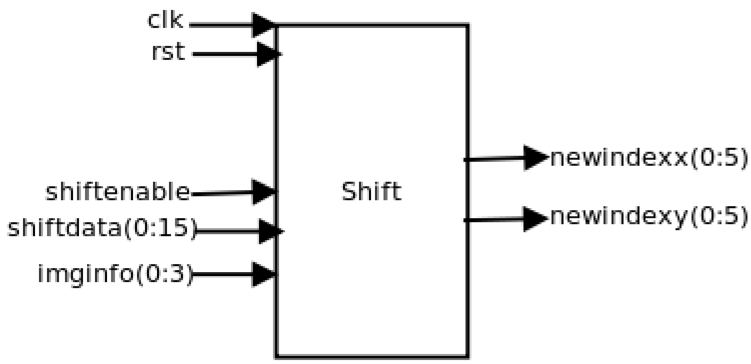


ELEMENT	DESCRIPTION
clk	Clock Signal
wenable	Enables the Corner's Block RAM
dirw	Corner's Block RAM index
datain[4]	Data to be saved(corner's x coordinate, corner's y coordinate, corner type, quadrant)
dirr	Read index from the distances between corners extraction process
data_read	Read index from the LCD driver
data_bin	Data requested from the LCD driver
dataout	Data requested from the distances between corners extraction process

This is the block RAM where the corners (x y) coordinates, type of each corner and the quadrant location of the corner is saved. Data received in the *datain* input port is only saved in the *dirw* received index if the *wenable* input port is high. Data request indexes are received through the *dirr* and *data_read* input ports and sent to the *Distances Between Corners* entity and *LCD Driver* trough the *dataout* and *data_bin* output ports.

4.2.6 Shift Entity

The *Shift entity* has the purpose of shift to a new position in polygon's edge where a corner may be extracted in new system iteration.

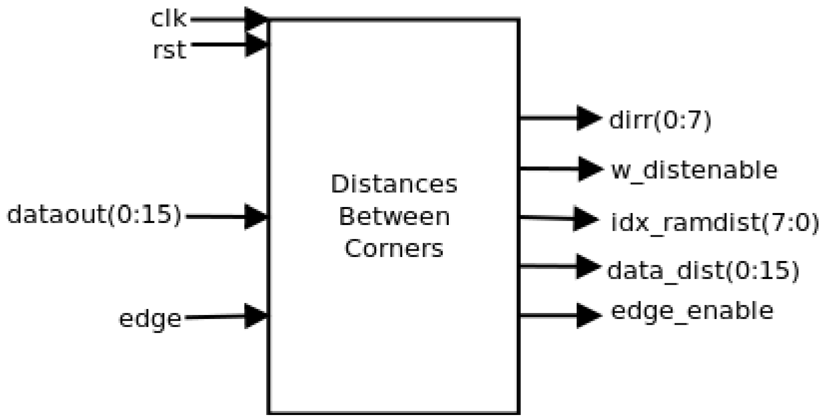


ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
shiftenable	Enables the Shift entity
shiftdata[4]	Data from the last saved corner to define the shift direction
imginfo[4]	Necessary pixel values to define shift direction
newindexx	New shifted index for new image data request
newindexy	New shifted index for new image data request

When the *shiftenable* input port is high, the entity updates the index of the new data that is going to be requested to the *Image's Block RAM*, this decision is taken from the data of the last saved corner received in the *shiftdata* input port and the surrounding pixel values of interest received in the *imginfo* input port. The updated index is sent to the *Corner Detection* entity through the *newindexx* and *newindexy* output ports.

4.2.7 Distances Between Corners Entity

The *Distances Between Corners* entity has the purpose of extract the distances between the stored corners by computing the absolute difference between contiguous corners of the same type.

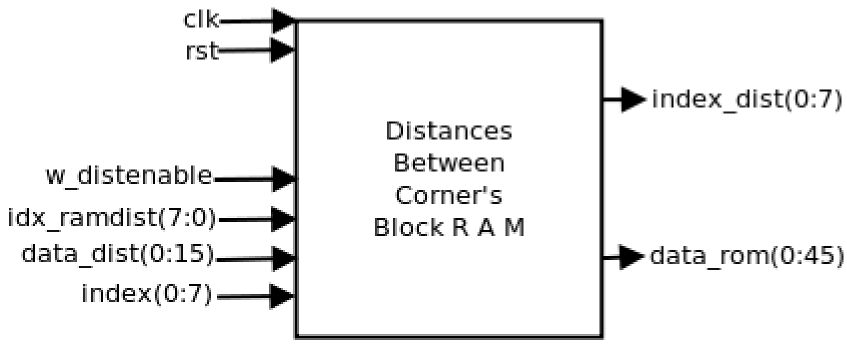


ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
dataout	Data request for the corner's block RAM
edge	Corners extracted from the whole edge flag signal
dirr	Read index for the corner's block RAM
w_distenable	Distances between corners block RAM save enable
Idx_ramdist	Distances between corners block RAM index
data_dist	Data to be saved in the distances between corners block RAM
edge_enable	All the operations from this system section have been performed

Once that all the edge corners have been extracted, the *edge* input port of this entity is turned to high, so that the distances between corners can now be extracted from the *Corner's Block RAM* saved data. This entity performs an absolute difference between the contiguous corners of the same type and then the result is sent to the *Distances Between Corners Block RAM* through the *idx_ramdist* and *data_dist* output ports to be saved. Once all the distances between corners have been extracted, the edge_enable output port is turned to high to signal that all the information needed from the image has been saved.

4.2.8 Distances Between Corners Block RAM

The *Distances Between Corners Block RAM* has the purpose of store all the absolute differences computed in the *Distances Between Corners* entity .



ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
w_distenable	Distances between corners block RAM save enable
idx_ramdist	Distances between corners block RAM index
data_dist	Data to be saved in the distances between corners block RAM
index	Read index from the vertices location section
index_dist	Last read index from the vertices location section
data_rom	Requested data from the vertices location section
edge_enable	All the operations from this system section have been performed

When the *w_enable* input port is high, the *Distances Between Corners Block RAM* saves the data received at the *data_dist* input port into the block RAM index received from the *idx_ramdist* input port. Read data can be requested through the *index* input port and sent to the vertices location section trough the *index_dist* and *data_rom* output ports.

4.3 Vertices Location

Figure 4-3 shows a diagram for the Vertices Location section of the VHDL system.

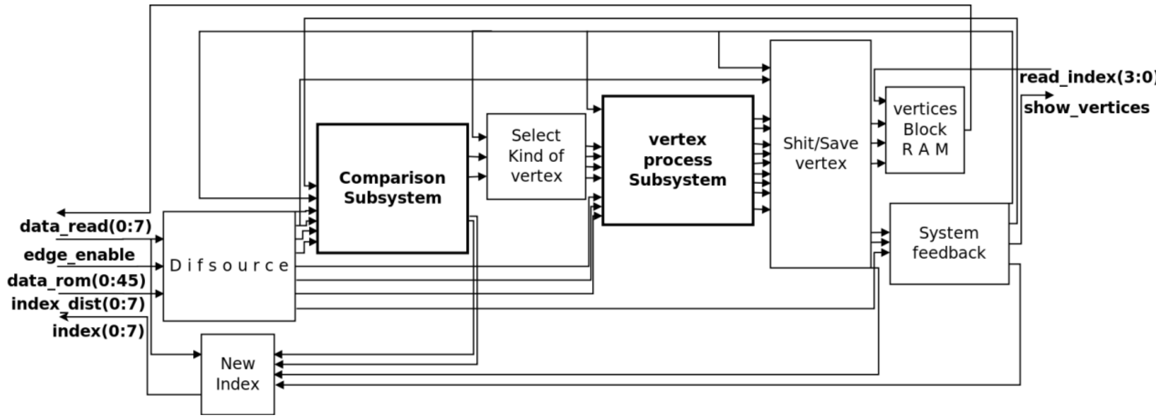


Figure 4-3 Vertices Location System

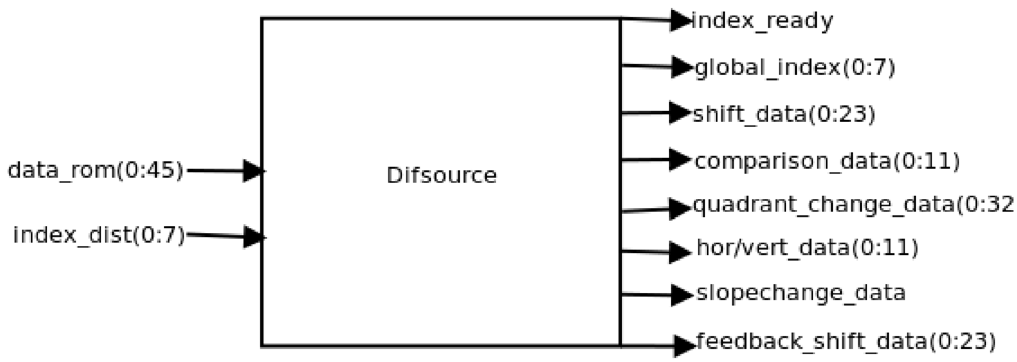
This section of the VHDL system operates to locate vertices from the Distances Between Corners list, as follows:

- First Stage. From the distances between corners list, find a line pattern for later comparison and identify a slope change in the same quadrant. To find the pattern, evaluate each distance, so in the process to identify: a quadrant change, a horizontal/vertical line and a one-pattern slope change in the same quadrant for candidate vertices.
- Second Stage. At this stage, the candidate vertices are defined according to the information extracted from the distances between corners and input parameters.
- Third Stage. A better vertex location is established by considering: corner type, quadrant and the criteria explained before in the MATLAB algorithm section.
- Fourth Stage. Save vertex or vertices indexes and the new list of distances between corners' index is traversed for a new vertex location process.

Each element is fully described below.

4.3.1 Difsource Entity

The *Difsource* entity has the purpose of request data from the *Distances Between Corners Block RAM* and transfer to the outputs as required from the process performed inside the system.

