THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MECHANICAL AND NUCLEAR ENGINEERING

IMAGE PROCESSING FOR APPLE ORCHARDS AND CHRISTMAS TREE FARMS USING UNMANNED AIRCRAFT

NATHANIEL SNYDER
SPRING 2017

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree in mechanical engineering
with honors in mechanical engineering

Reviewed and approved by the following:

H.J Sommer III
Professor of Mechanical Engineering
Thesis Supervisor

Sean Brennan
Professor of Mechanical Engineering
Honors Adviser

Signatures are on file in the Schreyer Honors College

ABSTRACT

Unmanned Aerial Vehicles have become increasingly popular in many different sectors of industry. Specifically, as drone prices continue to decrease, agriculturalists are becoming more interested in capturing aerial data of their orchards and fields. Farmers who would like to automate tasks such as surveying rows of trees for blossom coverage or counting apples in an orchard are beginning to look towards drone technology as a possible solution. This research focuses on creating image processing algorithms for specific agricultural settings using a DJI Phantom III quadcopter. Three specific types of agricultural settings were the subject of the research; images of blossoming apple tree orchards, images of apple orchards in harvest season, and high altitude images of Christmas tree farms. The computer vision algorithms would read each set of images in, and batch process each for their attributes, respectively. Blossoming apple orchards proved the easiest for the automated object inspection (AOI). Characteristics such as number of rows, number of trees in a row, and apple tree blossom canopy coverage could be determined. Apple orchards images during harvest proved more difficult, as tree bunching created difficult shapes for the algorithms to identify individual trees. Properties such as number of rows and number of apples per row were determined. A graphical user interface (GUI) provided Christmas tree farmers a method to analyze tree density in a specific area of their field. The program was successful in counting all the trees within a specific region of field that the user would choose through their computer mouse. This paper explores the role of image processing in small to medium scale agriculture, and the algorithms that were created for the task of AOI. A discussion of further research regarding how AOI for agricultural images could be improved is provided as well.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thank you to those who have helped me develop my understanding of image processing.

Chapter 1: An Introduction to Image Processing within the Agricultural Sector

Beginning in the 1980's, drone technology was first introduced to large scale farming operations as a way to quickly and efficiently gauge plant health over large distances. Drones made scanning fields for unhealthy crops a quick endeavor, and sophisticated image stitching software was developed alongside to help process images and create a map of plant health for the area the drone flew. Employing drones for this task has now become a fairly common practice within large scale agriculture. Drone technology has only begun to be explored by small and medium scale farmers to determine the health and characteristics of their crops, however. While many plant-health indexing methods exist, a 'Normalized Difference Vegetation Index' (NDVI) image is used commonly as a means of gauging plant health with UAVs [1].

Though the topic of drones produces mild contention among some social circles, few can deny the presence of the drone market. By 2018, approximately 600,000 commercial drones are estimated to be airborne and flying missions [2]. Drone technology is now being explored not just for NDVI image analysis on fields, but also for other agricultural operations as well. Through integrating drones with agriculture, researchers and farmers hope a solution can be created that cuts farming costs associated with land and crop maintenance. Due to the continual improvement of GPS accuracy within newer drone units, flying autonomous missions are on track to become increasingly easier to implement and execute. Though NDVI images can be very helpful to gage plant health, new agricultural methods using drone technology are being developed with techniques other than NDVI image analysis to find crop attributes. This could be

especially significant for the small to medium scale farmer looking for cost and time saving solutions to menial time-consuming counting tasks. Image based drone technology can play a particularly useful role for identifying what proportion of a tree is covered in blossoms, or to count fruit on the tree. Autonomous drone technology has the promise to start performing these time-consuming tasks farmers would once have to complete on their own, or hire a worker to accomplish. Due to the competitive pricing of new drone units, agricultural drone units and software could form a unique method for smaller scale farmers to automate lengthy and tedious tasks.

With current drone quadcopter models running less than $1000 [3], the cost of a drone based agricultural systems is financially feasible for smaller scale farming operations to employ. Flying missions over smaller sized fields also proves advantageous for the farmer, as it can be done on as little as a single battery charge. Though NDVI image operations are relatively well explored, other machine vision algorithms have not been standardized to such an extent. For drone-based agriculture to effectively expand to the small to medium scale farmer, robust algorithms and software that can process aerial agricultural image content must be developed.

