Digital Image Object Extraction

GRADUATE PROJECT TECHNICAL REPORT

Submitted to the Faculty of
The Department of Computing and Mathematical Sciences
Texas A&M University-Corpus Christi
Corpus Christi, Texas

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

By

Dennis Ma
Summer 2003

Committee Members

Dr. Dulal Chandra Kar
Committee Chairperson

Dr. Mario Garcia
Committee Member

Dr. John Fernandez
Committee Member
# Table of Content

<table>
<thead>
<tr>
<th>Acknowledgement</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Digital Image Object Extraction</td>
<td>3</td>
</tr>
<tr>
<td>2.1. Overview</td>
<td>3</td>
</tr>
<tr>
<td>2.2. Graphical User Interface</td>
<td>5</td>
</tr>
<tr>
<td>2.3. Image Acquisition</td>
<td>9</td>
</tr>
<tr>
<td>2.4. Image Format</td>
<td>10</td>
</tr>
<tr>
<td>2.5. BMP Header Information</td>
<td>12</td>
</tr>
<tr>
<td>3. System Design</td>
<td>15</td>
</tr>
<tr>
<td>3.1. System Used in the Project</td>
<td>15</td>
</tr>
<tr>
<td>3.2. System Requirement</td>
<td>16</td>
</tr>
<tr>
<td>3.3. Edge Detection Methods</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1. Sobel Method</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2. Sobel Explanation</td>
<td>23</td>
</tr>
<tr>
<td>3.4. Problem with Sobel Edge Detection</td>
<td>24</td>
</tr>
<tr>
<td>3.5. Highlighting Edges</td>
<td>26</td>
</tr>
<tr>
<td>3.6. Filling Highlighted Edges</td>
<td>31</td>
</tr>
<tr>
<td>3.7. Extraction</td>
<td>32</td>
</tr>
<tr>
<td>4. Evaluation and Result</td>
<td>34</td>
</tr>
<tr>
<td>4.1. Meeting the Objectives</td>
<td>34</td>
</tr>
<tr>
<td>4.2. Testing</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1. Input File Format</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2. Image Size</td>
<td>35</td>
</tr>
</tbody>
</table>
4.2.3. Variety of Objects in Images.................................................................36
4.2.4. Color Depth.........................................................................................38
4.2.5. Dark Background and Light Background........................................39
4.2.6. File Opening and Saving.................................................................40
4.2.7. Images Taken by Digital Camera..................................................40

5. Future Work............................................................................................46

6. Conclusion...............................................................................................47

Reference
ACKNOWLEDGEMENT

I am grateful for the enlightenment given by Dr. Kar who encouraged me to conduct experiment on this project. I take this opportunity to express my gratitude and deep regard to his guidance and mentoring. Sincere appreciation is extended to Dr. Fernandez for his help on editing my project and his personal concern for my career. I am also grateful to Dr. Garcia for being one of my committee members.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3</td>
<td>5</td>
</tr>
<tr>
<td>Figure 4</td>
<td>6</td>
</tr>
<tr>
<td>Figure 5</td>
<td>7</td>
</tr>
<tr>
<td>Figure 6</td>
<td>9</td>
</tr>
<tr>
<td>Figure 7</td>
<td>11</td>
</tr>
<tr>
<td>Figure 8</td>
<td>17</td>
</tr>
<tr>
<td>Figure 9</td>
<td>17</td>
</tr>
<tr>
<td>Figure 10</td>
<td>18</td>
</tr>
<tr>
<td>Figure 11</td>
<td>19</td>
</tr>
<tr>
<td>Figure 12</td>
<td>20</td>
</tr>
<tr>
<td>Figure 13</td>
<td>25</td>
</tr>
<tr>
<td>Figure 14</td>
<td>26</td>
</tr>
<tr>
<td>Figure 15</td>
<td>27</td>
</tr>
<tr>
<td>Figure 16</td>
<td>29</td>
</tr>
<tr>
<td>Figure 17</td>
<td>30</td>
</tr>
<tr>
<td>Figure 18</td>
<td>31</td>
</tr>
<tr>
<td>Figure 19</td>
<td>32</td>
</tr>
<tr>
<td>Figure 20</td>
<td>32</td>
</tr>
<tr>
<td>Figure 21</td>
<td>33</td>
</tr>
<tr>
<td>Figure 22</td>
<td>37</td>
</tr>
<tr>
<td>Figure 23</td>
<td>39</td>
</tr>
<tr>
<td>Figure 24</td>
<td>40</td>
</tr>
<tr>
<td>Figure 25</td>
<td>41</td>
</tr>
<tr>
<td>Figure 26</td>
<td>42</td>
</tr>
<tr>
<td>Figure 27</td>
<td>43</td>
</tr>
<tr>
<td>Figure 28</td>
<td>43</td>
</tr>
<tr>
<td>Figure 29</td>
<td>44</td>
</tr>
<tr>
<td>Figure 30</td>
<td>45</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>13</td>
</tr>
<tr>
<td>Table 2</td>
<td>21</td>
</tr>
<tr>
<td>Table 3</td>
<td>28</td>
</tr>
</tbody>
</table>
ABSTRACT

