Object Recognition for Real-Time Java

Steven R. Howard

under the direction of
Professor Martin Rinard
MIT Laboratory for Computer Science

Research Science Institute
July 30, 2002

Abstract

Object recognition is a common computer vision problem with a wide range of applications. This paper describes a method of object recognition based on matching edge detection models to a database of images. The entire process is then executed inside a multithreaded architecture executing with real-time requirements so that the engine can process input from a live video stream. The final engine successfully identified a simple object in a variety of pictures and was able to distinguish the object from other simple objects in the picture. This system could easily be used in a large number of applications due to it's flexible database design and reliable edge detection algorithms.

1 Introduction

Object recognition is an important and challenging problem with a wide range of applications, especially in industry and robotics. In order to do something useful with vision, a machine must be able to identify objects in an image and compute the location of those objects [5]. Object recognition involves training computers to locate and identify specific objects in an image or set of images. This problem has been the subject of frequent research for several decades and although no computer today can recognize objects as well as a human can, a multitude of techniques have been invented to allow computers to analyze images and attempt to identify some objects in the scene. These techniques can vary greatly in complexity, from looking at the color of each region of the image to building a three-dimensional model of an object from several pictures. One problem of more complex algorithms, however, is that they can take a long time to process a single image or a short sequence of images. Processing time is not a problem for working with previously photographed images, but it is unacceptable for real-time image processing. Real-time processing is necessary whenever a continuous image data stream must be analyzed and a response must be generated quickly, such as when performing vehicle control.

1.1 Image processing techniques

Implementation of a real-time object recognition system involves two main problems. The first problem is designing a system to consistently recognize an object without taking too much processor time. A number of image processing algorithms were considered for this task. The simplest algorithm is *thresholding*. Thresholding is a means of separating foreground pixels from background pixels by simply setting to full intensity all the pixels that are above a certain value, and setting the rest of the pixels to zero [3]. This method is useful in later image processing steps, but not as the first step. Another simple algorithm is *connected component labeling*. Connected component labeling finds distinct foreground objects by labeling sets

of connected pixels that have a similar color [3]. This algorithm is effective only for simple images and is not capable of working with the raw images in this project, but becomes useful in later processing. The next set of algorithms is known as *region segmentation*. These algorithms are all designed to split the image into several regions, each of which is similar in terms of color, intensity or some other criteria. This category includes *split-and-merge, region growing, recursive splitting,* and several other algorithms [6, 7, 9]. These algorithms could be used to effectively find the regions of the image, but edge detection was chosen for this project since it the edge data would be more useful in later processing.

Edge detection describes a collection of algorithms designed to find the edges of objects in an image. There are many different edge detection algorithms, including *Roberts Cross, Sobel, Canny,* and *Laplacian of Gaussian* [2, 3, 4]. Edge detection is a fast and effective method of finding objects in an image and further processing can use the edge data to identify objects. After initial edge detection, there are a variety of algorithms designed to enhance the edge data, including *graph searching, hysteresis tracking*, and several morphological operators such as *erosion, thinning* and *pruning* [3, 7]. Next, the edges must be grouped together to form distinct objects. This can be done with *contour tracking* or with connected component labeling performed on the edge data [7].

Once an object is found, it must be analyzed to determine if it is the desired object. One method of identifying an object involves first finding key points on the object. This can be done with *symmetry detection*, which finds the points of maximum rotational symmetry, or *corner detection*, which finds the corners of lines in the image [7, 10]. Next, these object can be analyzed using *point similarity*, which examines the key points for certain features. Another method of identification involves looking for *geometric invariants*, or features of the object which are always present. Finally, a database of template images can be used to match up an object with previously processed data [7]. This approach is know as *template matching.*

1.2 Real-time execution

The second problem involves writing the program on an architecture that can run in real-time without falling behind the video stream. Running in *real-time* means that the application will execute in a predictable way every time and conform to given timeliness constraints [1]. To accomplish this goal, the object recognition engine will use a multithreaded design, with each thread working on a different stage of the object recognition process. This is similar to the pipeline design used in many processor architectures or to the assembly line idea used in manufacturing. The basic idea behind this method is that a complex operation can be split up into a series of simpler operations. One way to perform such a complex operation on a large amount of data is to sequentially run all the simple operations to get a final result for one piece of data and then start over with the next piece of data. Using this method, however, the program can easily get delayed on a single step and it is difficult to keep the program on schedule without dropping data.

The solution to this problem is to have a separate thread performing each simple operation. Every thread has a deadline, the maximum amount of time it may spend working on a single frame. Each thread takes a single frame and works on it, generating and refining results until it has accomplished its task or hit its deadline. At this point, the results from this thread are passed off to the next thread. If thread hits it deadline before it has obtained acceptable results, it will complete its processing and immediately begin the next cycle. Most threads, however, should generate acceptable results in a very short time and then expand or refine those results for the rest of the cycle. Each frame passes through the threads one by one until the final thread generates an output, at which point the frame is discarded. An illustration of this process can be found in Appendix D.