For an agricultural drone to be useful for a small scale farmer, it must be able to accomplish a task for a farmer with equal or less financial and physical effort. That is, the opportunity cost of using the drone must be less than the opportunity cost of manually accomplishing the task. These tasks could include identifying geographic locations of orchard rows, counting fruit, counting crops, finding unhealthy crops, identifying blossom canopy coverage, etc. If a UAV is able to gather the photos either autonomously or driven by the farmer’s guidance, then computer vision algorithms could be constructed to process the field

data. If the algorithms produced closely matching or better results when compared to the human eye, then the computer vision algorithm could be said to have equivalent efficacy.

To test how well an automated set of programs could accomplish the task of object identification, C++ classes of agricultural image processing algorithms were created. UAV images for apple and Christmas tree farms were then fed to the computer vision (CV) algorithms for analysis. Figure 1 below shows a typical test image the CV algorithms were given. The apple orchard images (both blossom and at harvest) were taken at a close proximity to the canopy whereas the Christmas tree images were taken at a higher level above ground. For the scope of this research, five main tasks were tested using CV algorithms. The algorithms would be tested on how well they were able to identify individual rows, identify individual trees, identify blossom canopy coverage, identify the number of apples per tree, and lastly count and identify the number of Christmas trees within a region of a high-altitude image.

Apple Orchard Blossoms	Apple Trees at Harvest	Christmas Trees
		

Figure 1: A Preview of Apple Orchard, Apple Orchard at Harvest, and Christmas Tree Photos Processed

Three separate C++ classes were created to meet the challenges posed by the three different types of images. The ‘BlossomOrchard’ class was used for image processing when the apple trees were still in blossom. The class object would read in an image, find each row within an image, count the number of trees within the first row of the image, identify all blossoms on

each tree within the first row, and determined its blossom canopy coverage. The ‘AppleOrchard’ class dealt with the apple trees at harvest. The AppleOrchard class would read an image, identify each row of an image and count the number of apples within the first row. Statistics such as mean apple size, standard deviation and variance were collected per image to estimate the mean apple size. The ‘ChristmasTree’ class was used for all Christmas tree counting operations. This class had the responsibility of allowing a user to select a specific region of their image, and then have all of the Christmas trees within that area counted. Each class, their tasks, and overall outcomes will be discussed in Chapters 4 to 7.

A DJI Phantom III drone was used for image capture. The Phantom III is a standard quadcopter drone equipped with a high resolution HD camera capable of taking high quality 12 megapixel photos [2]. The Phantom III provides advanced stabilization control during flight patterns and can fly smoothly even with powerful winds. At $700, it is affordable for purchase for many small to medium scale farmers. It is GPS driven and can autonomously fly back to the point of origin with a single button command. For orchard image capture, the drone was manually flown up and down orchard rows at approximately 30 feet altitude with the camera facing towards the orchard trees at a downwards 45-degree angle. For Christmas Trees, the drone was flown at 200 feet above ground level (AGL) with the camera pointing directly downwards. Once taken, the images were stored and batch processed to test whether the CV algorithms were able to identify the image’s physical traits.

The C/C++ language was chosen due to its strength within computer vision applications. The C++ language can be compiled into exceptionally fast assembly language, with very little overhead. In most cases, a well programmed C++ script can achieve approximately 90% of the run time the same code programmed directly into assembly language is able to achieve. The

advantage of C++ to higher level languages like Python or Java is efficiency. Most computer vision libraries are written innately in C++ and then ‘wrapped’ into other languages. For example, the MATLAB programming language relies on C based code to carry out all instructions typed into its editor or command window. Similarly, most computer vision libraries written in Python and Java (such as the widely used OpenCV) depend on underlying C code wrapped in high level Python or Java function handles, respectively. Using C++ provides fast execution, as well as access to professional level open source CV libraries.

C++ was used in conjunction with the OpenCV library and Visual Studio 2013 to develop the algorithms and classes. OpenCV proved to be an extremely valuable tool for all image processing operations, as well as a powerful pairing to C++’s useful standard template library (STL). The next section introduces basic image processing theory, color spaces, C++ STL and OpenCV libraries necessary to understand the theory, methodology, and findings of the research. Once some basic image processing and methodology is introduced, a discussion of the classes and image analysis environment used to process the apple orchard and Christmas tree images will be had.

