

Object Recognition

on the iPhone

New Mexico

Supercomputing Challenge

Final Report

April 6, 2011

Team Number 34

Desert Academy

Team Members:

Megan Belzner

Matt Rohr

Teachers:

Jocelyne Comstock

John Paterson

Table of Contents

0.0 - Executive Summary	3
1.0 - Introduction	4
<i>1.1 - Background</i>	
<i>1.2 - The Project</i>	
<i>1.3 - Reasoning</i>	
<i>1.4 - Uses</i>	
2.0 - Implementation	7
<i>2.1 - The Algorithm</i>	
<i>2.2 - Assumptions and Limitations</i>	
3.0 - Results and Conclusions	12
<i>3.1 - The Program</i>	
<i>3.2 - Achievements</i>	
<i>3.3 - Conclusion</i>	
<i>3.4 - The Future</i>	
Appendix A - References	16
Appendix B - Pictures	17
Appendix C - Code	20

0.0 – Executive Summary

The field of object recognition, wherein a computer is made able to recognize and define objects, has both a vast array of techniques and a variety of applications. Common techniques include feature detection, text recognition, and template matching through edge detection, and the applications vary from face recognition and detection, to identifying abnormal cells, to reading the labels on objects. However, one thing which object recognition software has yet to achieve is general, wide-ranging recognition of objects of any sort, not just those in a specific class. This project attempts to create a generalized object recognition program for Apple's iPhone, designed for maximum range, portability, and day-to-day usefulness.

The program uses an algorithm of our own design, which is intended to overcome the limitations inherent in other methods. Though we begin with edge detection, the result is not just a map of the lines but rather a picture of the object based on geometric shapes. From this “shape map”, the object can be matched with similar ones in a database of images which has been pre-processed in a similar way. Using shapes rather than lines lessens the need for exact similarities, and with a sufficiently large database the program should not be limited to any specific class of objects.

Though we were not able to implement the full algorithm, we have completed the program to the point where it can recognize basic shapes such as rectangles and acute triangles. Though there are several issues with our current method, such as difficulties identifying shapes with obtuse angles, we have attempted to correct these issues and have come up with ways to solve the problems present. Despite the issues, we have shown that our method is a feasible one and could be extremely successful once these changes are implemented.

1.0 – Introduction

1.1 – Background

Object recognition, a sub-field in the realm of computer vision, is a field of study which attempts to utilize a vast array of different methods with a single end goal: allowing computers to recognize and define objects, whether those in a specific class or more generally. Despite the wide range of methods, most methods have certain similarities. Many methods can be divided into one of two groups: appearance-based or feature-based [5].

Appearance-based methods are based on comparing images to template images. “Local” appearance-based methods focus on finding points or regions of interest characterized by things such as corners or edges. “Global” methods, meanwhile, examine the image as a whole: that is, all pixels are considered rather than centering on a small region [6]. Edge detection is commonly the first step in an appearance-based method, and once the outline of the object in question is found it can be compared with previously “learned” images.

Edge detection algorithms are based on finding discontinuities within the image, typically in image brightness which can be assumed to represent a boundary of some sort. Many edge detection algorithms are based on the first or second derivative of the pixel intensity. As an edge is characterized by a sharp increase or decrease in intensity, in one dimension maximums and minimums in the first derivative can denote an edge. The second derivative can also be used, as a maximum or minimum in the first derivative will always have a value of 0 in the second. This method is thus known as the “zero-crossing” method. The Sobel method of edge detection uses this idea in two dimensions, by using two 3-by-3 arrays which estimate the gradient (that is, the first derivative of intensity) in the x and y directions, respectively. The Laplacian method, meanwhile, uses a single 5-by-5 grid and the second derivative. [2]

Feature-based methods use features specific to objects of a given type to determine whether the object is present or not. For example, in faces the eye region is typically darker than the cheek region. Thus, a common method in face detection is to examine two adjacent

Object Recognition on the iPhone – Team 34

rectangles and determine the difference between pixel intensities [3]. For accurate determination, a large number of such features must be used. So, for example, an algorithm constructed to determine the presence of a mug searches for parallel lines, various geometric shapes, and other things to determine whether or not the object is present in the image [1].