This project is the design and implementation of a digital image-processing program that can eliminate unwanted background area within a digital image. This process allows the extraction of desired objects in the image file. The extraction of desired objects can allow users to manipulate the image a lot easier for image editing. This program provides a GUI (graphical user interface) environment for ease of use. Users will be able to see the original image and extracted object images side by side before any saving to a separate file is applied. This will guarantee users reaching their desired objects.
1. Introduction and Background

Much of image processing done today requires a lot of picture combining steps, such as taking objects of one picture and combining with another image, e.g., the background image, to create a new one. However, object extraction is a tedious process. First, users need to outline the desired object. The difficulty in that most objects have irregular curve and shape. Then, users need to make sure all the pixels that are considered to be the object are inside the outline they have drawn; and pixels that are not considered to be the object outside of outline. Finally, the user can cut the object from the image and place it in a new file. A lot of disadvantages can be noticed using the traditional way of object extraction. Such as,

- **Time needed to outline the edges of the object**
- **Precision of outlining the object**
- **Unable to have a clear picture of the result before the process**

As technology improves day after day, doing things faster and with more precision is much more important. With the program designed here, a lot of great things can be done with great ease. With this program, Digital Image Object Extraction, users can receive the following benefits:

- **Reduce time needed to extract an object from an image**
- **Great precision on outlining an edge**
- **Selection of desired objects**
- **Before and after product clearly shown before action is taken**
- **Easy to use GUI**
There are many commercial image-editing programs that provide the capability for object extraction. Adobe Photoshop is one of most powerful among them. It is a software package that provides filters and image editing functions to the user. Many of professional digital image editors use it on much of their work. However, object extraction remains a tedious work on irregularly shaped objects. Even though Adobe Photoshop does provide a fast edge detection tool, most of the time, the edges found are neither precise nor complete. Furthermore, the Adobe Photoshop requires a lot of computer resource and storage to be executed.

Digital Image Object Extraction is designed to accomplish a precise and fast digital image object extraction. When a program focused on doing one thing and one thing only, it provides the user with great stability and a friendly interface. With less tedious image extraction work needing to be done, more users would enjoy digital image editing and it would be more fun and less frustrating.
2. Digital Image Object Extraction

2.1 Overview

In this project, a program that detects the surrounding edges of relevant objects was designed and implemented. After the edges of a relevant object are found, the program first highlights the edges found in the original image. Then it proceeds to white out the useless areas outside of the edges and show the processed image next to the original image. In the case where there are multiple objects found in the given image, the user can have the option to see the other processed image result derived by the program and optionally save each object to a specific file.

Here are the pictures that illustrate how the program works. First, it takes the image in Figure 1(a), applies the edge detection algorithm to derive the edges dashed in Figure 1(b). After knowing the edges of the desired object, a processed image can be acquired as shown in Figure 1(c).

![Image](image-url)  
Figure 1. Object Extraction
In a more complex image that consists of more than one object, the user is granted the option of choosing which object to save. This situation can be illustrated in the following picture (see Figure 2), which was originally derived from Figure 1. Now, Figure 2(a) is the input image. After edge detection is applied, all objects’ edges are highlighted, as shown in the dashed line in Figure 2(b). In this case, there are three objects in the image, the program grants the user the option to choose which object to save which can be the star, the circle, or the cross (see Figure 3(c)).