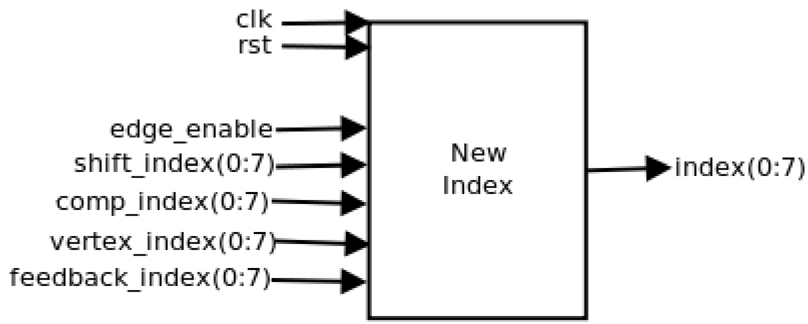


ELEMENT	DESCRIPTION
data_rom[12]	Requested data from the distances between corners block RAM
Index_dist	Updated index of data from the distances between corners block RAM
Index_ready	Index ready for a new system iteration flag
global_index	Last updated index
shift_data[4]	Necessary data for the comparison traversing
comparison_data[2]	Necessary data for the comparison process
quadrant_chage_data[7]	Necessary data to evaluate if the current distance between corners is a quadrant change vertex candidate
hor/vert_data[2]	Necessary data to evaluate if the current distance between corners is a horizontal/vertical line vertex candidate
slopechange_data	Necessary data to evaluate if the current distance between corners is a slope change in the same quadrant vertex candidate
feedback_shift_data[4]	Necessary data to find the new distances between corners index for a new system iteration

This entity receives all the needed data from the *Distances Between Corners Block RAM* in the *data_rom* input port and manage it in a way that all required data be ready at every request from any process within the system. The *index_ready* output port indicates that an updated index has been received in the *index_dist* input port and new system iteration is in course.

4.3.2 New Index Entity

The *New index* entity has the purpose to identify the last updated index and send it to the output for a new data request.

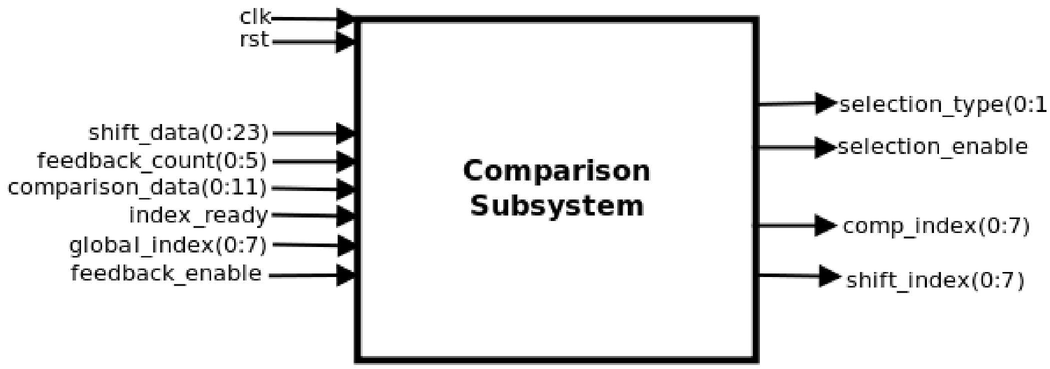


ELEMENT	DESCRIPTION
clk	Clock signal
rst	Reset
edge_enable	Vertices Location section enable
shift_index	Last updated index from the traversing process
comp_index	Last updated index from the comparison process
vertex_index	Last updated index from the shift/save process
feedback_index	Last updated index from feedback
index	Index to request data from the distances between corners block RAM

This entity updates the index that will be sent to the distances *Between Corners Block RAM* through the *index* output port for new data request. The most updated received index depends on the last performed process within the system.

4.3.3 Comparison Subsystem

The C*omparison Subsystem* has the purpose to evaluate distances between corners to extract a line pattern and compare with the remaining distances between corners until a mismatch occurs and a slope change in the same quadrant is identified. While extracting the line pattern, a quadrant change or horizontal/vertical line can also be identified by considering a zero distance between corners either in x or y axes.



ELEMENT	DESCRIPTION
Input Parameter	Minimum Contiguous Corners with equal Distance
clk	Clock signal
rst	Reset
shift_data[4]	Necessary data for the comparison shift
feedback_count	The count of contiguous corners with same distance that comes from the feedback
comparison_data[2]	Necessary data for the comparison process
index_ready	Index ready for a new system iteration flag
global_index	Last updated index
feedback_enable	Feedback flag that gives advice that a new iteration is in course so some signals and variables need to return to initial values
selection_type	Necessary information from the comparison to find a vertex candidate
selection_enable	Enable to select the vertex candidate from the *selection_type* information
comp_index	Comparison process index
shift_index	Shift index

For a better understanding, Figure 4-4 shows a diagram of the entities within the *Comparison Subsystem.*

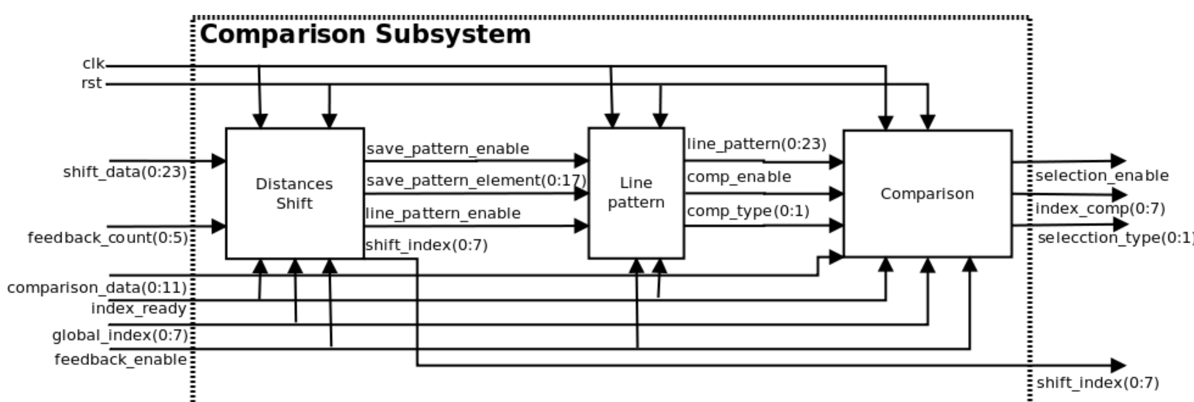


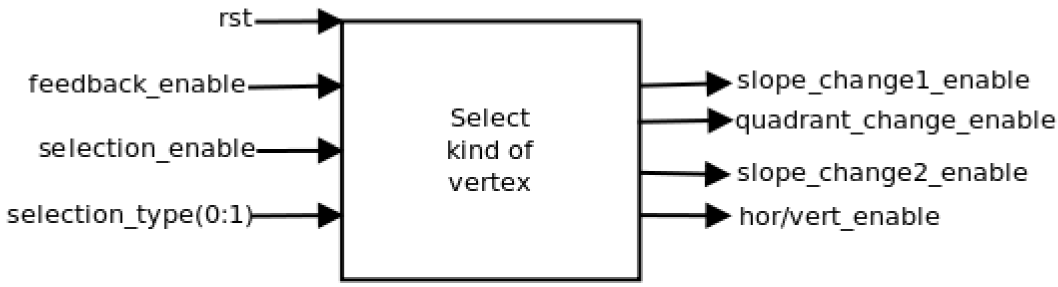
Figure 4-4 Comparison Subsystem

At first the *Distances Shift* entity has the purpose to identify a two-element line pattern, so it requests distance between corners(n) and distance between corners(n+1) from the *Difsource* entity and compare them until a mismatch occurs. If the count of equal compared distances is smaller than the *Minimum Contiguous Corners with equal Distance* input parameter, the line parameter is saved in the *Line Pattern* entity and a new parameter is element is extracted in the same way, else the *Line Pattern* entity is directed to save just one line parameter element. On the other hand, if while comparing distances, a zero distance is found either in the x or y axes, or the current evaluated distance between corners is part of a quadrant change or horizontal/vertical line so no more distances comparison will be needed and the current data will be sent to the *Line Pattern* entity. The *Line Pattern* Entity decides from the received data if a comparison needs to be made or not and send saved data to the *Comparison* entity. If a comparison is needed, the *Comparison* entity requests data from the *Difsource* entity through the *comparison_data* input port and compares it with the saved line pattern until a mismatch occurs, else it just receives data from the Line Pattern entity and establishes three possible types of vertex to be located:

- Type 1. Quadrant change, horizontal/vertical line or slope change in the same quadrant if a comparison process was performed.
- Type 2. A slope change in the same quadrant if the count reached the *Minimum Contiguous Corners with equal Distance* input parameter.
- Type 3. A quadrant change or horizontal/vertical line if a zero distance was found.

4.3.4 Select Kind of Vertex Entity

The *Select Kind of Vertex* entity has the purpose of enable the candidate vertices outputs by analyzing the selection type data received from the *Comparison Subsystem*.

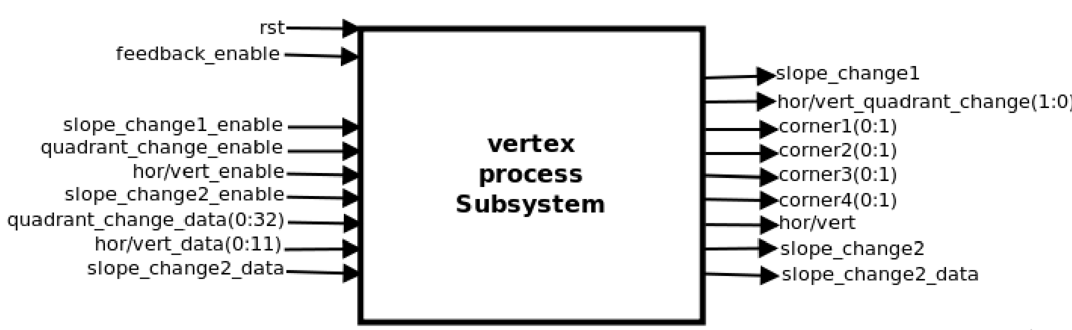


ELEMENT	DESCRIPTION
rst	Reset
feedback_enable	Feedback flag that gives advice that a new iteration is in course so some signals and variables need to return to initial values
selection_enable	Enable to select the vertex candidate from the *selection_type* information
selection_type	Necessary information from the comparison to find a vertex candidate
slope_change1_enable	Candidate vertex of a slope change in the same quadrant with one line pattern element enable
quadrant_change_enable	Candidate vertex of a quadrant change enable
slope_change_enable	Candidate vertex of a slope change in the same quadrant with two line pattern elements enable
hor/vert_enable	Candidate vertices of a horizontal/vertical line enable

When the *slection_enable* input port is high, the *Select Kind of Vertex* entity enables the corresponding candidate vertices output ports indicated by the type received in the *selection_type* input port.

4.3.5 Vertex Process Subsystem

The *Vertex Process Subsystem* has the purpose to define the vertex from the vertex candidates.