2 Theory behind object recognition

2.1 Edge detection

Edge detection is the process of attempting to find the edges of actual objects in an image. It is very useful in computer vision because edge data is much easier to analyze and understand than the original picture. For example, edges can be further processed to analyze an object's shape. In addition, edge detection produces a simple representation of an object which can be matched to a database of edge-detected pictures to identify an object. This method is described later.

The edge detection algorithm used for this project, *Roberts Cross*, is based on examining the spatial gradient of an image [3, 7]. Roberts Cross was chosen for this project because its simple design allows for very fast computation. Roberts Cross begins with an image represented as an array of pixels, each pixel having a single intensity value. For color images, each pixel is represented by three intensity values, one for each of the three color components red, green, and blue, so there are really three images to be analyzed. The spatial gradient at each pixel is a vector that describes how quickly the intensity is changing at that point and in which direction the change occurs. The Roberts Cross operator estimates the magnitude of this gradient vector at each pixel. At points that are on the edge of an object, color intensity will be changing quickly, so the magnitude of the spatial gradient will be high and the Roberts Cross operator will generate a high output value. To perform the estimation of the gradient magnitude at each pixel in an image, the Roberts Cross operator uses *convolution*. See Appendix C for a full explanation of convolution.

The Roberts Cross operator uses a set of two convolution kernels to estimate gradient components G_x and G_y. These kernels are given below.

$$G_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad (1)$$

$$G_y = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad (2)$$

These kernels find the difference in intensity between adjacent pixels along 45° angles. The final magnitude is then given by:

$$|G| = \sqrt{G_x^2 + G_y^2} \quad (3)$$

Sample output from this algorithm is shown in Figure 5.

2.2 Edge refinement

The output from the Roberts Cross operator alone is not simple enough to be analyzed for object recognition. The data still contains too much noise, some real edges may be weak, and some false edges may be present. There are several methods of "cleaning up" the edge detection data to make it easier to analyze.

2.2.1 Hysteresis tracking

Hysteresis tracking attempts to set all real edges to full intensity and the rest of the pixels to zero by thresholding the edge pixels at two different values [2]. The first step of hysteresis involves dividing the edge pixels into three categories using two thresholds, T_1 and T_2, as shown below.

1 For each pixel P:

2 If $(value(P) \geq T_1)$ P is definitely an edge

3 If $(T_2 \leq value(P) < T_1)$ P is a possible edge

4 If $(value(P) < T_2)$ P is definitely not an edge

All definite edges are set to full intensity, and all definite non-edges are set to zero. Next, each pixel that is a "possible edge" is examined. If that pixel is next to a definite edge, then it becomes a definite edge and gets set to full intensity. Otherwise, the pixel is set to zero. The process of resolving possible edges repeats until it achieves convergence, i.e. no more pixels are changed. The result is an image consisting of only full intensity and zero intensity pixels. This is illustrated in Figure 6. However, the image may still contain some false edge pixels and some edges that are more than one pixel thick, which causes unnecessary complexity and must be removed before analyzing individual objects.

2.2.2 Thinning and pruning

Thinning and *pruning* are two similar operations that attempt to remove extraneous edge pixels [3, 7]. Both thinning and pruning are *hit-and-miss operations*. Hit-and-miss operations are a way to find certain features in images where each pixel is either "on" or "off," as is the case after hysteresis. Each hit-and-miss operation is based on a set of structuring elements, which are somewhat similar to convolution kernels. A structuring element defines a certain pattern of pixels in the immediate region around a target pixel. For each one in the structuring element, the underlying pixel must be "on," and for each zero, the underlying pixel must be "off." If every number in the structuring element matches its underlying pixel then the value of the target pixel is switched.

The thinning operator attempts to reduce each line to a single pixel wide by turning off pixels that belong to "thick" lines. This is accomplished by deleting pixels one by one without eliminating points that are on single-pixel wide lines. The structuring elements for

thinning are given in Figure 1.

0	0	0
	1	
1	1	1

	0	0
1	1	0
	1	

Figure 1: Structuring elements for thinning. The matrix is overlayed onto the image with the target pixel at the center. Blank squares are not examined.

For each iteration, these structuring elements are applied in all four possible 90° rotations to every edge pixel. This process repeats until convergence. The output of thinning is given in Figure 7.

Thinning leaves an image consisting mainly of single-pixel width lines, but often results in "spurs" left over on some edges. Figure 2 illustrates some spurs in a thinned image. Pruning eliminates these spurs by removing the endpoint pixels for single-pixel width lines on each iteration, effectively deleting spurs pixel by pixel. The structuring elements for this operation are shown in Figure 3.