(a) Starting Image  (b) Possible Objects  (c) Possible extractions

Figure 2. Image with multiple objects extraction
2.2 Graphical User Interface

The program provides the user an easy to use graphical user interface (GUI) (see Figure 3). The window shown below is the graphical user interface display once the program is executed. The GUI was developed under Microsoft Visual Studio using Visual C++ MFC.

Figure 3. Graphical User Interface (GUI)

In the following table, a detail list of the buttons and their function are explained.
**OPEN FILE** button – This button works as a file opening interface. At the push of the button, the program invokes a Microsoft File Opening Dialog (see **Figure 4**) providing the user the option to choose a file to open. The main purpose of this function is to load an image into the program for processing. This button can be accessed anytime when the program is in its idle state.

![Figure 4. Opening a file for loading.](image)

**Path of Loaded Image** – Once a selection of an input image has been made, this area indicates the file path being loaded for processing.
**EXIT, CLOSE, or X** button – The EXIT, the CLOSE, or the X buttons can be used to close or terminate the program.

**SAVE** button – The Save button works as a file saving interface. Upon pushing the button, the user can invoke the Microsoft File Saving dialog box (see **Figure 5**). The main purpose of this function is to save a processed image in a desired location. This button can be accessed only when there is a processed image present in the window.

![Save the Extraction Image As](image)

**Figure 5.** Save the image to a desired destination.
**PROCESS** button – The button works as an image data retrieving interface. By pushing it, a user can make the program receive the input image file and decode it to workable form, then update the input image with the highlighted edge, and then show the first object extracted in the processed image area. It can be accessed any time when there is an input file loaded in the program and the program is at its idle state.

**Original Image Area** – This is where the image is shown when an input image is loaded.

**Processed Image Area** – This is where the object image is shown when the image has been processed. If the NEXT button is pushed, the program shows the next object image available in the area, replacing the previous object image.

**HELP** – This is where a user can get an instant help on the program itself in a popup window (see Figure 6).
2.3 Image Acquisition

Image Acquisition is the first step of the process. In order to run this program, an input image is required. There are two elements required to acquire digital images. The first is a physical device that is sensitive to a band in the electromagnetic energy spectrum (such as the x-ray, ultraviolet, visible, or infrared bands) and that produces an electrical signal output proportional to the level of energy sensed; an example of such would be an image scanner. The second, called a digitizer, is a device for converting the electrical output of the physical sensing device into digital form. One good example of such a device is a digital camera [Gonzalez 1992]. To test this program, images were gathered from both a digital camera and intentionally drawn by Microsoft Paint. Images are also manipulated to different file types and different color intensity and resolutions.
2.4 Image Format

A digital image is a two-dimensional array of small square regions known as pixels. In the case of a monochrome image, the brightness of each pixel is represented by a numeric value. Gray-scale images typically contain values in the range from 0 to 255, with 0 representing black, 255 representing white and values in between representing shades of gray.

There are many types of digital image file formats available. However, in the project, only the 8-bit bitmap (BMP) file format is considered. For any incorrect input file format, the program only produce an error message and no output result are shown. The reason for choosing the bitmap format as the designated image format type is because of its fast reading of each pixel value, and also for its compatibility to the Windows operating system.

BMP files always contain RGB (Red Green Blue) data. The following show how many colors can be generated with various bits schemes.

- 1-bit: 2 colors (monochrome)
- 4-bit: 16 colors
- 8-bit: 256 colors
- 24-bit: 16777216 colors

Here is an example of an 8-bit picture blown up to show the underlying pixel values (see Figure 7).
Figure 7. A sample pixel value representation on a 8 bit, monochrome image

There are 256 possible colors ($2^8$) that can be stored in an 8-bit image with 255 being the largest value for a pixel. Likewise, a 24-bit true color BMP has a possible 16 million colors ($2^{24}$). In a 24-bit BMP image, a pixel represents a RGB (Red, Green, Blue) data value. A pixel has three 8-bit colors in it each having an intensity value between 0-255. A pixel's RGB data value shows how much Red, Green and Blue are in that particular pixel. So a pixel with a data value of (255, 0, 0) is equivalent to (Red=255, Green=0, and Blue=0), or Red! And the composite of the RGB color values produces that pixel's actual color. For example, we know that red and green make yellow. Therefore we would need all red, all
green and no blue. Being 255 is the maximum for each color; one would need an RGB data value of (255, 255, 0) to achieve an accurate representation of yellow. [Tanimoto 1997]