1.2 – The Project

Our goal in this project was to create general object recognition software for Apple's iPhone. We intended to use an edge detection algorithm which would allow us to determine the geometric shapes present in an image, which would enable us to identify the object by comparing it to a database of similarly processed images. We designed our program for maximum range of object recognition and maximum portability, to allow for increased usefulness in everyday situations compared to existing object recognition techniques.

1.3 – Reasoning

We decided to create our program for the iPhone because of the mobility and power it offered. The newest iteration of Apple's iPhone has 512 megabytes of RAM and a 1 GHz processor, which means it is even slightly better than the standard 2003 iMac. The iPhone is about half as powerful as most current netbooks. Though most computers today are far more powerful than the iPhone, the portability of the device is a huge advantage. Object recognition, when used for general purposes, is most likely to be of use in situations where a laptop or desktop computer would be highly impractical. While this project would work equally well on many Android devices, the iPhone was chosen because it can be guaranteed that the newest iPhones have a camera while not all Android devices do.

We wanted to take a general approach with our algorithm, allowing it to recognize many objects rather than just a small set. We used an appearance-based method, though with some notable differences compared to most methods. We decided to create a shape map, rather than stopping at an edge map, as this theoretically allows for matches to be less exact without hurting the accuracy of recognition. In addition, while text recognition has been utilized before to create object recognition software for portable devices, we did not want to take this approach

Object Recognition on the iPhone – Team 34

because this would severely limit the usefulness of the program, as the object would need to be labeled for the program to work.

1.4 - Uses

Mobile object recognition software as we attempted to create could be useful in a variety of situations. One such possible use is image-based search. Using a program such as this, one could photograph an object that perhaps they did not know the name of, then search the web for more information on it. Another use we envisioned is augmenting a vision-impaired person's perception of their surroundings, by providing more information about objects around them than they might be able to gain through other means. Particularly if the object recognition could be made to work in real-time, one could point the iPhone at an object and, by auditory cues from the device, figure out exactly what the object is without having to rely on other senses which may not be exact enough.

2.0 – Implementation

2.1 – The Algorithm

The first step in the algorithm is setting up the picture, which begins by loading the image as a bitmap. In the standard format for images in Objective-C, UIImage, pixel data cannot be easily accessed. Thus, we have used a piece of code [4], the only part not written by us, which initializes images as bitmaps, allowing for easy access to the color data. The next step, once the image is loaded, is to determine the background color by noting the red, green, and blue values of the first pixel in the upper left corner of the image. This data is put into the “bg” array, where `bg[0]` is the red value, `bg[1]` is green, and `bg[2]` is blue. However, at this point the image is still not easy for the computer to analyze. In order to make image analysis simpler, at this point the image is translated into an array of numbers. A 2-dimensional array is created with the same size as the image, and nested for loops are used to loop through the image row by row. Any pixel which is within the allowable margin-of-error range from the background color (which is set at $bg \pm 0x30$) is considered to be part of the background, which is noted by a 0 in the array at that point. Any pixel outside of this range is considered to be part of the object, noted by a 1 in the array. Thus, at this point, a sample image could be as shown in figure 2.1.1, where the indicated point of the star is represented as an array of 1s and 0s.

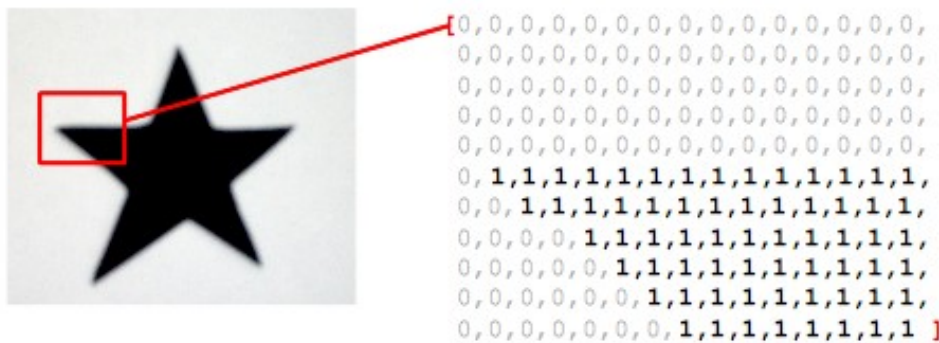


Figure 2.1.1

Once the area covered by the shape is determined, the next step is to find the outline of the shape. Again, for loops are used to check each pixel, though this time the array, rather than the image, is being examined. Any pixel which is part of the background is ignored, but

Object Recognition on the iPhone – Team 34

whenever the algorithm finds a “1” in the array (that is, when it finds an object pixel), it checks the surrounding 4 (directly adjacent) pixels. If any of the four pixels are part of the background, then the pixel in question is on the edge of the shape and is thus considered to be part of the outline. To denote this, the “1” is replaced with a “2” in the array. Figure 2.1.2 shows both the adjacent locations which are examined, as well as a sample outline in the array.