ELEMENT	DESCRIPTION
Input Parameter	Minimum Length of a Horizontal/Vertical Line

rst	Reset
feedback_enable	Feedback flag that gives advice that a new iteration is in course so some signals and variables need to return to initial values
slope_change1_enable	Candidate vertex of a slope change in the same quadrant with one line pattern element enable
quadrant_change_enable	Candidate vertex of a quadrant change enable
hor/vert_enable	Candidate vertices of a horizontal/vertical line enable
slope_change_enable	Candidate vertex of a slope change in the same quadrant with two line pattern elements enable
quadrant_change_data[7]	Necessary data to evaluate if the current distance between corners is a quadrant change vertex candidate
hor/vert_data[2]	Necessary data to evaluate if the current distance between corners is a horizontal/vertical line vertex candidate
slope_change_data	Necessary data to evaluate if the current distance between corners is a slope change in the same quadrant vertex candidate
slope_change1	Identified vertex in a slope change in the same quadrant with one line parameter element
hor/vert_quadrant_change	Identified vertices of a horizontal/vertical line from a quadrant change
corner1	Information to find the better location of a quadrant change vertex
corner2	Information to find the better location of a quadrant change vertex
corner3	Information to find the better location of a quadrant change vertex
corner4	Information to find the better location of a quadrant change vertex
hor/vert	Identified vertices of a horizontal/vertical line
slope_change	Identified vertex in a slope change in the same quadrant with two line parameter elements
slope_change_data	Information to find the better location of a slope change in the same quadrant vertex with two line parameter elements

For a better understanding, Figure 4-5 shows a diagram of the entities within the *Vertex Process Subsystem.*

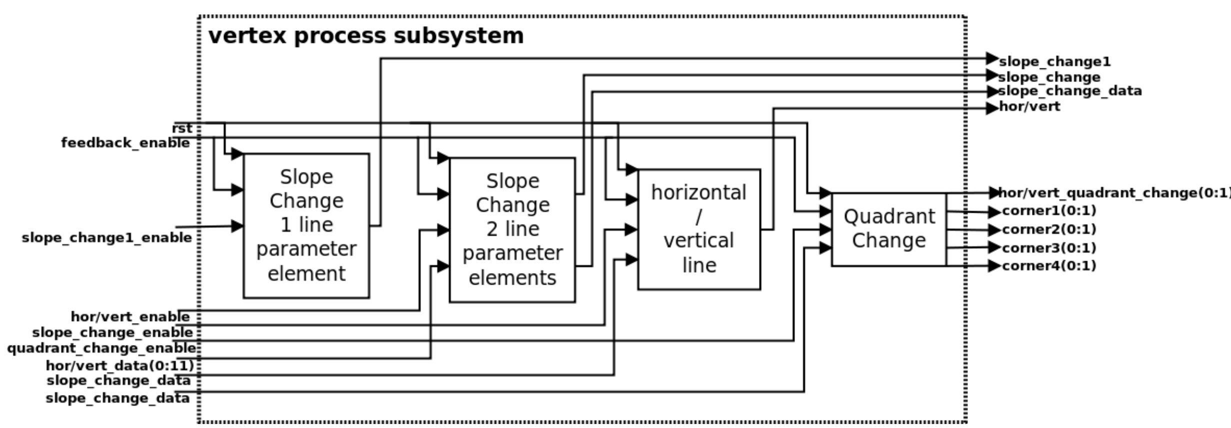
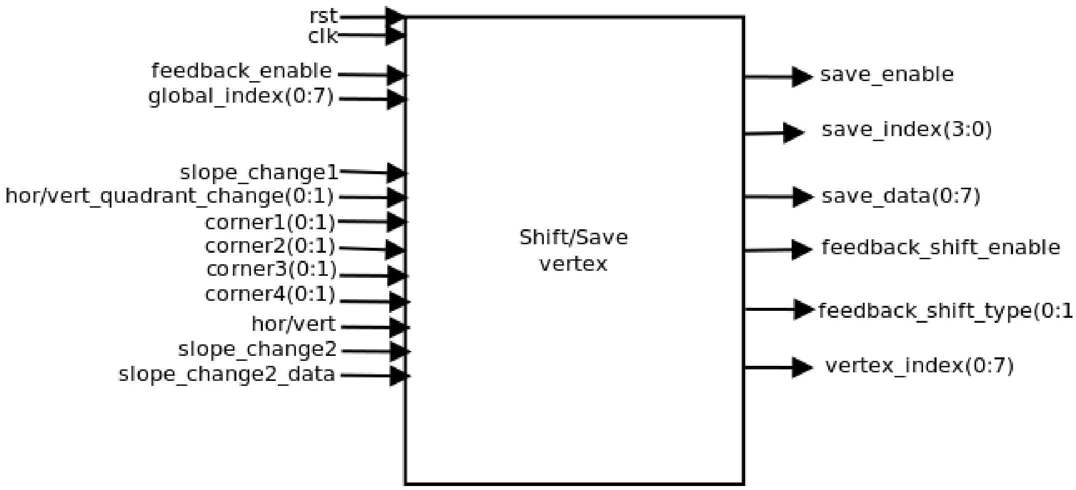


Figure 4-5 Vertex Process Subsystem

This subsystem evaluates data received from candidate vertex enabled input ports (*slope_change1_enable, hor/vert_enable, slope_change_enable* and *quadrant_change_enable*) and decides which of the candidate vertex succedes with the condition to become a vertex, so then it enables the kind of vertex candidate output port (*slope_change1, slope:change, slope_change_data*, *hor/vert and hor/vert_quadrant_change*). If the *Slope Change 1 Line Parameter Element* entity is enabled, the *slope_change1* output port will be set to high. If the *Slope Change 2 Line Parameter Elements* entity is enabled, the *slope_change* output port will be set to high and the *slope_change_data* will be sent to the *Shift/Save Vertex* entity to find the better location of the vertex. If the Horizontal/Vertical entity line is enabled, it will verify if the current evaluating distance between corner in x or y axis is longer than the *Minimum Length of a Horizontal/Vertical Line* input parameter the *hor/vert* output port will be set to high. If the *Quadrant Change* entity is enabled, it will verify if the current evaluating distance between corner in x or y axis is longer than the *Minimum Length of a Horizontal/Vertical Line* input parameter to identify horizontal/vertical line vertices in a quadrant change, else from the quadrant in which a zero distance between corners was found a *corner"n"* output port will be set to high.

4.3.6 Shift/Save Entity

The Shift/Save entity has the purpose to decide which of the vertices is saved and if required, traverse the list for a better location. It also decides what index traverse is needed for the new system iteration.

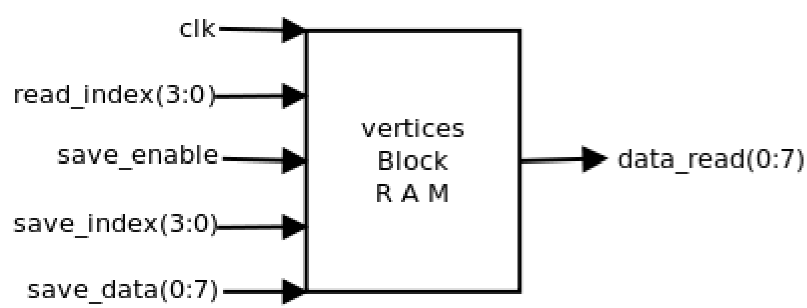


ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
feedback_enable	Feedback flag that gives advice that a new iteration is in course so some signals and variables need to return to initial values
global_index	Last updated index
slope_change1	Identified vertex in a slope change in the same quadrant with one line parameter element
hor/vert_quadrant_change	Identified vertices of a horizontal/vertical line from a quadrant change
corner1	Information to find the better location of a quadrant change vertex
corner2	Information to find the better location of a quadrant change vertex
corner3	Information to find the better location of a quadrant change vertex
corner4	Information to find the better location of a quadrant change vertex
hor/vert	Identified vertices of a horizontal/vertical line
slope_change	Identified vertex in a slope change in the same quadrant with two line parameter elements
slope_change_data	Necessary data to evaluate if the current distance between corners is a slope change in the same quadrant vertex candidate
save_enable	Save located vertex index enable
save_index	Vertices block RAM index
save_data	Located vertex index to be saved
feedback_shift_enable	Feedback shift enable
feedback_shift_type	Type of feedback shift to perform
vertex_index	Updated index

If the *slope_change1* input port is high, the index of the vertex identified by a slope change in the same quadrant with one line pattern element will be sent to the *Vertices Block RAM*. If either of the *hor/vert* or the *hor/vert_quadrant_change* input port is high, the beginning and end of the line vertex indexes will be sent to the *Vertices Block RAM*. Finally if the slope_change or any of the corner"n" input ports is high, a better location decision will be performed to find the resulting index to be sent to the Vertices Block RAM. This entity also decides the type of traversing that the System Feedback needs to perform for new system iteration.

4.3.7 Vertices Block RAM

The Vertices Block RAM has the purpose of store the identified index of vertices.

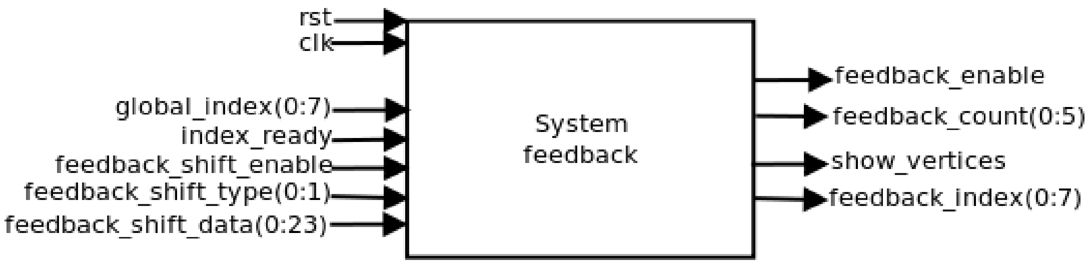


ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
read_index	Block RAM index for data read
save_enable	Save located vertex index enable
save_index	Vertices block RAM index
save_data	Located vertex index to be saved
data_read	Requested data read

When the *save_enable* input port is high, the *Vertices Block RAM* saves the data received at the *save_data* input port into the block RAM index received from the *save_index* input port. Read data can be requested through the *read_index* input port and sent to the vertices location section through the *data_read* output port.

4.3.8 System Feedback Entity

The *System Feedback* entity has the purpose to traverse the index of the last evaluated distance between corners and provide a new index for next system iteration.