### 2.5 BMP Header Information

The Table 1 below shows how the pixel data is stored from the first byte to the last. This file format is the MS-Windows standard format. It holds black and white, 16-color, 256-color and true-color images. The palletized 16-color and 256-color images may be compressed via run-length encoding.
Table 1. BMP header information

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header</strong></td>
<td></td>
<td>Windows Structure: BITMAPFILEHEADER</td>
</tr>
<tr>
<td>Signature</td>
<td>14 bytes</td>
<td>‘BM’</td>
</tr>
<tr>
<td>FileSize</td>
<td>2 bytes</td>
<td>File size in bytes</td>
</tr>
<tr>
<td>Reserved</td>
<td>4 bytes</td>
<td>Unused (default = 0)</td>
</tr>
<tr>
<td>DataOffset</td>
<td>4 bytes</td>
<td>File offset to Raster Data</td>
</tr>
<tr>
<td><strong>InfoHeader</strong></td>
<td>40 bytes</td>
<td>Windows Structure: BITMAPINFOHEADER</td>
</tr>
<tr>
<td>Size</td>
<td>4 bytes</td>
<td>Size of InfoHeader = 40</td>
</tr>
<tr>
<td>Width</td>
<td>4 bytes</td>
<td>Bitmap Width</td>
</tr>
<tr>
<td>Height</td>
<td>4 bytes</td>
<td>Bitmap Height</td>
</tr>
<tr>
<td>Planes</td>
<td>2 bytes</td>
<td>Number of Planes (default = 1)</td>
</tr>
<tr>
<td>BitCount</td>
<td>2 bytes</td>
<td>Bits per Pixel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = monochrome palette. NumColors = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = 4bit paledized. NumColors = 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 = 8bit paledized. NumColors = 256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 = 16bit RGB. NumColors = 65536</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24 = 24bit RGB. NumColors = 16M</td>
</tr>
<tr>
<td>Compression</td>
<td>4 bytes</td>
<td>Type of Compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = BI_RGB no compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = BI_RLE8 8bit RLE encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = BI_RLE4 4bit RLE encoding</td>
</tr>
<tr>
<td>Image Size</td>
<td>4 bytes</td>
<td>(compressed) Size of Image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is valid to set this = 0, if Compression = 0</td>
</tr>
<tr>
<td>XpixelsPerM</td>
<td>4 bytes</td>
<td>Horizontal resolution: Pixels/meter</td>
</tr>
<tr>
<td>YpixelsPerM</td>
<td>4 bytes</td>
<td>Vertical resolution: Pixels/meter</td>
</tr>
<tr>
<td>Colors Used</td>
<td>4 bytes</td>
<td>Number of actually used colors</td>
</tr>
<tr>
<td>ColorsImportant</td>
<td>4 bytes</td>
<td>Number of important colors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = all</td>
</tr>
<tr>
<td><strong>ColorTable</strong></td>
<td>4 x NumColors bytes</td>
<td>Present only if Info.BitsPerPixel &lt;= 8 colors should be ordered by importance</td>
</tr>
<tr>
<td>Red</td>
<td>1 byte</td>
<td>Red intensity</td>
</tr>
<tr>
<td>Green</td>
<td>1 byte</td>
<td>Green intensity</td>
</tr>
<tr>
<td>Blue</td>
<td>1 byte</td>
<td>Blue intensity</td>
</tr>
<tr>
<td>Reserved</td>
<td>1 byte</td>
<td>Unused (default = 0)</td>
</tr>
<tr>
<td>Repeated</td>
<td></td>
<td>Numcolors times</td>
</tr>
<tr>
<td><strong>Raster Data</strong></td>
<td>Info.ImageSize bytes</td>
<td>The pixel data</td>
</tr>
</tbody>
</table>

13
The first 14 bytes are dedicated to the header information of the BMP. The next 40 bytes are dedicated towards the info header, where one can retrieve such characteristics as width, height, file size, and number of colors used. Next, is the color table, which is 4 x (number of colors used). So for an 8-bit grayscale image, the number of colors would be 256. And the color table would be 4 x 256 bytes = 1024 bytes. The last byte of data in a BMP file is the pixel data, or raster data. The raster data starts at byte 54 (header + info header) + 4 x number of colors (color table). For an 8-bit grayscale image, the raster data would start at byte 54 + 1024 = 1078. The size of the raster data is (width x height)-1 bytes. Therefore, a 100 row by 100 column 8-bit grayscale image would have (100 x 100)-1 = 9,999 bytes of raster data starting at byte 1078 and continuing to the end of the BMP file. [Miano 1999]
3. System Design

3.1 System and Software Used in the Project

This project was implemented on a personal computer, which is about three years old. The hardware and software involved are as follows.