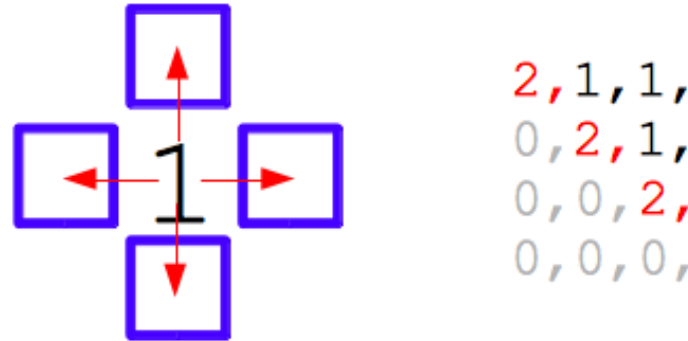


Figure 2.1.2

Following the outline marking, the final step so far is to determine the number of vertices in the shape. For this step, there are two different algorithms which can be used. The first one, which is currently more complete but also more limited, merely searches for simple patterns. The algorithm loops through each outline pixel, building a 3-by-3 viewing window around the pixel in question. The positions of other outline pixels in this 3-by-3 array are noted, and the algorithm checks for one of three patterns in their positions, with the basic patterns shown in figure 2.1.3.



Figure 2.1.3

The other algorithm, while being extremely limited at this time, has more potential to be expanded. The algorithm is based on the idea of using a larger viewing window. At this point an 11-by-11 window is considered, but in the future this could be expanded as needed. The algorithm as it stands now merely looks at the center row and column, the ones including the pixel in question. If the type of pixel (either “2” or “not 2”) is different on either side of the

Object Recognition on the iPhone – Team 34

center pixel, the pixel is considered to be a vertex as shown in figure 2.1.4

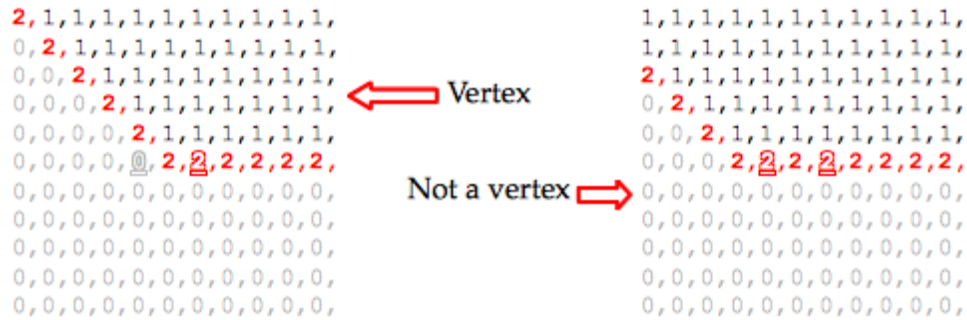


Figure 2.1.4

In either case, once a pixel is found which fits into one of the patterns, the “vertex” variable is increased by 1. Once the entire shape has been examined the shape represented is determined based on the number of vertices. While at this point this is limited to triangles (3 vertices) and quadrilaterals (4 vertices), with any shape with more (or less) vertices being called a polygon, the algorithm should theoretically be expanded to recognize any shape with between 3 and 8 vertices, with any other shape being called a circle. While octagons are fairly common in day to day life, nonagons and above are not prevalent enough to be particularly noted.

The remainder of the algorithm has yet to be implemented, though we have determined what steps the program should take from here. First, it must be noted that of course most objects are not composed of a single shape, but rather many shapes put together. Once vertices are found, the theoretical algorithm would break the shape apart into simple polygons by “cutting” the shape at concave vertices, resulting in a shape map similar to the one in figure 2.1.5.