ELEMENT	DESCRIPTION
clk	Clock Signal
rst	Reset
global_index	Last updated index
index_ready	Index ready for a new system iteration flag
feedback_shift_enable	Feedback shift enable
feedback_shift_type	Type of feedback shift to perform
feedback_shift_data[4]	Necessary data to find the new distances between corners index for a new system iteration
feedback_enable	Feedback flag that gives advice that a new iteration is in course so some signals and variables need to return to initial values
feedback_count	Contiguous corners with same distance count
show_vertices	This flag indicates that all the possible vertices have been extracted from the distances between corners
feedback_index	Updated index

When a vertex or vertices have been located, in some cases is necessary to traverse the distances between corners to assure that the new index belongs to a corner that has possibilities to become a vertex, so depending on the last saved vertex index, the *feedback_shift_type* and *feedback_shift_data* will provide information about the traverse to be performed. While performing a traverse, we maintain the number of contiguous corners with the same distance. Now if a change in slope is identified, this count is sent through the *feedback_count* output port. The *feedback_enable* informs that a new iteration is in course so some signals and variables need to return to initial values and be prepared for new system iteration. When all the polygon's vertices are located, the *show_vertices* output port is set to high to enable the *LCD driver.*

4.4 External Peripherals

To build a complete physical system, there is a need for an input medium to load the image's pixel data into the FPGA and an output medium to visualize which vertices were obtained by the system from the image. The FPGA loads an image from a micro Secure Digital memory card and then it shows vertices (x y) coordinates in an alphanumeric LCD. Both peripherals implementation will be explained below.

4.4.1 Micro SD Memory Driver

This *Micro SD Memory Driver* has the purpose to establish communication between the micro SD memory card and the FPGA through the microdrive for read/write operations.

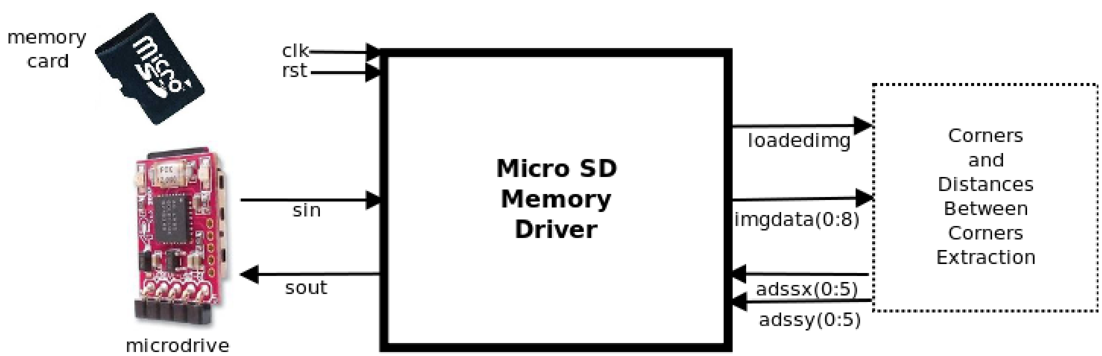


Figure 4-6 Micro SD memory driver

ELEMENT	DESCRIPTION
clk	Clock signal
rst	Reset
sin	Serial input
sout	Serial output
loaded_img	Indicates that the image has been loaded into the Image Block RAM
Imgdata[9]	Values of the requested pixel and its surrounding pixels
adssx	X Position of the requested pixel from the loaded binary image
adssy	Y Position of the requested pixel from the loaded binary image

Figure 4-7 shows a diagram of the entities within the *Micro SD Memory Driver*

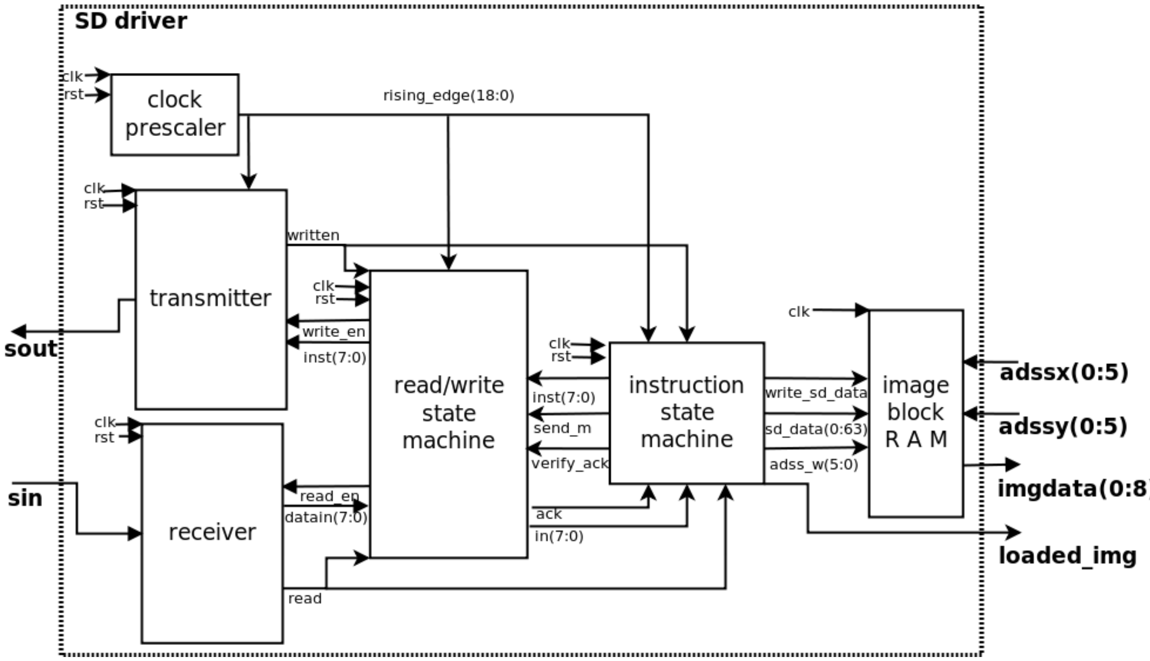


Figure 4-7 Internal component entities for Micro SD memory driver

The microdrive is a device that integrates a microcontroller that makes possible to access the memory card by standard commands that provide direct access for storing and retrieving data through a simple serial interface; such that, the microdrive provides both a physical and a software interface.

- The physical interface is provided by serial Reciver/Transmitter, this way the FPGA communicates with the microdrive to send commands and save/read data.
- The software interface provided by the microdrive is a set of easy to use serial commands.

The microdrive has an auto-baud feature which can automatically detect the host clock speed and set its internal baud rate to operate from 300 to 256K baud so a C*lock Prescaler* entity is needed to adapt the FPGA clock as required by the microdrive. Prior to any commands being sent to the microdrive, it must first be initialized by sending an auto-baud command (0x55) after any power-up or reset, this will allow determining and locking onto the baud rate of the *clock prescaler* signal. Once the microdrive has locked onto the baud rate, it will respond with an acknowledge byte (06hex) and the micro SD memory card can now be initialized. The micro SD memory card initialize command is used when the memory card is inserted after power-up or reset, else if at power-up or reset cycle the memory card is already inserted, there is no needed to send the micro SD memory card initialize command. This command is two bytes long (0x40 and 0x69).

The FAT32 file system is used for the micro SD memory card; this ensures that the image written by the FPGA system can be readable in a computer from the micro SD memory card of any standard image file format. To retrieve the image from the micro SD memory card: first point at the file system sector in which the image is saved to request data bytes. The microdrive software interface has a command that sets memory address pointer for byte wise reads and writes; after each byte read or write, the *memory Address pointer* is automatically incremented to the next address location. The *set memory address* command is eight bytes long (0x40, 0x44, address Umsb, address Ulsb, address Lmsb and address Llmsb). Once the *set memory address* command is sent, a read byte data can be sent and the microdrive will return a byte of data from the pointing memory location. The read byte data command is two bytes long (0x40 and 0x72).

The *Instruction State Machine* entity outputs the corresponding sequence to be sent to the microdrive for each command.

1. Auto-baud command.
2. Micro SD card initialize memory command.
3. Set memory address command.
4. Read byte data command.

The *Read/Write State Machine* entity controls when does data is received and when it is sent. If a command is sent, the *Read/Write State Machine* entity waits for acknowledge before another command is sent or data is received by the *Transmitter/Receiver* entities. When image data is received, it is stored in the *Image Block RAM*.

4.4.2 LCD Driver

The *LCD Driver* has the purpose to interface the LCD device with the FPGA and send the resulting data of the system process to be displayed by the LCD.

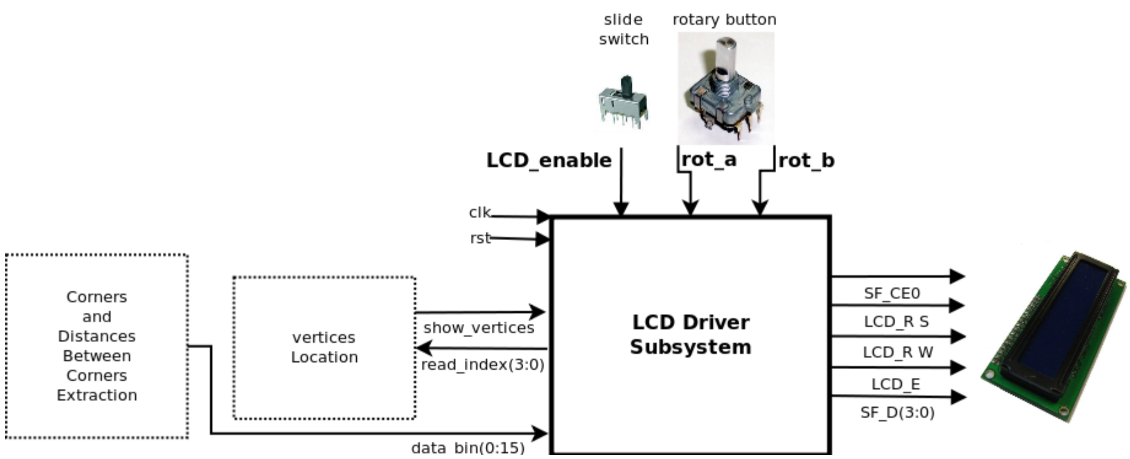


Figure 4-8 LCD Driver Subsystem

ELEMENT	DESCRIPTION
clk	Clock signal
rst	Reset
show_vertices	Signal that indicates that vertices are ready to be displayed
read_index	Vertices Block RAM read index
data_bin[2]	Vertex (x y)coordinates in binary
SF_CE0	Starter Kit Strata Flash Chip Enable (data signal shared between the LCD and StrataFlash)
LCD_RS	LCD Register select
LCD_RW	LCD Read/Write control
LCD_E	LCD Read/Write enable
SF_D[4]	LCD Data bits(0:3)