- Intel Pentium III 600MHz
- Windows XP
- Visual Studio 6.0
- Borland Turbo C++ 4.5
- CYGWIN

The main program is written in C and VC++. Some parts of the source code were compiled and tested under Borland Turbo C++ and some under CYGWIN. CYGWIN is a Linux-like environment for Windows. It consists of two parts:

- DLL (cygwin1.dll) which acts as a Linux emulation layer providing substantial Linux API functionality.
- Collection of tools, which provide Linux look and feel.

They were all later put together in VC++. Some testing of images was made possible using Adobe Photoshop 6.0.
3.2 System Required

This project was implemented under the Windows environment. To run the program, a computer must be able to run Windows 95/98/XP operating system and have at least 10 MB of free hard drive space and 32MB of memory. However, larger image files require larger free storage space.

3.3 Edge Detection Methods

In this project, two methods of detecting edges of an image file were tested. One is called Sobel method, another is Laplace method. Edges characterize boundaries and therefore, edge detection is a problem of fundamental importance in image processing. Edges in images are areas with strong intensity contrasts – a jump in intensity from one pixel to the next. Edge detection can significantly reduce the amount of data in an image by filtering out useless information in the image while preserving the important structural properties in it. There are many ways to perform edge detection. However, the majority of the methods may be grouped into two categories, the gradient method and the Laplacian method. A gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. The Laplacian method searches for zero crossings in the second derivative of the image to find edges. An edge has the one-dimensional shape of a ramp and calculating the derivative of the image can highlight its location. Let us suppose we have the following signal, with an edge shown by the jump in intensity below in Figure 8. [Green 2002]
If we take the gradient of this signal (which, in one dimension, is just the first derivative with respect to $t$), we get the following graph (see Figure 9).

Clearly, the derivative shows a maximum located at the center of the edge in the original signal. This method of locating an edge is the characteristic of the “gradient filter” family of edge detection filters, which includes the Sobel method. A pixel location is declared an edge location if the value of the gradient exceeds some threshold. As mentioned before, edge pixels that have higher pixel intensity values than those surrounding it. So once a threshold is set, one can compare the gradient value to the threshold value and detect an edge whenever the threshold is exceeded. Furthermore, when the first derivative is at a maximum, the second derivative is zero. As a result, another alternative to finding the location of an edge is to locate the zeros in the second derivative.
This method is known as the Laplacian method and the second derivative of the signal is shown below in **Figure 10.** [Petrou 1999]

![Figure 10. Laplacian of the signal](image)

Figure 10. Laplacian of the signal
Before deciding which method to use for this particular project, both the Sobel edge detection and the Lapalace edge detection were tested. Here are the two of the example images after the process (Figure 11 and Figure 12).

Figure 11. (a) Original Picture, (b) Sobel Edge Detection Algorithm, (c) Lapalacian Edge Detection Algorithm.
Figure 12. (a) Original Picture, (b) Sobel Edge Detection Algorithm, (c) Laplacian Edge Detection Algorithm.
Let us look at some major advantage and disadvantage points of each edge detection algorithm here.

Table 2. Features of Laplace and Sobel edge detection algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Laplace</th>
<th>Sobel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Noise Level</strong></td>
<td>More sensitive to noise level. As we can see in figures above.</td>
<td>Better than Lapalce edge detection method. Not only does it give a cleaner look, it also smoothes out the edges.</td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>Laplace edge detection shows a very complete edge of an object.</td>
<td>Weak edges depending on the background of an image.</td>
</tr>
</tbody>
</table>

After comparing both of the methods, it is clear to see that Lapalace edge detection methods were found to be too sensitive to noise level, which might cause some problem when the program tries to find a valid object edge. Therefore, the Sobel edge detection method was chosen as the primary method of detecting edges for this project since the algorithm produces a clean, smooth image. As for the weak edges problem involved in the Sobel edge detection algorithm, an inverse-and-combine method was used, which is discussed in a later section of this report.