As with thinning, these elements are applied in all four possible 90° rotations. Unlike thinning, pruning is usually not applied to convergence since it would eliminate all lines that do not form closed loops. In this project, however, pruning is applied to convergence since it is capable of separating edges into closed objects, which is desirable for this application. The output of this operator is shown in Figure 8.

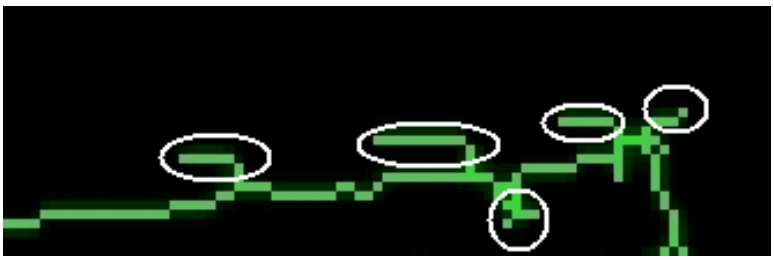

Figure 2: A close-up image showing spurs left over in thinned edge detection data.

0	0	0
0	1	0
0		

0	0	0
0	1	0
		0

Figure 3: Structuring elements for pruning.

2.3 Object labeling

After the image has been pruned to convergence, it will normally consist of several separated groups of pixels. The next step is to scan the image and label each separate object. This is done using connected component labeling. Connected component labeling simply assigns a distinct number to each contiguous group of pixels. Once this is completed, the width and height of each object are measured and stored. Finally, the objects which are too small or too large are deleted by comparing each object's dimensions to a set of threshold values. The output of this stage is a set of objects of reasonable size, each consisting of a group of edge pixels. Figure 9 illustrates this output by assigning colors to each object.

2.4 Template matching

The final stage in object recognition is to attempt to match each object against a previously generated database of template images. This method allows the same engine to be used to recognize different objects by generating a database of images for each object that needs to be identified. The database is generated by taking a large set of pictures of the object to be identified and running the edge detection and refinement algorithms on each image. The final result is then stored and can be compared with the edge representation of an unknown object to find a match.

Once object labeling is complete the program has a set of pixels which represent each unknown object. The program can line up each object with each database picture and analyze the images to find how closely the two correspond. If the correspondence if above a certain threshold value, then the object is a match and has been identified. Figure 10 shows

an example of a scaled representation of an unknown object and Figure 11 shows a database used to recognize this object. Specific methods of lining up and comparing images used in this project are discussed in Section 3.2.4.

3 Implementation

3.1 Multithreaded architecture

Implementation of this program began with the multithreaded architecture. This architecture is based on the concept of using a separate thread for each stage of image recognition. The program is designed such that all threads run in parallel, with each thread working on a different frame and data getting passed off between threads. Each thread executes a short piece of logic repeatedly for each frame that gets processed. In order to ensure that the program keeps up with the video stream, every thread has a deadline, a certain amount of time after which the thread is interrupted and forced to begin processing the next frame. Consequently, data processing could be interrupted at any time, so each thread's logic must be designed to generate rough results very quickly, then progressively refine the results until the deadline handler is triggered. To facilitate the transfer of data from thread to thread, each frame is processed in a separate block of memory. When a thread finishes with a frame, it passes to the next thread a reference to the memory area in which that frame is being processed. This design is illustrated in Figure 23.

3.1.1 The Real-time Specification for Java

The architecture was implemented in the Java programming language using the Real-time Specification for Java (RTSJ). This specification expands on the Java programming language to allow timely execution of Java threads by implementing a scheduler that supports periodic threads with deadlines and a memory manager that allows allocation of memory without interference from the garbage collector. The memory manager is necessary because the garbage

collector can interrupt normal Java programs and spend significant time to move objects in memory. The RTSJ allows threads to run at higher priority than the garbage collector, and it allows threads to allocate objects that are guaranteed to remain untouched by the garbage collector. In addition, it specifies memory allocation methods that are guaranteed to finish in linear time (in the amount of the memory allocated), further ensuring that no unexpected slowdowns will occur. Programs using the RTSJ allow for much more timely execution than can be achieved using a standard Java program. Real-time Java (RTJ) programs can be compiled and run using the RTJ reference implementation which was developed by TimeSys Corporation [8].