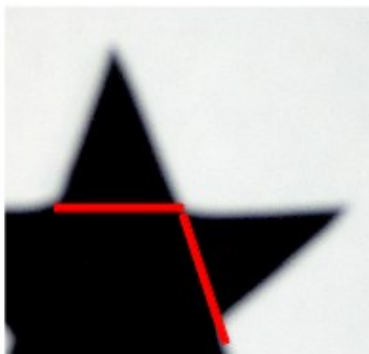


Figure 2.1.5

Based on this the general shape layout of the image could be determined, at which point this “blueprint” of the object would be compared to a database of images which had been pre-processed in a similar manner. Based on the numbers of various shapes present in the image, and possibly other factors if necessary, the set of possible matches could be narrowed down until a good match was found. The tags on this image could be used to name the object, and this information would be relayed back to the user through visual or auditory means.

2.2 – Assumptions and Limitations

At this point, the program is very limited. Although a large part of this is because the algorithm as planned has not been fully implemented, there are still some issues inherent in the method itself. One of the first and most prominent is the limitations of both current vertex finding algorithms. The first one, while it is able to accurately find many triangles and quadrilaterals, it is limited to acute angles which means it cannot easily be expanded to shapes with 5 or more vertices. At this point obtuse angles would be extremely common, but any pattern in a 3-by-3 window which could denote an obtuse angle could also denote a continuation of the line, as shown in figure 2.2.1.



Figure 2.2.1

The second vertex algorithm is even more limited at this point, only really able to identify perfectly straight right angles and thus limited to rectangles. However, this general approach has far more potential to be expanded, as discussed later in section 3.1. Another major issue apparent in the current approach is that it assumes a single color background. Though the shape could theoretically be multi-colored (within reason), the background must be a single color for the program to work. This, of course, would not be the case in a real-life situation. However, this limitation will be extremely difficult to overcome as differentiating between the

Object Recognition on the iPhone – Team 34

object and the background will become increasingly harder.

In addition to these limitations, some assumptions had to be made. The major one is the assumption that the entire object is in the frame. Again, this cannot always be guaranteed in real-life situations. However, this may not be as much of an issue as it seems at first glance. With a sufficiently large comparison database, there are likely to be images with off-center objects. Thus, as long as the shape map can still be drawn, the object may still be recognizable. Another issue could arise if the object happens to be in the upper left corner such that the “background color” is actually the color of the object. However, the outline would still end up in more or less the same place in this case, and corrections could be made for the fact that the image is reversed.

Finally there are those things which appear to be potential problems, but in fact are not. How the object happens to be rotated is one such non-issue. Similarly to the off-center objects note above, with a sufficiently large image database there would be images of similar objects from many perspectives. Another possible issue is similarities in shape maps between different objects. However, if simple appearance of shapes is not enough to determine the object represented, there are many other things which could theoretically be taken into account. On a basic level, the program could consider the relative sizes and positions of the shapes, or the color(s) of the object. If even this is not enough, once multi-colored backgrounds are usable the size of the object compared to other objects in the scene could be considered.

3.0 – Results and Conclusions

3.1 – The Program

At this point, the program is able to recognize most quadrilaterals and triangles (provided they only have acute angles). Though the program cannot work with multi-colored backgrounds it is able to handle some variations in shade (that is, “noise”), and has also been proven to work with images the size that an image from the camera would be. The program is not able to handle imperfect lines, with one such example shown in figure 3.1.1.

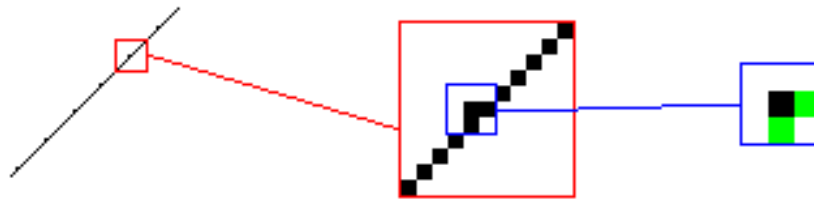


Figure 3.1.1

Though the eye still perceives the first part as a (rather jagged) line, the computer would identify several vertices along the line. The program may also run into difficulties if the background color and object color are too similar, though for the most part it has been successful within the current limitations. The images which it can currently run with can be seen in Appendix B.