3.3.1 Sobel Method

Based on the one-dimensional analysis, the theory can be extracted to two-dimensions as long as there is an accurate approximation to calculate the derivative of a two-dimensional image. The Sobel operator performs a 2-D spatial gradient measurement on an image. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The Sobel edge detector uses a pair of
3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time. The actual Sobel masks are shown below: [Fisher 1994]

\[
\begin{array}{ccc}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{array}
\quad
\begin{array}{ccc}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{array}
\]

\(G_x\) \quad \(G_y\)

The magnitude of the gradient is then calculated using the formula:

\[ |G| = \sqrt{G_x^2 + G_y^2} \]

An approximate magnitude can be calculated using:

\[ |G| = |G_x| + |G_y| \]

The \(G_x\) mask highlights the edges in the horizontal direction while the \(G_y\) mask highlights the edges in the vertical direction. After the process, the resulting output can be used to detect edges in both directions.
3.3.2 The Sobel Explanation

The mask is slid over an area of the input image. The mask changes the center pixel's value after convolution operation on the image sequent, and then the Sobel image is shifted one pixel to the right and the process continues to the right until it reaches the end of the row. It then starts at the beginning of the next row. The example below shows the mask being slid over the top portion of the input image represented by the thicker outline. The formula shows how a particular pixel in the output image would be calculated. The center of the mask is placed over the pixel to be manipulated in the image. The i and j values (\( a_{ij} \) or \( m_{ij} \)) are used to move the file pointer so one can multiply, for example, pixel (\( a_{22} \)) by the corresponding mask value (\( m_{22} \)). It is important to notice that pixels in the first and last rows, as well as the first and last columns cannot be manipulated by a 3x3 mask. This is because when placing the center of the mask over a pixel in the first row (for example), the mask would be outside the image boundaries. [Green 2002]

\[
b_{22} = (a_{11} \times m_{11}) + (a_{12} \times m_{12}) + (a_{13} \times m_{13}) + (a_{21} \times m_{21}) + (a_{22} \times m_{22}) + (a_{23} \times m_{23}) + (a_{31} \times m_{31}) + (a_{32} \times m_{32}) + (a_{33} \times m_{33})
\]
3.4 Proceeding with the Sobel Edge Detection

Due to the nature of the Sobel edge detection algorithm, a particular direction of edges of the object seems to have weak response to this method. The images tend to have only two-thirds of clear edges shown on the output. Depending on the background and object color contrast, for images with darker background, lower right edges would stand out much clear. For images with lighter background compared to the object, upper left edges would stand out much clear. In order to get complete edges of an object (see Figure 13), first the original image (Figure 13a) has to go through the Sobel edge detection algorithm process, which detects the edges of either upper left or lower right corner. Afterward, an inverse of original image has to be created (see Figure 13c), and it has to go through the Sobel algorithm process as well to cover the missing edges of the original image. Finally, these two processed images are combined to synthesize and complete the process of edge detection (see Figure 14).
To inverse an image, each pixel value is to be subtracted from the maximum value of grayscale which is 255. In the case when a pixel is 0 (black), $255 - 0 = 255$ which is the value of a white pixel. On the other hand, when a pixel is 255 (white), $255 - 255 = 0$, which is the value of a black pixel. In the combination of edge detected images, the example
given above was setup in a way that when a pixel was not 0 after the Sobel edge detection, is considered as a valid edge pixel. However, under image with non-uniform background, assuming every non-white pixel is an edge pixel would be a problem. To solve this problem, setting a value that is closer to black value as an edge would work nicely. Further detail provided later in the evaluations and testing part of this report.

3.5 Highlighting Edges

After edges of the objects in an image have been detected, the next step is to find the first object that is closest to the origin point (row 0, column 0). Before anything can be accomplished, a layout of image in arrays is created. The program first retrieves the image information from the machine memory, and then makes a duplicate copy into an array. For example, in a 10 x 10 pixels image (see Figure 15), the array would look like the way as shown in Table 3.
Figure 15. Example of pixel Image.
After pixel data value have been placed in the array, a connecting algorithm is applied that traces the most outer pixel of the edges until it returns to the starting point (see Figure 16). If it is unable to reach back to the starting point, it would not be considered as an “object”.