Figure 4-9 shows a diagram of the entities within the *LCD Driver*

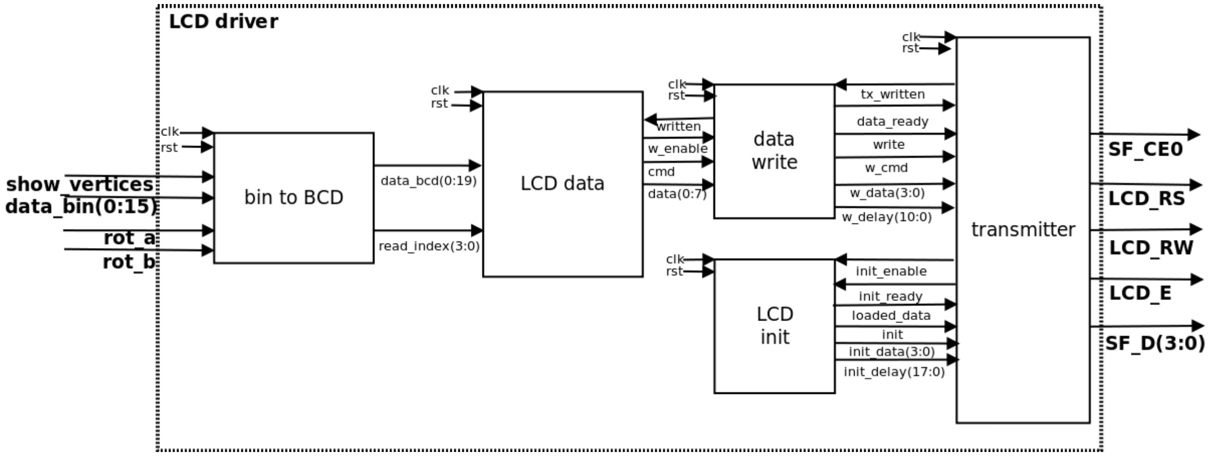


Figure 4-9 LCD Driver

Once that all vertices from the input image are located, the *LCD Driver* is enabled and resulting the vertices (x y) coordinates are visualized on the LCD.

When the show_vertices input is high, the LCD driver requests a vertex (x y) coordinate, then it makes a conversion between the received binary data to BCD code and is sent to the *LCD data* entity. The LCD data entity receives the BCD data and sends it to the LCD in the proper sequence and format to be displayed. The LCD is first initialized through the *LCD Init entity,* this entity has all the initialization sequence and runs just once at power-up. While in the Starter Kit, the FPGA controls the LCD via a 4-bit data interface, the *Data Write* entity separates the data to be displayed into two nibbles. The *Transmitter* entity manages time and delays of LCD signals and sends data to be displayed. The Switch Slide button enables/disables image and the Rotary Button sends the signal to a new vertex display process, so just one vertex is displayed at the time, so the rotary Button controls the vertices visualization by turning it to right(display the next vertex) or left(display previous vertex).

4.5 Summary

This chapter showed how the hardware version of the system was developed. As in the Simulink block system where Simulink blocks were built from decomposed elements of the MATLAB algorithm, VHDL entities were concurrently designed, this part of the work explains the function of each entity and how it was designed. To verify results in the VHDL system, it was needed to add some external I/O peripherals; the chapter also explains how the micro SD memory card and the LCD were interfaced to the FPGA evaluation board.

The next chapter shows the resulting data from each system mode and by testing with different kinds of polygons.

Chapter 5 RESULTS

5.1 Introduction

This section presents results obtained in the three different implementations for the algorithm (MATLAB function, Simulink block system and VHDL system). A group of binary images in bmp format of size 64x64 - 574 bytes(512 bytes raw image + 62 bytes bmp format header) representing a combination of all possible type of vertices to be found were selected as examples. Each result comes from evaluating the identified vertices within each input binary image considering the minimum length of a horizontal/vertical line (M.L.H.) and the minimum contiguous corners with same distance (M.C.C.S.D.) input parameters. Figure 5-1 shows the images to be processed. Polygons 1 to 4 represent each type of vertex and polygons 5 to 7 are a combination of each type of vertex.

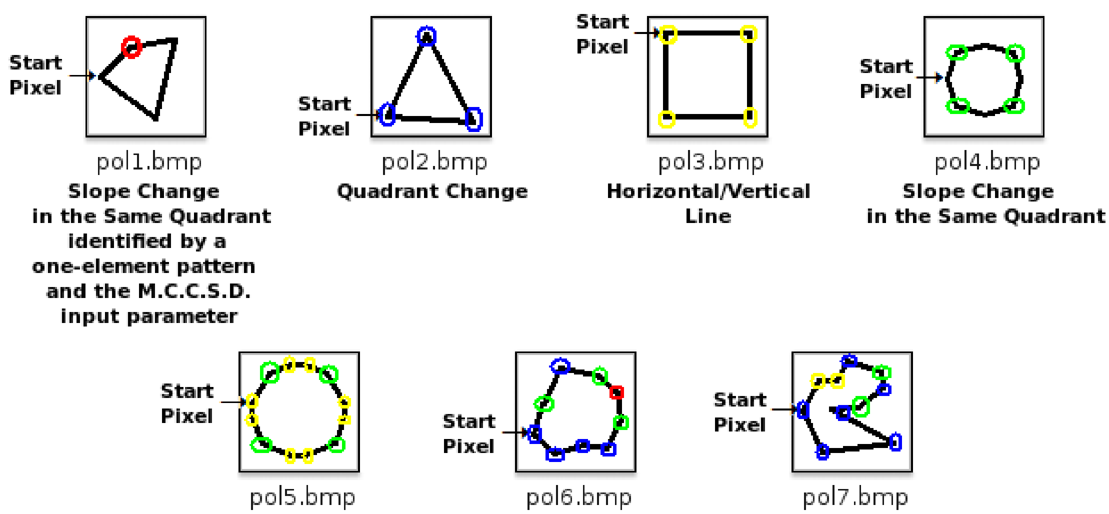


Figure 5-1 Test Images

5.2 Matlab Function Results

As mentioned before, while the main objective is to design a hardware system capable to locate vertices from polygons on a given input image, designing the vertex extraction algorithm, usage of software greatly simplifies the process. Computing and testing with images is an easy task in MATLAB, thus the algorithm was designed in MATLAB. Once that the algorithm was designed using MATLAB, for ease in algorithm testing, a MATLAB function called 'vertices' was written. The vertices function can be executed at the MATLAB command window. As output, the function plots the image showing the extracted vertices, and a list containing the (y x) coordinates of the identified vertices. The function format is as follows:

```
vertices('binary image route', M.L.H., M.C.C.S.D.)
```

Table 5-1 shows the located vertices (y x) coordinates from each image and its input parameters.

MATLAB FUNCTION INPUT DATA			**MATLAB FUNCTION OUTPUT DATA**
Binary Image	**M.L.H.**	**M.C.C.S.D**	**Vertices (y x)coordinates**
pol1.bmp	50	20	(15,23), (11,50), (57,39), (32,6)
pol2.bmp	50	50	(8,29), (59,57), (54,7)
pol3.bmp	5	50	(7,57), (56,57), (56,8), (7,8)
pol4.bmp	50	50	(17,19), (14,35), (21,51), (32,54), (50,50), (55,32), (47,15), (33,11)
pol5.bmp	5	50	(10,15), (5,27), (5,37), (10,49), (27,59), (36,59), (52,51), (58,37), (58,27), (51,11), (36,5), (27,5)
pol6.bmp	50	50	(27,13), (7,23), (11,46), (21,57), (38,59), (54,51), (52,35), (57,20), (44,8)
pol7.bmp	10	50	(14,13), (14,24), (3,29), (9,50), (20,51), (31,37), (32,27), (49,57), (56,15), (31,4)

Table 5-1 Matlab Function Results

Figure 5-2 shows the MATLAB function output for each image.

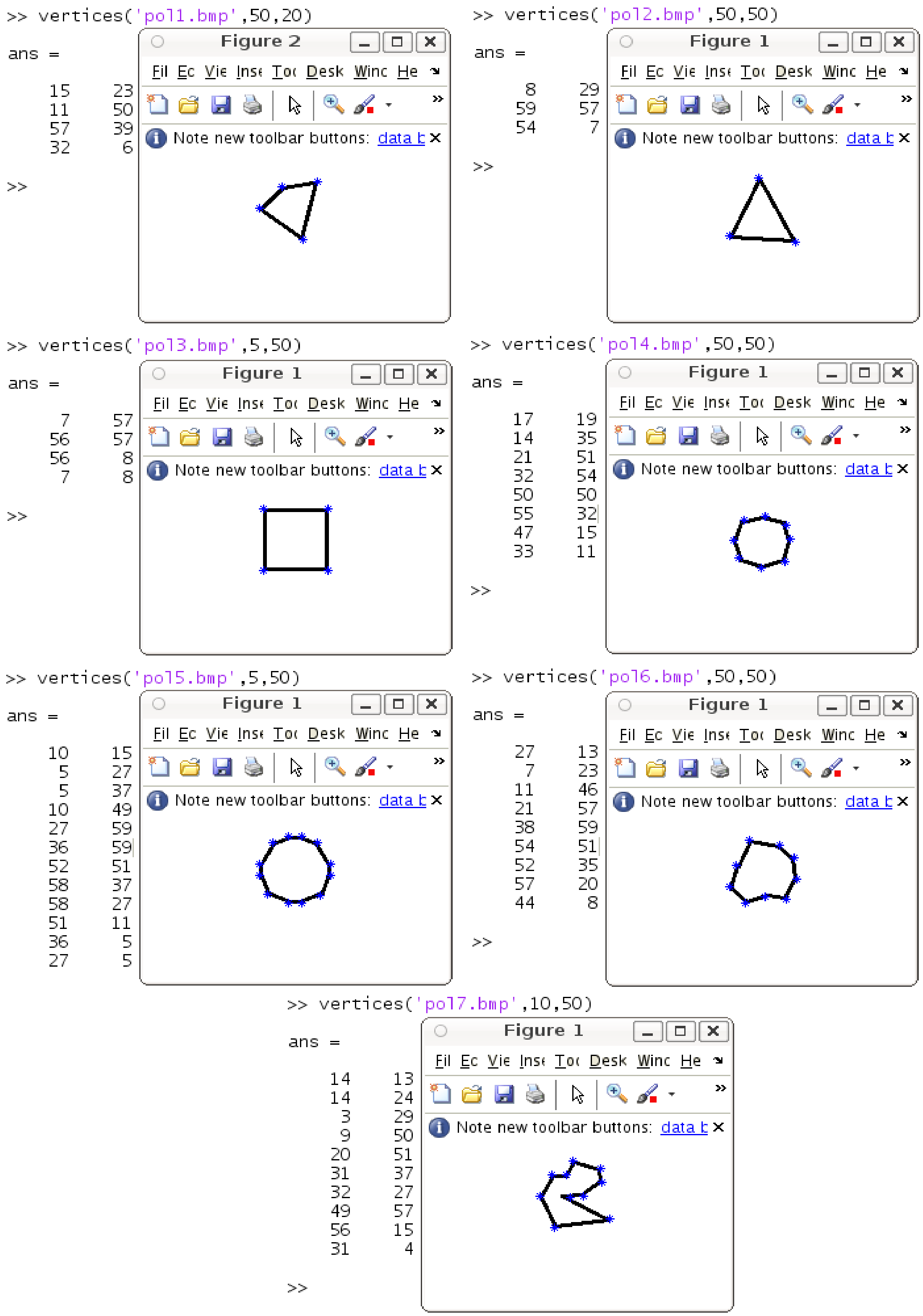


Figure 5-2 MATLAB Output Results

5.3 Simulink Block System Results

The Simulink block system was implemented to have a more understandable representation of the designed algorithm in MATLAB, thus hardware implementation could be easier. The main purpose of the Simulink block system was to assist in designing a VHDL equivalence of the algorithm. For Simulink block system testing, a binary image called *image.bmp* is needed to be placed in the current work directory. The *minimum contiguous corners with same distance* and *the minimum length of a horizontal/vertical line* input parameters are introduced as a constant in the required system block by double-clicking it as shown in Figure 5-3.