Chapter 2: An Introduction to Image Processing Theory, STL and OpenCV 3

An Introduction to Image Processing Theory:

To better understand computer vision, it is helpful to first understand basic theory of how humans perceive images. Unlike a computer, the human eye effortlessly takes in lighting, brightness, shapes and color information from the surroundings. This information is sorted immediately into different cognitive containers and checked for significance. There are four noticeable characteristics of our visual system [4]. Our visual system is first sensitive to low frequency content. Low frequency content correlates to visual content that does not change pixel value rapidly. High frequency content would then refer to regions with corners and edges which fluctuate in intensity more severely. Second, our visual system is more sensitive to changes in brightness than to changes in color, as well as changes in motion. Third, human visual systems are exceptional at deciphering if an object is moving even if our focus is not directly on the object. Lastly, the visual system is very good at finding any noticeably distinguishable features, such as a brushstroke of yellow on a dark black background.

While the human visual system is good at locating contours and objects, it is exceptional at quickly distinguishing objects; a task that a computer finds very hard to accomplish. For example, when a person enters a room, he or she may see other people, furniture, bright lighting, and all sorts of devices. While sorting out the visuals within the room would not prove challenging to the person, this would be a very tough task for a computer. Much research has

been devoted to further understanding what makes humans so good at recognizing objects, and machines quite poor at the task [4].

Visual data processing and object recognition for humans occurs in the ventral visual stream. This part of the brain takes visual data and instantly sorts it for any particularly important features [4]. Researchers have found the human brain does not use factors such as size of an object, illumination or orientation to normally decipher objects; even if a person is swinging upside down, a table is still a table. Humans recognize objects using relative geometries within the object in part with distinguishing its other notable characteristics. In fact, to process objects with greater complexity, complex cells within the visual cortex are trained to process the visual data. As people go about their day to day lives, they continually train their object recognition system, and do it so well we do not even have to think about performing such operations.

Machines unfortunately do not have all the advantages of the human brain. Many mysteries still remain regarding how data is stored upon entering the human visual system, making it difficult to effectively mimic such storage techniques with a computer. Computers store an image's content as a discrete matrix of data points. This makes object recognition for a computer difficult, since an image of a chair from the front may look nothing like the data pattern of the chair from the side. Image size, orientation and brightness all make it difficult for a computer to identify whether an object is related to another. Since a holistic picture for how our brain deals with object has not yet been discovered, programming a computer to identify objects as the human brain does becomes problematic.

One method would be to store all perspectives, angles, brightness of the object, but such a method would not be practical if many objects needed to be considered (not to mention computationally expensive). Machine learning algorithms could prove beneficial to recognize

various object shapes, but due to the limited time and resource, such a technique was postponed for further research. To identify objects from one another, specific object properties such as color, size, shape and brightness must be manipulated in order to find specific objects within a given agricultural image. Robust algorithms to store large amounts of image data were developed using the open source C++ computer vision library 'OpenCV' in conjunction with the Standard Template Library for storage and container efficiency.

Open Source Computer Vision (OpenCV) is a library of templated computer algorithms aimed at solving real time computer vision tasks. Originally written for C++, OpenCV uses the C language's speed and effectiveness to build fast code. While other languages support OpenCV such as Python and Java, the library is written natively in C++ and follows the C++ Standard Template Library (STL), allowing for seamless implementation with the STL. OpenCV has over 2500 optimized algorithms, which rely on various classes and functions the CV library has defined. A basic overview of the commonly used OpenCV functions, classes, and image processing theory will briefly be touched upon and discussed. See Appendix A for the select list of library functions used, respectively.

As briefly mentioned, all images are stored in computer vision programs as matrices, with each pixel encapsulating a unique attribute of physical space. Numbers within the matrix stand for a measured wavelength intensity for a discrete patch of the surroundings, as seen in a typical RGB image. The numbers can also represent a wavelength range as seen with panchromatic devices [4]. Each number value is considered to be a small part of the image and what is known as a pixel. A pixel can store either a single value as seen with a grayscale image, or can be composed of different channels. For example, Figure 2 below shows number values corresponding to a grayscale image and a RGB image. The gray scale image to the left can be

represented by numbers usually within the 0 to 255 range and consists of a single channel. RGB images however have three times the information density due to the red, blue, and green channels that make up the image, with data values similarly ranging from 0 to 255.