Scheduling of the threads and execution deadlines are both handled by the RTSJ scheduler. Threads are constructed using the `PeriodicParameters` class, which takes a start time and period in milliseconds. The period is the duration of each cycle, after which time the deadline occurs. From these two parameters, the scheduler will automatically split up execution time among each thread and invoke the appropriate handlers when deadlines happen. Memory management is handled by the `LTMemory` class, which is a subclass of the `ScopedMemory` class. *Scoped memory* is a block of memory that only exists while it is being using by at least one thread. To use scoped memory, a thread enters a scope and then runs code within the scope. While inside a scope, all objects are allocated inside that scope. To pass scoped data between threads, the `SetPortal()` and `GetPortal()` methods are used. `SetPortal()` is first called on the main data structure from the first thread, to make that object the portal object for this scope. Then, when another thread enters that scope, it can call `GetPortal()` to get a reference to the data structure. With that reference, the thread can do its work and then pass data off to the next thread. The `LTMemory` class simply guarantees that memory allocation will take a linear amount of time.

3.1.2 Class design

The first class to be implemented was the `ProcessThread` class. This class extends the `RealtimeThread` class, which is the basic thread class in the RTSJ. `ProcessThread` implements only the functionality for handling data and passing data between threads. In particular, it implements an initialization method to retrieve a reference to the next frame, and a finalization method to pass off a reference to the next thread.

The next class to be implemented was the `PeriodicThread` class. `PeriodicThread` extends `ProcessThread`, adding logic to execute periodically and to handle missed deadlines. The main loop of this class calls the `Initialize()`, `ProcessData()`, and `Finalize()` methods to process each frame. The `Initialize()` and `Finalize()` methods are defined in the `ProcessThread` class but must be overridden by the first and last threads to allocate new memory and to handle the final output, respectively. The `ProcessData()` method is declared in `ProcessThread` and must be overridden by subclasses of `PeriodicThread` to implement custom processing logic. These two classes implement the basic logic for the multithreaded architecture.

The `ProcessThread` and `PeriodicThread` classes define all the necessary functionality to run periodic threads with deadlines and to pass data between threads. By subclassing `PeriodicThread` several times and overriding the `ProcessData()` method with custom logic for each class, a set of thread classes can be implemented that will run in parallel and process data as explained above. The only other necessary code is an `Initialize()` method in the first thread to allocate a new `ScopedMemory`, and a `Finalize()` method in the last thread to do something useful with the output rather than passing it to another thread.

3.2 Image recognition

3.2.1 Edge Detection

The Roberts Cross edge detector was implemented by iterating through all pixels in the image and performing the convolution operation in a single computation:

$$G_x(i,j) = I(i,j) - I(i+1,j+1) \tag{4}$$

$$G_y(i,j) = I(i+1,j) - I(i,j+1) \tag{5}$$

$$|G(i,j)| = \sqrt{G_x^2 + G_y^2} \approx |G_x| + |G_y| \tag{6}$$

$$|G(i,j)| = |I(i,j) - I(i+1,j+1)| + |I(i+1,j) - I(i,j+1)| \tag{7}$$

The magnitude is approximated as the sum of the absolute values of G_x and G_y since this is much faster to compute. The computation is performed separately for each channel. After convolution, the results are combined into a single channel image. The final output value O is given by:

$$O(i,j) = max(R(i,j), G(i,j), B(i,j)) \tag{8}$$

where R, G, and B and the red, green, and blue components, respectively. The maximum value of the three channels is taken since the algorithm is looking for edges on any channel and a strong edge found on one channel overshadows the result of the other two channels. This does result in loss of data, but the loss is insignificant. The output of this operator is shown in Figure 5.

3.2.2 Edge refinement

Hysteresis tracking was first implemented as a simple iterative operator which repeated until convergence. However, it is inefficient and proved to be very time consuming. The iterative method examines every "possible edge" pixel on every iteration, moving from top to bottom, which is very inefficient. A much faster way to perform hysteresis tracking is to use a recursive

function which starts from a definite edge pixel and moves outward, following all "possible edge" pixels that are connected to it. The algorithm for this function is shown below.

HYSTERESIS(P)

1 set pixel P to full intensity
2 for each pixel Q adjacent to P
3 if Q is a possible edge
4 HYSTERESIS(Q)

The program iterates through every pixel and calls the hysteresis function once for each definite edge pixel. This method runs very quickly and works just as well as the iterative method. The output of this operation is shown in Figure 6.

Thinning was implemented with an iterative design. In each iteration, an array is filled with the values of the eight adjacent pixels. The program then performs comparisons with these values to test for a match. If a match is found, the pixel is set to zero. The output from this operator is shown in Figure 7.

Pruning was implemented recursively, since it must run to convergence and would be inefficient if implemented iteratively. The algorithm for the recursive pruning function is given below.

PRUNING(P)

1 if pixel P matches a structuring element
2 set pixel P to zero
3 for each pixel Q adjacent to P
4 if Q is an edge
5 PRUNING(Q)

This function effectively traces along each non-closed line from the endpoint inwards, deleting it until it reaches a closed loop. The mechanism for testing a structuring element works in the same way as with thinning. Output for this operator is shown in Figure 8.