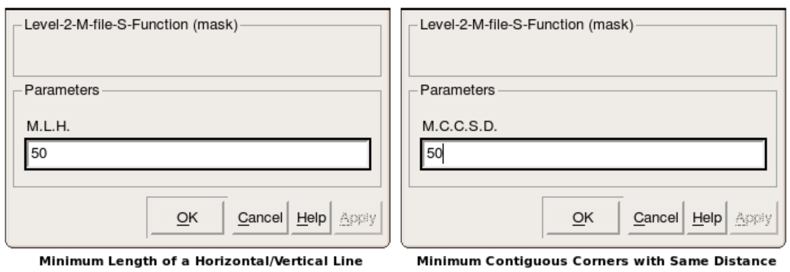


Figure 5-3 Input Parameters

As output, the system shows a plot of the introduced image with the identified corners and vertices, and generates a text file called *vertices.txt* in the current work directory which contains the vertices (y x) coordinates and the corner type and quadrant in which the vertex is located. Table 5-2 shows the input and output data for each image.

SIMULINK BLOCK SYSTEM INPUT DATA			SIMULINK BLOCK SYSTEM OUTPUT DATA
Binary Image	**M.L.H.**	**M.C.C.S.D**	**Vertices (y x)coordinates**
pol1.bmp	50	20	(15,23), (11,50), (57,39), (32,6)
pol2.bmp	50	50	(8,29), (59,57), (54,7)
pol3.bmp	5	50	(7,57), (56,57), (56,8), (7,8)
pol4.bmp	50	50	(17,19), (14,35), (21,51), (32,54), (50,50), (55,32), (47,15), (33,11)
pol5.bmp	5	50	(10,15), (5,27), (5,37), (10,49), (27,59), (36,59), (52,51), (58,37), (58,27), (51,11), (36,5), (27,5)
pol6.bmp	50	50	(27,13), (7,23), (11,46), (21,57), (38,59), (54,51), (52,35), (57,20), (44,8)
pol7.bmp	10	50	(14,13), (14,24), (3,29), (9,50), (20,51), (31,37), (32,27), (49,57), (56,15), (31,4)

Table 5-2 Simulink Block System Results

Figure 5-4 shows the Simulink block system output for each image.

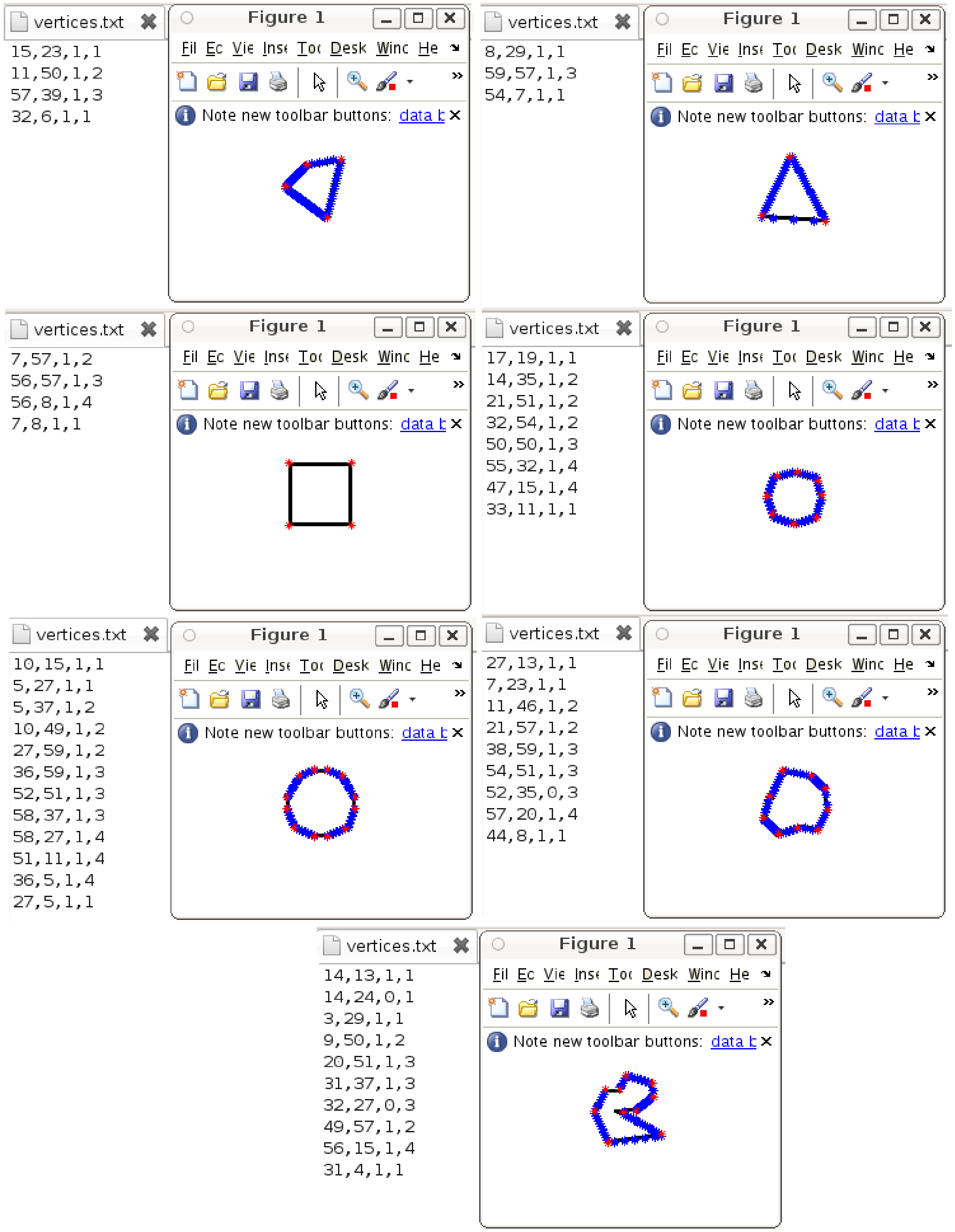


Figure 5-4 Simulink Output Results

5.4 FPGA System Results

Finally the algorithm was implemented in hardware. Each time a Simulink block was built from the refactored algorithm, a VHDL equivalent was concurrently designed, and thus I/O equivalence between the three systems was assured.

The FPGA system loads each image from the micro SD memory card and the input parameters are modified by constants into the VHDL code of the required entity. Results ((x y) coordinates, type of corner and quadrant in which vertices are located) are showed by a 16x2 alphanumeric LCD. The LCD shows results of only one vertex at once, so a rotary button is used to display all stored results within the *FPGA Block RAMs* (rotating to the right displays next result and to the left displays the previous result).

Table 5-3 shows the input and output data for each image.

FPGA SYSTEM INPUT DATA			**FPGA SYSTEM OUTPUT DATA**
Binary Image	**M.L.H.**	**M.C.C.S.D**	**Vertices (x y)coordinates**
pol1.bmp	50	20	(23,15), (50,11), (39,57), (6,32)
pol2.bmp	50	50	(29, 8), (57, 59), (7,54)
pol3.bmp	5	50	(57,7), (57,56), (8,56), (8,7)
pol4.bmp	50	50	(19,17), (35,14), (51,21), (54,32), (50,50), (32,55), (15,47), (11,33)
pol5.bmp	5	50	(15,10), (27,5), (37,5), (49,10), (59,27), (59,36), (51,52), (37,58), (27,58), (11,51), (5,36), (5,27)
pol6.bmp	50	50	(13,27), (23,7), (46,11), (57,21), (59,38), (51,54), (35,52), (20,57), (8,44)
pol7.bmp	10	50	(13,14), (24,14), (29,3), (50,9), (51,20), (37,31), (27,32), (57,49), (15,56), (4,31)

Table 5-3 FPGA System Results

Figure 5-5 to Figure 5-11 show the FPGA system output for each image.