In its current version, the program has two different vertex algorithms it could use as noted in section 2.1. Though at this point the first algorithm is the only useful one in most situations, it cannot be easily expanded. With a 3-by-3 grid there is not much else the program could do besides what it does now. Thus, the next step is to widen the viewing field of the vertex algorithm. This is what the second method begins to do. However, at this point one runs into a problem. While the first algorithm was based on finding certain exact patterns, with a larger viewing window the number of possible patterns becomes far too large to use the same method. Two possible methods that could be of use are noting where in the grid most of the object pixels are concentrated, or finding the directions of the lines. Though both of these

Object Recognition on the iPhone – Team 34

approaches have possible issues associated with them, such as handling curves in the first method and determining the exact point of direction change for the second, either method would be more useful than the current algorithm. Thus, although the second method is very limited at this point, this is the one to pursue further.

Finally, another variable to examine is the size of the image. A photograph taken from the camera is 720-by-960 pixels, which means the program has 691200 pixels to loop through. As the size of the image decreases, so will the time it takes to process the image. Timing for the image processing function to run was taken for ten compression levels of the same image. Each level is a certain percent of the dimensions (for example, 10% compression is a 72-by-96 pixel image), so the amount of pixels will decrease more rapidly (the 10% image has only 1% of the pixels in the full size image). Graphs of the timing compared to both compression level and number of pixels are shown in figure 3.1.2.

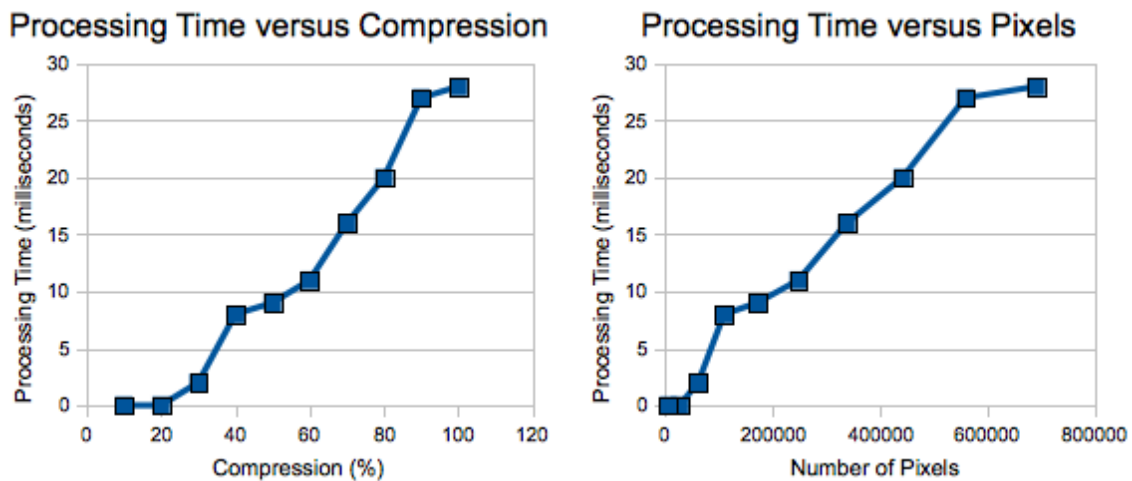


Figure 3.1.2

In both graphs the time decreases more or less linearly, suggesting that the smaller the picture in question is, the better. However, accuracy must be balanced with speed. While the 50% image is still easily recognizable, around 30% the quality of the image begins to interfere with seeing the details. Examples of the compression levels are given in Appendix B. For some objects, the program would not need a high level of detail, but for others this lack of detail could completely throw off the program. In addition, the time for the full size image is still fairly

Object Recognition on the iPhone – Team 34

short. Although there is a noticeable delay after opening the program, this delay is not enough to interfere with the use of the program. It must also be noted that these images were compressed manually beforehand, while in the program a compression algorithm would have to be run first. This would take additional time, with the result that it is probably not even worth it to compress the image.

3.2 – Achievements

Although our program is far from complete, we have experimented with new ideas in the field of object recognition. Although edge matching methods are common in object recognition, we have taken this a step further by using shape matching instead of just leaving the edges alone. If only the lines are considered, the edge maps will be more wildly varied than if shapes are considered, as with shapes it is easier to not have an exact match without hurting the accuracy too much. And while typical edge matching methods do not take into account color or lighting, in theory the final version of our algorithm would take these things into account which would increase the accuracy and range of object recognition tremendously.