3.2.3 Object labeling

Since object labeling involves tracing along continuous edges, it was implemented recursively in a similar fashion to pruning. The algorithm is given below.

LABELING(*P, id*)
1 set pixel *P* to value *id*
2 for each pixel *Q* adjacent to *P*
3 if *Q* is an edge
4 LABELING(*Q, id*)

The program iterates through each pixel and calls this function on each unlabeled edge pixel, incrementing *id* each time it makes a new call. Next, the bounding box of each object is examined. If the width or height are not within specified bounds, the object is removed from the picture. The output at this point is illustrated in Figure 9.

3.2.4 Template matching

The first step of template matching is to line up the unknown image with the database image. This is done using the bounding box found in the previous stage. The unknown image is scaled such that it's bounding box matches the database image's dimensions. Several examples of scaled images are given in Figure 10.

Once the images are lined up, a nearest-neighbor operation is performed to find the correlation between the two images. This operation iterates through each pixel in the template image and finds the nearest edge pixel in the unknown object image, recording the distance between the two pixels. The correspondence value is the sum of all such distances, which indicates how closely the template image can be found in the unknown image. If the correspondence value is below a certain threshold value, the object is considered a match. This process is repeated for each database image and for each object. Finally, the program takes

the best match and uses it as the final result. Figure 11 shows the database used for testing this program, and Figure 12 shows the final output for one frame.

4 Results

The object recognition engine successfully recognized a Zagi© airplane in a variety of frames captured from a movie file. Several example frames and output for those frames are given in Appendix B. The engine was also able to distinguish the Zagi from other objects in the sky, such as birds (Figures 19 and 20). This demonstrates the accuracy of the edge detection process and the database matching method. The engine was also able to recognize the plane when the edge detection returned significant noise in the body of the plane.

This tolerance for noise, however, can also be a drawback, since an object which scales to a very noisy image will match up with almost any database image. Ideally the thinning and pruning operators would eliminate enough noise to avoid this problem, but it is still a potential cause of a false positive.

Another weakness of the engine lies in the edge detection itself, most importantly in the hysteresis tracking. The edge detection process sometimes fails to correctly pick up the edges of the plane, especially in blurry images, and the results in a false negative. One such example is illustrated in Figure 21.

Timeliness results for execution on the multithreaded architecture are not available since the Real-time Java reference implementation is very slow and contains bugs and no other implementation is currently available. Once an improved implementation is available, the complete engine can be tested to see how well it can maintain a steady framerate.

5 Conclusions and Future Work

This paper describes a complete method of recognizing objects in real-time. A collection of image analysis algorithms was designed to run as quickly as possible while retaining the ability to recognize an object in a variety of environments. The multithreaded architecture and use of the RTSJ allows the program to conform to strict deadlines so that results will always be available when they are expected. The final implementation was able to recognize a simple object in a variety of pictures and distinguish between different shapes.

Future extensions to this project may include attempting to optimize the database to improve speed or reliability as well as testing to find out how well the engine recognizes other objects with a new database. In addition, the image correspondence algorithm used for database comparison could be improved so that it produces less false positives. Another possibility is to make some additional output based on which database picture generated the best match. For example, the sample database used for testing this project (Figure 11) could be used to determine the direction in which the plane is moving by assigning a direction to each database image. This information could be used to produce more detailed output.

6 Acknowledgments

I would like to thank my mentor, Professor Martin Rinard of the MIT Laboratory for Computer Science, and my day-to-day mentor, Mr. Wes Beebee, for their guidance and assistance during my research. They gave me invaluable help with my project and helped me learn many new concepts and techniques. I would also like to thank the Center for Excellence in Education and the Research Science Institute for giving me the opportunity to do this research and finding me an excellent mentor. Finally, I would like to thank my tutor, Jenny Sendova, for her help with my paper and my presentation.

References

[1] Bollela, Greg, et al. *The Real-time Specification for Java.* Addison Wesley, 2000.

[2] Canny, J. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis.* Vol. 8, No. 6, November 1986.

[3] Fisher, R., et al (2002). *Image Processing Learning Resources.* Retrieved July 29, 2002 from `http://www.dai.ed.ac.uk/HIPR2`.

[4] Freeman, H., Ed. *Machine Vision.* San Diego: Academic Press, 1988.

[5] Grimson, W. E. L. *Object Recognition by Computer.* Cambridge: MIT Press, 1990.

[6] Haralick, R. M. and Shapiro, L. G. A Survey: Image Segmentation Techniques. *Journal of Computer Vision, Graphics and Image Processing.* Vol. 29, No. 1, 1985: 100-132.

[7] Pratt, W. K. *Digital Image Processing.* New York: John Wiley & Sons, Inc., 1991.

[8] TimeSys Corporation RTSJ Reference Implementation. Retrieved July 2, 2002 from `http://www.timesys.com/prodserv/java/index.cfm`.