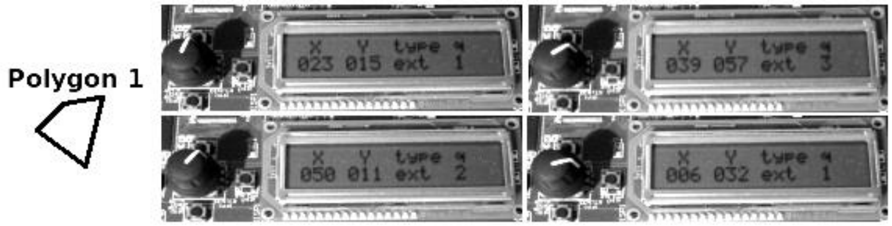


Figure 5-5 Polygon 1 FPGA results

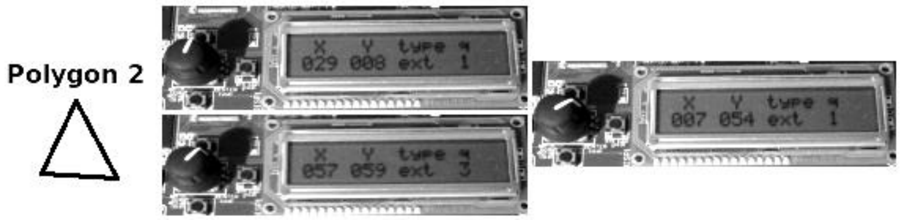


Figure 5-6 Polygon 2 FPGA results

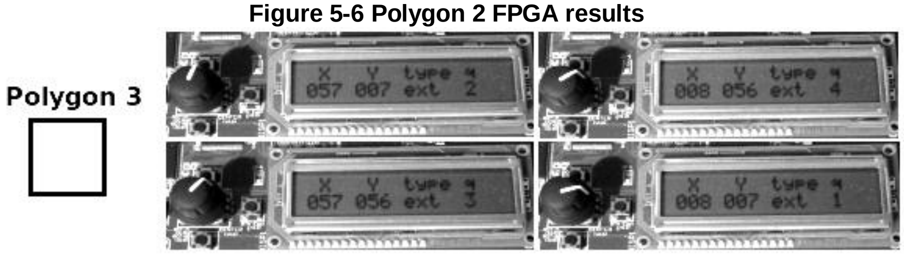


Figure 5-7 Polygon 3 FPGA results

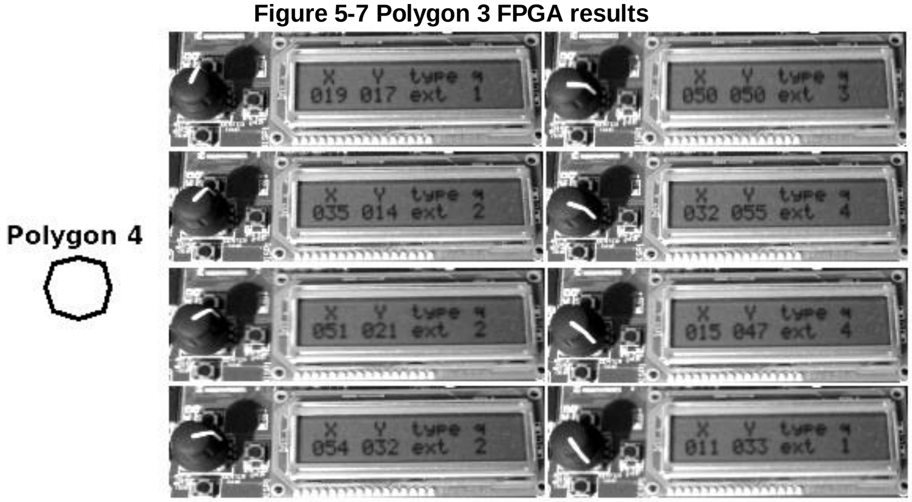


Figure 5-8 Polygon 4 FPGA results

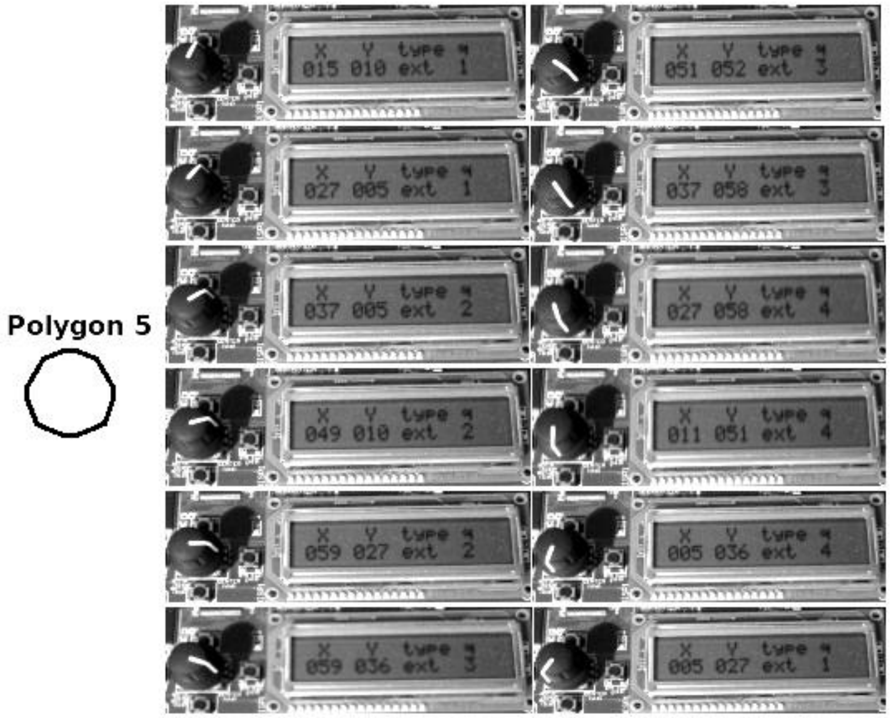


Figure 5-9 Polygon 5 FPGA results

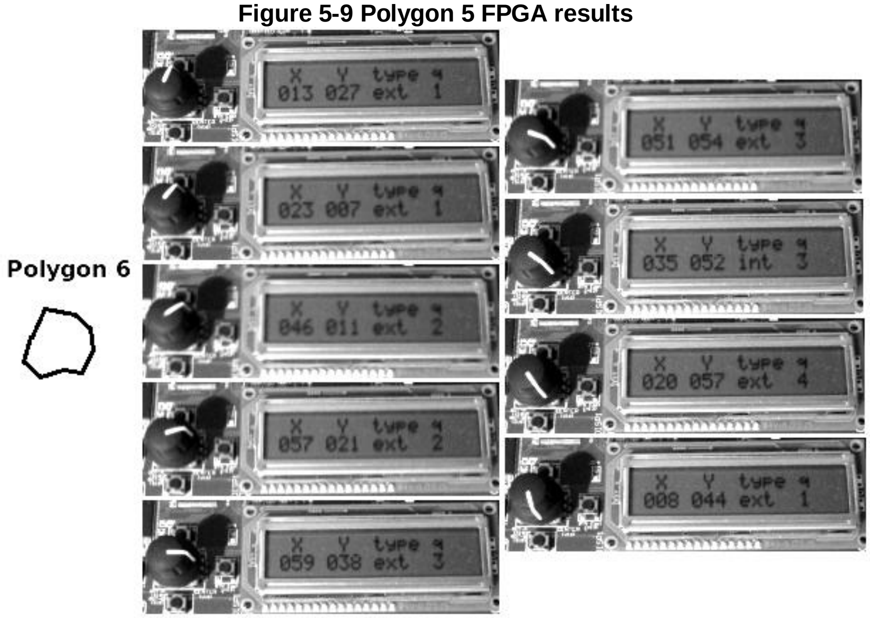


Figure 5-10 Polygon 6 FPGA results

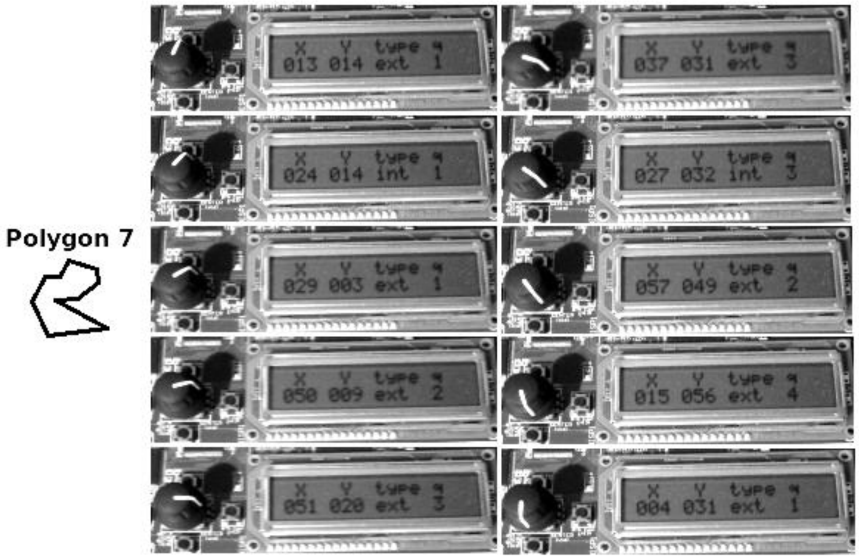


Figure 5-11 Polygon 7 FPGA results

5.5 Summary

MATLAB and Simulink Testing was performed by using MATLAB R2008a running under Linux Operating System (Ubuntu 9.10) in a Intel Core 2 Duo processor at 1.83Ghz of clock frequency, with 2 GBytes of RAM.

Although time performance is not within the purpose for the development of this system, by executing the *tic toc* MATLAB function, an estimated execution time for implemented algorithm was measured as an indicator that the algorithm execution time depends linearly of the combination between the number of pixels forming the polygon's edge, the number of edge pixels that are corners and the number of vertices found. Table 5-4 shows the obtained results.

Binary Image	Number of Pixels Forming the Edge	Number of Corners on the edge	Number of Located Vertices	Approximated Execution Time in Milliseconds
pol1.bmp	181	115	4	11
pol2.bmp	203	107	3	12
pol3.bmp	197	4	4	2
pol4.bmp	169	77	8	9
pol5.bmp	215	115	12	11
pol6.bmp	207	117	9	12
pol7.bmp	261	139	10	14

Table 5-4 Execution Time Performance

FPGA implementation was made using a Spartan 3E Starter Kit, the image was loaded into the FPGA from a micro SD memory card. The micro SD memory card was interfaced with the Starter Kit through the Micro Drive device. Results were displayed through the Starter Kit built-in 16x2 LCD. The FPGA system was tested with a 50Mhz clock. FPGA execution time performance measure was not contemplated in the design, so that no execution time results are documented.

Table 5-5 shows the Device Utilization Summary taken from the synthesis report. The synthesized VHDL system includes the designed micro SD memory and LCD communication drivers.

Device utilization summary	
Device	Xilinx Spartan 3E (500k gates)
Number of Slices	2212 out of 4656 47%
Number of Slice Flip Flops	883 out of 9312 9%
Number of 4 input LUTs	7287 out of 9312 78% *3255 35% used as logic* *4032 43% used as RAMs*
Number of IOs	15
Number of bonded IOBs	15 out of 232 6%
Number of GCLKs:	1 out of 24 4%

Table 5-5 Synthesis Report Device Utilization Summary

Although the larger number of located vertices inside an image is twelve (pol5-bmp), the algorithm, the maximum of vertices that can be located, depends on the distance between corners of each line segment and the M.L.H. and M.C.C.S.D. input parameters. Table 5-6 shows I/O equivalence between each system through the resulting vertices (x y) coordinates located in each of tested images.