In this project, we assume that the pixels are using eight-connected neighborhood, which means there are eight directions a pixel can go from itself (see Figure 17), any of the eight direction neighbors will be considered a connected pixel. With the starting point always at bottom left, the loop connecting method used is to go with direction 6 first and check for un-marked edge pixels that guarantees the most outer pixel of the edges. If 6 is not viable, then move on to 7, or 8 or 1, 2, 3, 4, 5. However, whenever there is a chance that direction 6 is applied, 2 has to start first, then 3, then 4 and so on. If direction 7 is applied, 5 is the starting direction. Finally, if direction 2 is applied, then 6 goes first again. With this method used, the program ensures the right most pixels is reached when the pixel is going upward and the left most pixel is reached when the pixel is going downward.
However, there is a problem if an object has a very thin edge extending out. In this case, a loop connection would not be able to come back from the edge tip, and therefore, it would be unable to form a closed loop. After searching through the detected edges, the loop connecting method guarantees the outline of an object, which is marked for further processing (See Figure 18).
3.6 Filling Highlighted Edges

After an outline of the object has been produced, the next step is to fill the object. The approach is to fill an object used in this work is simple. For a given pixel, the algorithm checks pixels at its four directions, up, down, left and right, then cycles through each direction towards the boarder of the image. If an edge is encountered, the edge counter adds one to itself. After all four directions has been checked, and if the edge counter represents as a value of 4, which means the pixel is surround by edges, the pixel is then assigned a filled value, which in this case is 0 (see Figure 19), else the pixel receives a white 255 value.

Figure 18. Outline of a detected edge.
3.7 Extraction

After having the filled object (see Figure 20a) in place, extracting the original data from the filled area is very straightforward. This is done by checking the original image (see Figure 20b) and the filled image at the same time. Wherever a filled pixel is encountered in the filled image, the algorithm places the exact pixel value of the original image to the output. Otherwise it sets the pixel to the value of the white color (see Figure 20c).
The remainder of the image after extraction is shown below (see Figure 21). This remaining picture is later cycled through the process for detection of the next object available. The process goes on until there are no more images to extract.

![Figure 21. Remaining of the image after first extraction.](image)

Using the algorithm below, extraction image and remaining image can be derived.

```plaintext
if pixel in filled highlight (ex. Figure 20a) is black
   write pixel in original image (ex. Figure 20b) to extraction image (ex. Figure 20c)
else write pixel in original image (ex. Figure 20b) to remaining image (ex. Figure 21)
```
4. Evaluation and Results

4.1 Meeting the Objectives

This program meets all the objectives that were proposed.

- It accurately outlines the edges of an object within an image
- It is easy to use
- It successfully removes data outside of the object’s edges
- It allows the user to choose which object is to be saved

In a successful execution, the input file remains unchanged and the object image file saved would be one of the images that are inside of the surrounding highlighted edges where an object is detected.

4.2 Testing

A good number of tests were performed on the program. Some worked great, but some did not turn out so well. There are seven categories of testing that were done on the program. They are:

- Input file type
- Image size
- Variety of objects in an image
- Color depth
- Dark background and light background
- File opening and saving operation
- Images taken by digital camera
4.2.1 Input File Format

There were several image formats taken into the test.

- GIF
- JPG
- PNG
- BMP

The outcome was just as expected. This program can only take BMP, since other types of image have different header files, and some of them with data compression, which requires different reading methods and decompression. Therefore, other formats are not compatible with the program written.

4.2.2 Image Size

There are several standard resolutions of images tested here,

- 1024x768
- 800x600
- 680x480
- Everything else

The source code actually only went as far as 800x600 for the array, so 1024x768 is not supported. Surprisingly, 800x600 also did not work as well; the program returns memory errors when an image of 800x600 is taken in for reading the header file. Only image with
680x480 or images with smaller size can be processed. This problem might be caused by large size of array under go the edge detection algorithm, or the convolution process.

### 4.2.3 Variety of Objects in Image

Different layouts of the images were also used to test for program ability of recognizing various objects.

- Image with overlapping objects
- Image with half-surrounded object edges
- Image with object with different large color variance
- Image with zero edges

In the first case, with image with overlapping objects, it passed the test. However, it would only be considered to be one object, since both objects stand out from the background see Figures 22 below for graphical explanations.
In the second case there is a half-surrounded object. Since the program only considers an object to extract when it makes a complete loop, so if outline process can not
make a complete loop, it would not be taken in as an object. However, if it is an incomplete object with thick edges, the program would simply present the edges as the object.