[9] Ullman, S., and Richards, W., ed. *Image Understanding.* New Jersey: Ablex Publishing Corporation, 1990.

[10] Winston, P. H. *Artificial Intelligence.* Reading: Addison-Wesley, 1992.

A Step by step illustration of object recognition

Figure 4: A sample image.

Figure 5: The result of applying the Roberts Cross operator to the sample image.

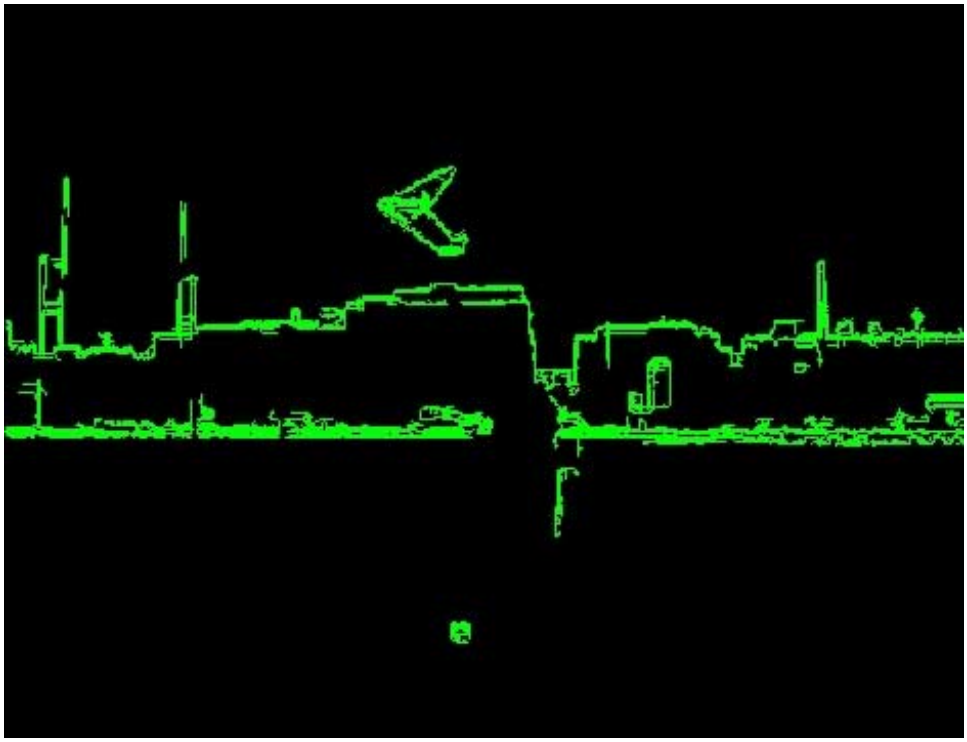

Figure 6: The result of running hysteresis tracking.

Figure 7: The result of applying thinning to convergence.

Figure 8: The result of applying pruning to convergence. All lines now form closed loops.

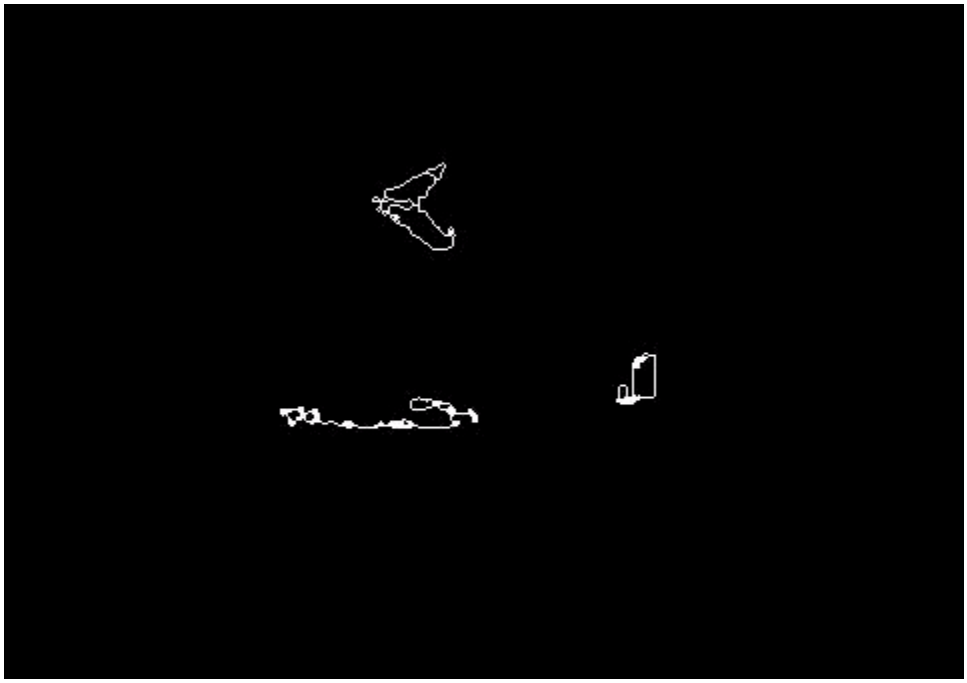

Figure 9: Image showing the objects identified by connected component labeling. Objects that did not fit reasonable dimensions have been removed.