	Vertices (x y)Coordinates		
Binary Image	**MATLAB FUNCTION**	**SIMULINK BLOCK SYSTEM**	**FPGA SYSTEM**
pol1.bmp	(23,15) (50,11) (39,57) (6,32)	(23,15) (50,11) (39,57) (6,32)	(23,15) (50,11) (39,57) (6,32)
pol2.bmp	(29, 8) (57, 59) (7,54)	(29, 8) (57, 59) (7,54)	(29, 8) (57, 59) (7,54)
pol3.bmp	(57,7) (57,56) (8,56) (8,7)	(57,7) (57,56) (8,56) (8,7)	(57,7) (57,56) (8,56) (8,7)
pol4.bmp	(19,17) (35,14) (51,21) (54,32) (50,50) (32,55) (15,47) (11,33)	(19,17) (35,14) (51,21) (54,32) (50,50) (32,55) (15,47) (11,33)	(19,17) (35,14) (51,21) (54,32) (50,50) (32,55) (15,47) (11,33)
pol5.bmp	(15,10) (27,5) (37,5) (49,10) (59,27) (59,36) (51,52) (37,58) (27,58) (11,51) (5,36) (5,27)	(15,10) (27,5) (37,5) (49,10) (59,27) (59,36) (51,52) (37,58) (27,58) (11,51) (5,36) (5,27)	(15,10) (27,5) (37,5) (49,10) (59,27) (59,36) (51,52) (37,58) (27,58) (11,51) (5,36) (5,27)
pol6.bmp	(13,27) (23,7) (46,11) (57,21) (59,38) (51,54) (35,52) (20,57) (8,44)	(13,27) (23,7) (46,11) (57,21) (59,38) (51,54) (35,52) (20,57) (8,44)	(13,27) (23,7) (46,11) (57,21) (59,38) (51,54) (35,52) (20,57) (8,44)
pol7.bmp	(13,14) (24,14) (29,3) (50,9) (51,20) (37,31) (27,32) (57,49) (15,56) (4,31)	(13,14) (24,14) (29,3) (50,9) (51,20) (37,31) (27,32) (57,49) (15,56) (4,31)	(13,14) (24,14) (29,3) (50,9) (51,20) (37,31) (27,32) (57,49) (15,56) (4,31)

Table 5-6 I/O Equivalence Results

Chapter 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

An algorithm to identify polygonal objects from binary images was designed in MATLAB. From the designed MATLAB algorithm, a MATLAB function called “vertices” was written to ease testing. As an input the vertices function uses the image’s border route, the minimum length of a horizontal/vertical line (M.L.H.) and the minimum contiguous corners with same distance (M.C.C.S.D.). As output, the function shows a plot of the processed figure with located vertices and a list of (x y) coordinates of each vertex.

A Simulink block library was also built by refactoring the MATLAB algorithm. Each block of the library was built by writing Level-2 M-file S-Functions from decomposed elements of the MATLAB algorithm. Once the library was completed, a block system was built and tested to match with the designed algorithm at I/O level.

A set of VHDL entities were almost simultaneously built with the Simulink block library thus I/O equivalence could be ensured at simulation level using Modelsim. Once that the set of VHDL entities was completed, the micro SD memory and Starter Kit built-in LCD were interfaced to the FPGA, this way, images could be loaded into the FPGA and results could be visualized through the LCD.

By testing the MATLAB function, the Simulink block system and the VHDL system all together, equivalence at I/O level was shown. A complete system, in software and hardware, was developed to contribute computer vision implementations of applications based on polygon identification.

6.2 Future Work

Make it possible not just to identify polygonal objects but also objects formed by curved segments.

Integrate the system into a computer vision application in which identifying polygons is part of the solution.

Use this work to design and develop a vertices' finding system of polygonal edges as a coprocessor in such a way that computer vision applications can be easier to develop.

FIGURES LIST

TABLES LIST

REFERENCES

[1] Yali Amit, *2D Object Detection and Recognition; Models, Algorithms, and Networks*. United States of America: MIT Press, 2002.

[2] Yong-Bin Zhang Fu-Cheng You, "A Mechanical Part Sorting System Based on Computer Vision," in *IEEE Computer Society, International Conference on Computer Science and Software Engineering*, 2008.

[3] William C. Regli Vincent Cicirello, "Machining Feature-based Comparisons of Mechanical Parts," in *IEEE Computer Society, Proceedings of the International Conference on Shape Modeling and Applications*, 2001.

[4] A. Kiritsakis, F. Stergiopoulos , Dimitris Bechtsis, V. Kilintzis, A. Antoniadis, K. Kiritsakis and K. Kokonas Efstratiadis, "Optical Inspection for Quality Separation of Olive Fruit," in *1st International Seminar Olive Biotechnology and the Quality of the end products in the Mediterranean Basin*, Errachida, Morocco, 2004.

[5] Ryoei Ito, Kenichi Nakanishi, Takashi Mishima, Toyokazu Hirozumi, Takaharu Kameoka, Masayuki Hirafuji, Seishi Ninomiya Atsushi Hashimoto, "An Integrated Field Monitoring System for Sustainable and High-Quality Production of Agricultural Products Based on BIX Concept with Field Server," in *IEEE Computer Society, Proceedings of the International Symposium on Applications and the Internet Workshops*, 2007.

[6] Flávio F. Feliciano, Flavius P. R. Martins Fabiana R. Leta, "Computer Vision System for Printed Circuit Board Inspection," in *ABCM Symposium Series in Mechatronics*, 2008, pp. pp.623-632.

[7] Dimitris N. Metaxas By Xiaolei Huang, "Metamorphs: Deformable Shape and Appearance Models," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008, pp. pp. 1444-1459.

[8] Paulo Lobato Correia Carlos Filipe Paulo, "Traffic Sign Recognition Based on Pictogram Contours," in *Ninth International Workshop on Image Analysis for Multimedia Interactive Services*, 2008.

[9] N. Magnenat-Thalmann and L. Moccozet, "Some New Challenging Research Topics in Human Animation," in *IEEE Computer Society, Proceedings of the 17th Spring Conference on Computer Graphics*, 2001.

[10] Golan Levin, "Computer Vision in Interactive Art," in *Processing, a Programming*

Handbook for Visual Designers and Artists.: MIT Press, 2007, pp. 547-555.

[11] Vijay Laxmi, Bhavitavya Bhadviya, Nemi Chand Singh and Ashwini Sharma Neeta Nain, "Decomposing a Polygonal Boundary as a List of Line Segments," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 2008.

[12] Carla Maria Modena Stefano Messelodi, "Detection of Polygonal Frames in Complex Document Images," in *10th International Workshop on Database and Expert Systems Applications*, 1999.

[13] Sriram Swaminarayan Lakshman Prasad, "Hierarchical Image Segmentation by Polygon Grouping," , 2007.

[14] Liu Shenglan, Wu Xi, Zhou Laishui Zhang Liyan, "Segmentation and Parametrization of Arbitrary Polygon Meshes," in *Proceedings of the Geometric Modeling and Processing*, 2004.

[15] B. Gatos and S.L. Mantzaris, "A novel recursive algorithm for area location using isothetic polygons," in *Proceedings of the International Conference on Pattern Recognition*, 2000.

[16] Gareth Loy, David Shaw and Antonio Robles-Kelly Nick Barnes, "Regular polygon detection," in *Proceedings of the Tenth IEEE International Conference on Computer Vision*, 2005.

[17] Richard E. Woods Rafael C. Gonzalez, *Digital Image Processing*. United States of America: Pearson-Prentice Hall, 2002.

[18] Roberto Marcondes Cesar Jr. Luciano da Fontoura Costa, *Shape Analysis and Classification*.: CRC Press LLC, 2001.

[19] Luc Vincent, Vance Faber Artem Mikheev, "High-Quality Polygonal Contour Approximation Based on Relaxation," in *Sixth International Conference on Document Analysis and Recognition (ICDAR'01)*, Seattle, Washington , 2001.

[20] Longin Jan Latecki and Rolf Lakämper, "Polygon Evolution by Vertex Deletion ," in *Lecture Notes in Computer Science, Springer*, 1999.

[21] Bimal Kr. Ray and Kumar S. Ray, "An algorithm for polygonal approximation of digitized curves," in *Pattern Recognition Letters, Volume 13, Issue 7*, 2003.

[22] Filip Krolupper and Jan Flusser, "Polygonal shape description for recognition of partially occluded objects," in *Pattern Recognition Letters*, *Volume 28, Issue 9*, 2004.

[23] Po-Hsuan Liaoa and Jia-Ming Changa Kuo-Liang Chung, "Novel efficient two-pass algorithm for closed polygonal approximation based on LISE and curvature constraint criteria," in *Journal of Visual Communication and Image Representation*, 2006.

[24] Ralf Juengling and Lakshman Prasad, "Parsing Silhouettes without Boundary Curvature ," in *14th International Conference on Image Analysis and Processing (ICIAP 2007)*, 2007.

[25] Cecilia Di Ruberto and Andrea Morgera, "A New Algorithm for Polygonal Approximation Based on Ant Colony Optimization ," in *Lecture Notes in Computer Science, Springer*, 2009.

[26] Marc Salotti, "Optimal polygonal approximation of digitized curves using the sum of square deviations criterion," in *Pattern Recognition Society. Published by Elsevier Science B.V.* , Octeville, France, 2000.

[27] Alexander Kolesnikov, "Constrained piecewise linear approximation of digital curves ," , 2008.

[28] Jack Chazin, Robert L. Hansen, Bruce J. Sklansky, "Minimum-Perimeter Polygons of Digitized Silhouettes ," in *Computers, IEEE Transactions*, 2009.

[29] Xavier Provençal and Jacques-Olivier, "Two Linear-Time Algorithms for Computing the Minimum Length Polygon of a Digital Contour," in *Discrete Geometry for Computer Imagery, Springer*, 2009.

[30] Robert Bergevin and Marielle Mokhtari, "Multiscale Contour Segmentation and Approximation: An Algorithm Based on the Geometry of Regular Inscribed Polygons," in *Computer Vision and Image Understanding, Volume 71, Issue 1*, 1996.

[31] Reinhard Klette Fajie Li, "Decomposing a Simple Polygon into Trapezoids ," in *Computer Analysis of Images and Patterns*, 2007.

[32] Partha Bhowmick and Bhargab B. Bhattacharya, "Fast Polygonal Approximation of Digital Curves Using Relaxed Straightness Properties," in *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 29, NO. 9*, 2007.

[33] George C. Stockman Linda G. Shapiro, *Computer Vision*. United States of America: Prentice Hall, 2001.

[34] Keith Price. *Computer Vision Information Pages*. [Online]. http://www.visionbib.com. [Accessed: Febrero2009]

[35] Denise Guliato, Juliano D. de Carvalho, Sérgio A.Santiago Rangaraj M. Rangayyan, "Polygonal approximation of contours based on the turning angle function," in *Journal of Electronic Imaging 17(2), 023016 (Apr–Jun 2008)*, 2008.