In addition, many existing object recognition methods are limited to a relatively small set of objects. Although we have not yet reached this point, our program aims for more general object recognition by using a comprehensive image database and detecting elements of the image which are not limited to only one object or type of objects. We have also tried to make object recognition technology more portable and useful in everyday situations by creating our project for the iPhone.

3.3 – Conclusion

Though our project was not entirely successful, we have proven that the basic idea is a feasible method for object recognition. Though the method would not be perfect, we have shown that it is possible to run these sorts of complex algorithms on mobile devices which, though powerful, do not have nearly the processing power of computers. The current implementation of the method has been successful within the realm it is designed for, and so with more time the algorithm could be made more useful and general object recognition could

be achieved through this method.

3.4 – The Future

The first and most obvious place to continue to is full implementation of the planned algorithm. While at this point the program is only able to recognize extremely basic shapes, with the full algorithm it would be able to recognize more complicated shapes such as stars and possibly even basic objects. However, simply implementing the full algorithm will still leave it somewhat useless in real-life situations as there is still the issue of working with multi-colored backgrounds. This might require a reworking of the algorithm, though things such as focus and depth through shading could be used to determine and differentiate the object in question.

Once the program is at a useful stage in the algorithm, the database must be made useful as well. For a truly comprehensive database a large number of pictures would be needed, more than could be collected by a small group of people. Thus, a possibility for filling the database is putting an option for users to submit photos, tagged by them, to the database. Although this would open up the possibility of misuse of the program by tagging images incorrectly, it would also allow for a wide range of objects to be collected with little effort, as well as taking care of the necessary preprocessing.

Finally, at this point the concept is based on a still image. However, real-time video feed object recognition also offers some interesting possibilities. While the processing power of any mobile device on the market is likely not enough for something like this, real-time object recognition would be far more useful in many situations. In addition, the various sensors on the iPhone (particularly the accelerometer and gyroscope) provide an interesting opportunity here. With video-feed based object recognition, motion of objects could also be a factor in determining the object represented. The iPhone's sensors would allow for perceived motion of the object to be corrected for user motion so that a proper picture of motion is possible.

Appendix A – References

[1] Bakker, H.H.C., R.C. Flemmer and J.W. Howarth, “Feature-Based Object Recognition”, Massey University, New Zealand.
<<http://www.massey.ac.nz/~rcflemme/feature-based%20object%20recognition.pdf>>

[2] Green, Bill, “Edge Detection Tutorial”, Drexel University, Philadelphia PA; 2002.
<<http://www.pages.drexel.edu/~weg22/edge.html>>

[3] “Haar-like Features”, Wikipedia. <http://en.wikipedia.org/wiki/Haar-like_features>



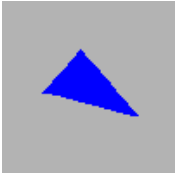


[4] “How to Figure Out 'Color of a Specific Pixel' in an Image?”, Stack Overflow; July 3, 2010. <<http://stackoverflow.com/questions/3172229/how-to-figure-out-color-of-a-specific-pixel-in-an-image>>

[5] “Object Recognition (Computer Vision)”, Wikipedia. <[http://en.wikipedia.org/wiki/Object_recognition_\(computer_vision\)](http://en.wikipedia.org/wiki/Object_recognition_(computer_vision))>

[6] Roth, Peter M and Martin Winter, “Survey of Appearance-Based Methods for Object Recognition”, Graz University of Technology, Austria; January 15, 2008. <http://web.mit.edu/~wingated/www/introductions/appearance_based_methods.pdf>

Appendix B - Pictures

Images to demonstrate the program:

	This picture simply demonstrates recognition of a standard rectangle.
	This picture demonstrates recognition of a standard triangle.
	This picture demonstrates that the program will work with any two colors, not just black and white.
	This picture demonstrates the program when using a picture with "noise", variations in shade of the background. In addition, this triangle contains examples of each of the three types of vertices under the first algorithm.
	This picture is used to demonstrate the second vertex algorithm.