Figure 10: Scaled representations for the three objects identified in the sample image. The scale used here is 100 x 100 pixels.

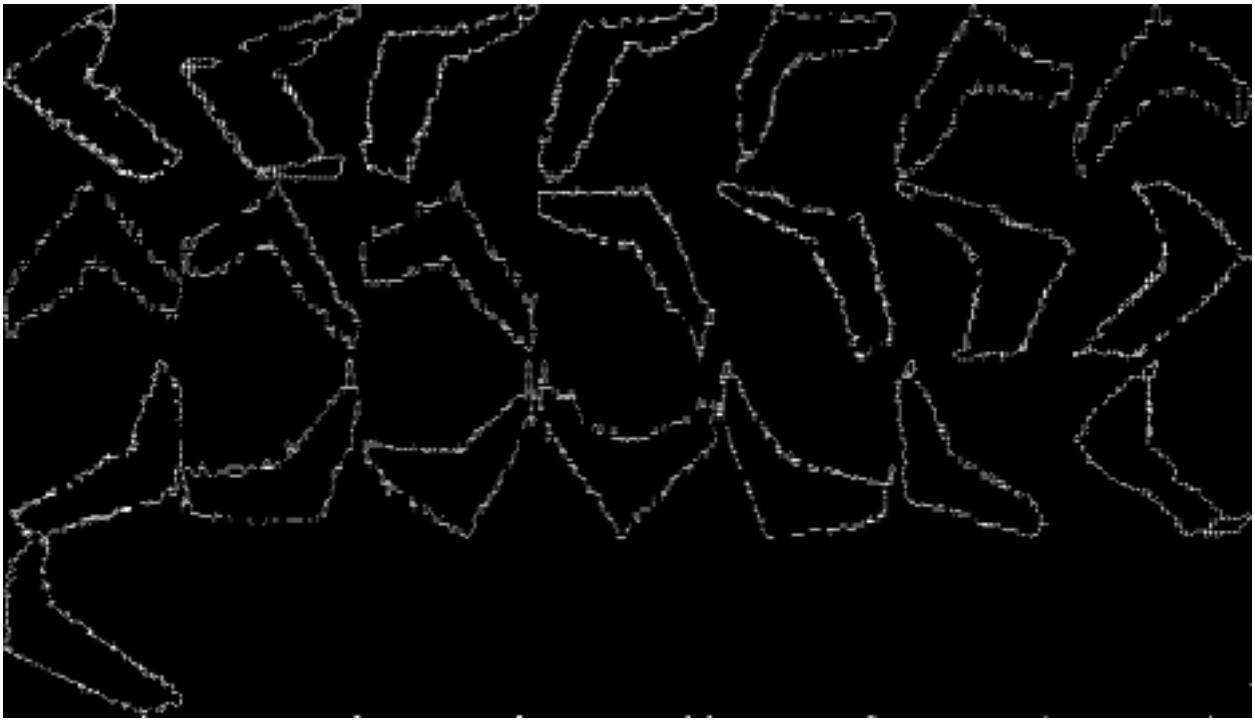

Figure 11: Database used for identifying the airplane.

Figure 12: Final output for the sample image. The plane has been identified and circled.

B Sample images

Figure 13: Sample image #1

Figure 14: Output for sample image #1

Figure 15: Sample image #2

Figure 16: Output for sample image #2

Figure 17: Sample image #3

Figure 18: Output for sample image #3

Figure 19: Sample image #4

Figure 20: Output for sample image #4

Figure 21: An image for which the engine fails to recognize the plane

C Convolution

Convolution (discrete spatial convolution in this case) is a simple method of calculating an output value for each pixel in an image as a linear combination of the values of nearby input pixels [3]. It involves applying a small matrix, or *kernel*, to each pixel of a large image. The kernel is overlayed onto the image at each pixel so that each value in the kernel corresponds to a pixel in the image. The output value of each pixel is then determined by multiplying the surrounding pixels by the corresponding value in the kernel and computing the sum of these values. For example, a simple smoothing algorithm involves averaging the value of each pixel with those of the eight surrounding pixels. The kernel for this convolution is shown in the following equation for finding output value O:

$$O = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{9}$$

For an image I and a kernel K of size $m \times n$, the formula for the value of a pixel can be written as:

$$O(i,j) = \sum_{k=1}^{m}\sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l) \tag{10}$$

Note that this equation applies to kernels which examine pixels down and right from the target pixel. Some kernels, such as the averaging kernel shown above, also examine pixels above and left from the target pixel.