In the case when there is a big color variance in an image, the program would simply consider it as a single object just like the first case where two objects overlap. And the result will turn out just like the previous test above on overlapping objects.

The last test of image object is the image without object. In this case, the program still goes through its routine, with a blank picture shown. However, if user decides to process the image, a blank bitmap would be sent in for the process.

### 4.2.4 Color Depth

Images with different color depth also have been tested. This test is to make sure the range of color that this program can accept.

- Image with 1 bit color depth
- Image with 2 bit color depth
- Image with 8 bit color depth
- Image with 24 bit color depth

Unfortunately, this program only decodes up to 8-bit color depth. Image with 24-bits requires different reading on the header file than those under 8-bit. However, it should not be a problem to implement just a separate header reading procedure is needed.
4.2.5 Dark Background and Light Background

This test was actually not much of a problem of concern before it was placed under the test. Even though Sobel Edge detection can detect image in dark background just fine, when it does the extraction from the original image, it leaves the area of extracted object white. Then when image process started again, it would think the white area under dark background is still an object, therefore, producing an output with blank data (see Figure 23). On the other hand, light back ground is ideal situation for the project, since Sobel does not detect an edge when there is not much of color variance.

Figure 23. Remain of the dark background image after extraction which could cause edge detection to repeat detecting its edges.
4.2.6 **File opening and Saving**

As for the file handling, since the program was later put together in Microsoft Visual C++, file duplication possibility, mistype of loading image files and disk space problem are all taken care of by the windows dialogs, and there is no problem at all.

4.2.7 **Images taken by digital camera**

After all the testing on the drawings that were purposely made for the test, it is time to get a real image directly from a digital camera, which is the popular form of image acquisition. This Image was taken from a Nikon Coopix 990, which is a 3.3 mega-pixel digital camera. The original image is a 24-bit bitmap (see **Figure 24**).

![Figure 24. Example of a picture taken from a digital camera.](image)

After acquiring the image, the image is then converted to grayscale for program process. The first process, Sobel edge detection, processed the image just fine. By
observing closely, there are little humps on the background wall (see Figure 25). This image is a perfect example of non-uniform background, which that might cause problem when selecting a pixel as an object.

![Figure 25. Sobel Image detection of non-uniform background](image)

After combining the two edge detection images we would get the following image. However, in this image, recognizing edge pixel is no longer anything other than white color value. Instead, the program had to be tweaked to a value that would take the humps in the background to a non-existence state. Meaning, everything white, and close to white, will be considered as “irrelevant background” (see Figure 26).
After knowing the edge pixel position of the object, the program can proceed on and outline the border of the object (see Figure 27). Afterwards, the program takes the image into the filling (see Figure 28) and extraction process.
Figure 27. Outline of the border edges.  Figure 28. Filling of the outline.

Taking a closer look at the extraction image (see Figure 29), the outline of edges is not exactly perfect. The Sobel edge detection actually produces edges that are a little off than the original image edge in some areas. These offset is causing the extraction to be as perfect as it would intended to be.
Another fact that is worth mention is the remaining of the image after extraction. It is unlikely that this image will be used again. The image contains a background that is not purely white. After the extraction, the object leaves a mark on the remaining image, which causes the Sobel to continue detect it as an object within the image (see Figure 30).
Figure 30. Remaining of the image after extraction.
5. Future Work

This program has provided a backbone to extract image objects, but it is far from perfect. It needs more input file format choices, better color depth handling, bigger image size handling and better optimization code on each algorithm applied, especially the area filling algorithm.

Besides making the program run faster and run without errors, user interaction can also be improved. For example, letting the user decide which object to extract first instead of rotating around among all the objects would be an improvement. Furthermore, extracting more than one object at the same time could be added.
6. Conclusion

The result of this project is a small, fast, and easy to use program that provides accurate processing for extracting an object among objects from image files. Not only does the picture show a better representation of the image, it also provides an accurate and faster way of object extraction. Furthermore, the program provides a friendly user interface, letting users to preview the images before and after the processing. Lastly, it provides the user with interaction to choose which object image to save, and later work on the specific object.
REFERENCES