Object Recognition on the iPhone - Team 34

Levels of Compression:



50% Compression

Object Recognition on the iPhone - Team 34



30% Compression



10% Compression

Appendix C – Code

Note: This section only gives the code which we programmed ourselves, it does not include the code from [4] or the standard code in a new iPhone view-based application in Xcode. To use this code, the code from [4] must be placed into the respective files, ImageBitmap.h and ImageBitmap.m.

```
//shapesViewController.h

#import <UIKit/UIKit.h>
#import "ImageBitmap.h"

@interface shapesViewController : UIViewController {
    UIImageView *image;
    ImageBitmap *image_d;
    UILabel *shape;
    Byte *pixel, *pixel2;
    int bg[3], line[11][11], num[11];
    NSInteger check[3][3];
    NSMutableArray *x, *y;
    int height, width, vertex, adj;
    NSNumber *x_0, *x_1, *y_0, *y_1;
}

@end

//shapesViewController.m

#import "shapesViewController.h"

@implementation shapesViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    /*
     initial setup
     sets variables, loads image, and displays info onscreen
     objective-c-style variables are allocated here
     the image is reloaded as a bitmap (for data writing)
     the background color is set from the pixel in the upper left
     */
    self.view.backgroundColor = [UIColor whiteColor];
    shape = [[UILabel alloc] initWithFrame:CGRectMake(50, 50, 500, 500)];
    shape.text = [NSString stringWithFormat:@"Shape:"];
    [self.view addSubview:shape];
    vertex = 0;
    image = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"squ01.png"]];
    height = (int)image.image.size.height;
    width = (int)image.image.size.width;
    image.frame = CGRectMake(0, 0, width, height);
    [self.view addSubview:image];
    image_d = [[ImageBitmap alloc] initWithImage:image.image
    bitmapInfo:kCGImageAlphaNoneSkipLast];
    pixel = [image_d pixelAtX:0 Y:0];    // pixel[0] = red value, pixel[1] = green, pixel[2] =
blue
    bg[0] = (int)pixel[0];
    bg[1] = (int)pixel[1];
}

@end
```

Object Recognition on the iPhone - Team 34

```
bg[2] = (int)pixel[2];
x = [NSMutableArray arrayWithCapacity:2];
y = [NSMutableArray arrayWithCapacity:2];

/*
writing data
loops through each individual pixel
if a pixel is within the 0x30 margin of error from the background color,
the corresponding point in the array is set at 0 to denote "background"
otherwise the array slot is set as 1 to denote "object"
*/
int data[height][width];
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        pixel = [image_d pixelAtX:j Y:i];
        if ((pixel[0]>=bg[0]-0x30 && pixel[0]<=bg[0]+0x30) && (pixel[1]>=bg[1]-0x30 &&
pixel[1]<=bg[1]+0x30) && (pixel[2]>=bg[2]-0x30 && pixel[2]<=bg[2]+0x30))
            data[i][j] = 0;
        else {
            data[i][j] = 1;
        }
    }
}

/*
finding outlines
loops through each individual slot in the array (each pixel)
for any pixel which is part of the object (==1),
it checks the directly adjacent (surrounding 4) spaces
if any are the background color (==0)
sets pixel to 2 to denote "outline"
*/
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        if (data[i][j]==1) {
            if (data[i-1][j]==0 || data[i+1][j]==0 || data[i][j-1]==0 || data[i]
[j+1]==0)
                data[i][j] = 2;
        }
    }
}

/*
finding vertices
loops through each pixel
for any pixel which is an outline (==2),
it checks to see if the pixel is a vertex
*/
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        /*
        algorithm 1
        this algorithm constructs a 3*3 array where any outline is denoted by 1
        and any non-outline is denoted by 0
        it notes the positions of the adjacent pixels to the center pixel
        if these (theoretically two) pixels are themselves adjacent
        or in the same (non-center) column or row,
        the center pixel is considered as a vertex
        which adds one to the vertex variable
        so far only for acute angles
        and thus only useful for triangles and rectangles
        */
        if (data[i][j]==2) {
            if (data[i-1][j-1]==2)
                check[0][0] = 1;
        }
    }
}
}
```