Convolution can be used for many image processing functions by varying the kernel used. To perform convolution on an RGB image, the convolution is simply performed separately for each channel and the results are combined into a single output image. The method for combining results varies based on the particular task being accomplished. Figure 22 illustrates convolution.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Image I

A	B
C	D

Kernel K

1	2	3	4	5	6
7	8xA	9xB	10	11	12
13	14xC	15xD	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Kernel overlayed at pixel 8

Figure 22: Illustration showing convolution of kernel K on image I. In the example given, $O(8) = 8 \times A + 9 \times B + 14 \times C + 15 \times D$.

D Multithreaded architecture illustration

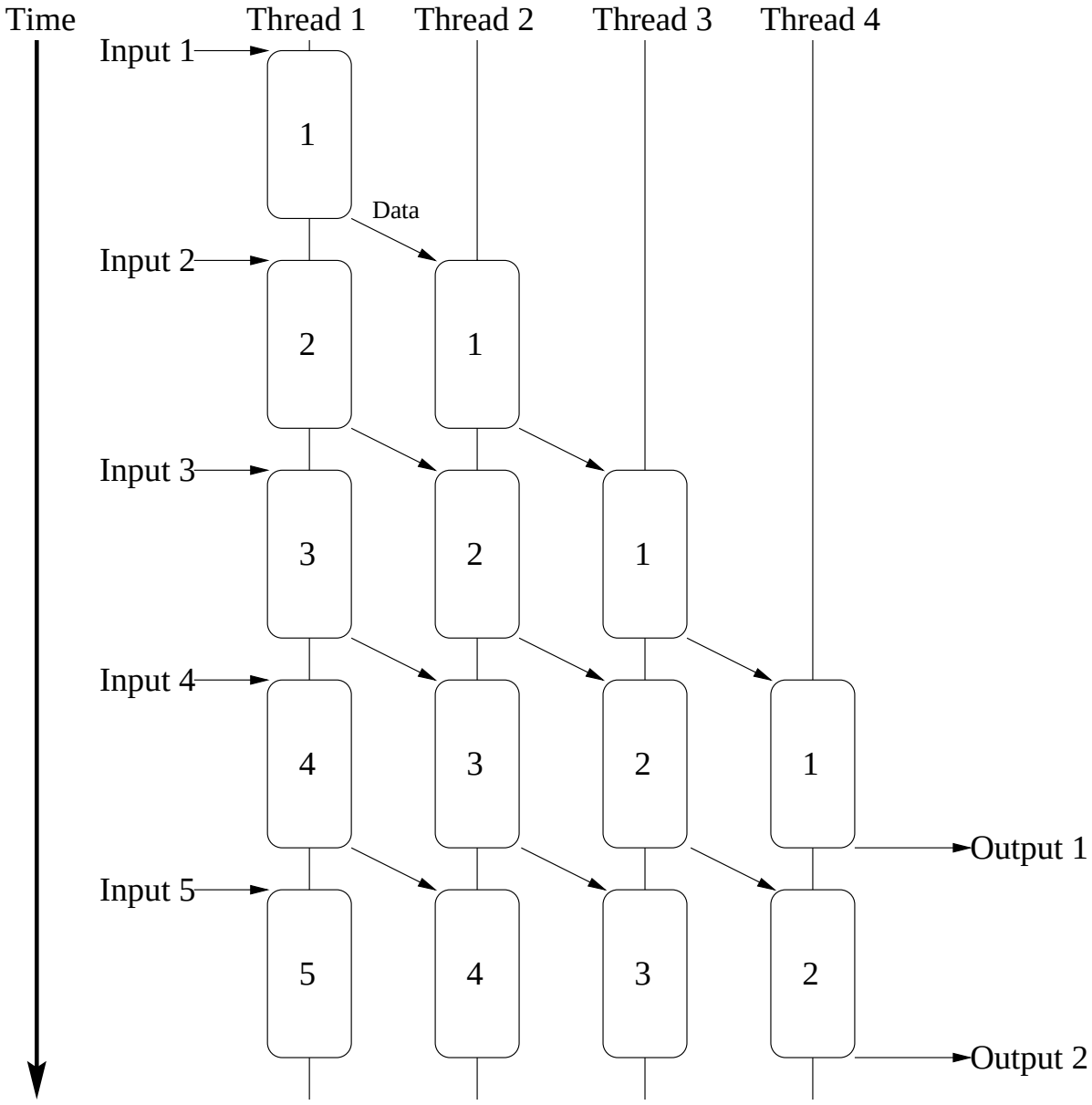

Figure 23: Illustration of the multithreaded architecture