Object Recognition on the iPhone – Team 34

```
        else
            check[0][0] = 0;
        if (data[i-1][j]==2)
            check[0][1] = 1;
        else
            check[0][1] = 0;
        if (data[i-1][j+1]==2)
            check[0][2] = 1;
        else
            check[0][2] = 0;
        if (data[i][j-1]==2)
            check[1][0] = 1;
        else
            check[1][0] = 0;
        check[1][1] = 0;
        if (data[i][j+1]==2)
            check[1][2] = 1;
        else
            check[1][2] = 0;
        if (data[i+1][j-1]==2)
            check[2][0] = 1;
        else
            check[2][0] = 0;
        if (data[i+1][j]==2)
            check[2][1] = 1;
        else
            check[2][1] = 0;
        if (data[i+1][j+1]==2)
            check[2][2] = 1;
        else
            check[2][2] = 0;
        adj = 0;
        for (int k = 0; k < 3; k++) {
            for (int l = 0; l < 3; l++) {
                if (check[k][l]==1) {
                    adj++;
                    [x insertObject:[NSNumber numberWithInt:1]
atIndex:adj-1];
                    [y insertObject:[NSNumber numberWithInt:k]
atIndex:adj-1];
                }
            }
        }
        if (adj>1) {
            x_0 = [x objectAtIndex:0];
            x_1 = [x objectAtIndex:1];
            y_0 = [y objectAtIndex:0];
            y_1 = [y objectAtIndex:1];
        }
        if (adj==2) {
            if (((x_0==[NSNumber numberWithInt:0] && x_1==[NSNumber
numberWithInt:1]) || (x_0==[NSNumber numberWithInt:1] && x_1==[NSNumber
numberWithInt:2])) &&
y_0==y_1) || (((y_0==[NSNumber numberWithInt:0] && y_1==[NSNumber
numberWithInt:1]) ||
(y_0==[NSNumber numberWithInt:1] && y_1==[NSNumber numberWithInt:2])) && x_0==x_1)) {
                vertex++;
            }
            else if ((x_0==[NSNumber numberWithInt:1] && x_1==[NSNumber
numberWithInt:0] && y_0==[NSNumber numberWithInt:0] && y_1==[NSNumber
numberWithInt:1]) ||
(x_0==[NSNumber numberWithInt:1] && x_1==[NSNumber numberWithInt:2] && y_0==[NSNumber
numberWithInt:0] && y_1==[NSNumber numberWithInt:1]) || (x_0==[NSNumber
numberWithInt:0] &&
x_1==[NSNumber numberWithInt:1] && y_0==[NSNumber numberWithInt:1] && y_1==[NSNumber
numberWithInt:2]) || (x_0==[NSNumber numberWithInt:2] && x_1==[NSNumber
numberWithInt:1] &&
y_0==[NSNumber numberWithInt:1] && y_1==[NSNumber numberWithInt:2])) {
                vertex++;
            }
        }
    }
}
```


Object Recognition on the iPhone - Team 34

```
        else if ((x_0!=[NSNumber numberWithInt:1] && x_0==x_1) || (y_0!
=[NSNumber numberWithInt:1] && y_0==y_1)) {
            vertex++;
        }
    }
}
/*
algorithm 2
this algorithm copies an 11*11 portion centered around each outline pixel
at this point it merely examines the center row and center column
if either one has all outline pixels to one side of the center,
it's considered a vertex
this algorithm can only identify non-rotated rectangles
though the general approach of
examining the placement of outlines in an 11*11 square
has potential to be expanded, far moreso than algorithm 1
*/
/*if (data[i][j]==2) {
    for (int m = 0; m < 11; m++) {
        for (int n = 0; n < 11; n++) {
            line[m][n] = data[i+m-5][j+n-5];
        }
    }
    for (int k = 0; k < 11; k++) {
        if (line[5][k]==2)
            num[k] = 1;
        else
            num[k] = 0;
    }
    if (num[4]!=num[6])
        vertex++;
    else {
        for (int l = 0; l < 11; l++) {
            if (line[l][5]==2)
                num[l] = 1;
            else
                num[l] = 0;
        }
        if (num[4]!=num[6])
            vertex++;
    }
}*/
}
}
if (vertex==3)
    shape.text = [NSString stringWithFormat:@"Shape: Triangle"];
else if (vertex==4)
    shape.text = [NSString stringWithFormat:@"Shape: Rectangle"];
else
    shape.text = [NSString stringWithFormat:@"Shape: Polygon"];
}

- (void) didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void) viewDidUnload {
}

- (void) dealloc {
    [super dealloc];
}

@end
```