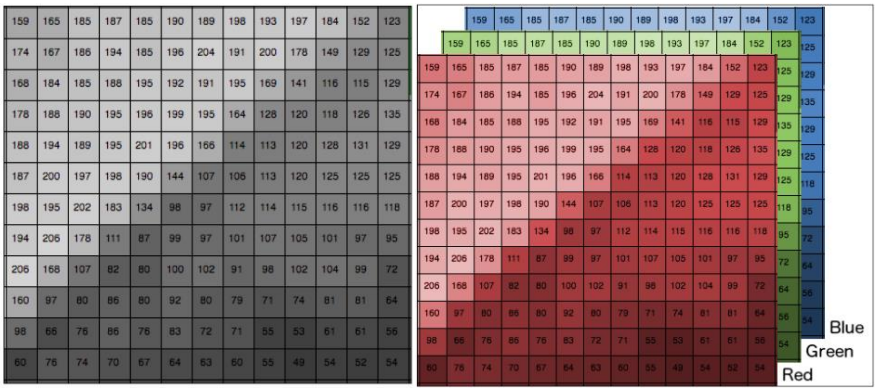


Figure 2: Example of Gray and RGB Image Storage

Depending on what type of image a user wishes to store, the data contents will be stored in computer memory slightly differently. Computer memory can be conceptualized as a long straight line of data points. Any three channel array will have the same organizational pattern as the RGB array. Since a grayscale image is a single channel array, it will not have a RGB channel, but instead a single value for each pixel between 0-255. Each pixel can be accessed in memory through using the value equation seen in Figure 3 below.

Row 0									**Row 1**									**Row 2**								
Col 0			Col 1			Col 2			Col 0			Col 1			Col 2			Col 0			Col 1			Col 2		
Pixel 1			Pixel 2			Pixel 3			Pixel 4			Pixel 5			Pixel 6			Pixel 7			Pixel 8			Pixel 9		
B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	B	G	R

```
Value= Row_i*num_cols*num_channels + Col_i + channel_i
```

Figure 3: Illustration of Image Memory Storage Pattern and Equation

Introduction to OpenCV 3:

Before results can be discussed, an introduction to the theory that make image processing possible using OpenCV and C++ should first be mentioned. Computer vision requires a strong knowledge of dynamic programming as well as image processing theory. Without such, new image processors may find it difficult to create useful algorithms. The next two sections list important functions, classes and algorithms for image processing agricultural data.

For beginners, the most widely used tool in image processing is the matrix. The OpenCV *Mat* class stores data just like a matrix does. It is the primary storage tool for reading an image into an image processing environment. The Mat class can be thought of as an n-dimensional single or multi-channel array [4]. Like any matrix, the Mat class has the ability to undergo matrix multiplication, addition, subtraction, inversion and transposition. A Mat object can be assigned an image matrix with the OpenCV function 'cv::imread()', or it can be assigned to any set of values the user wishes through its class constructor. The Mat class mimics the template pattern of STL containers such as 'vector< >' and can store most image types. If an image needs to be processed, the Mat class should be used.

The Mat class has a few member functions that are widely used with OpenCV matrix operations and support random access to the Mat elements. Table 1 below defines the most important four member functions associated with the Mat class.

Table 1: Common OpenCV 'cv::Mat' Class Member Functions

begin()	Function returns an iterator pointing to the first data value in the array.
end()	Function returns an iterator pointing to one past the last data value in the array. The contents of end() is undefined and should not be accessed.
copyTo(Mat& m)	Function performs a deep copy on the current matrix contents. Stores the copy within the parameter matrix "m".
at<>()	Template function returns a reference to the specified array element. This function is used for safe random access of Mat object pixels.

The *Moments* class is widely used for finding object locations and object areas. The most common method for using the moments class is to convert an object's contours into a moment object. Using the moment class is challenging for inexperienced C++ programmers since it relies on knowledge of multidimensional dynamic memory containers, STL iterators, and OpenCV functions such as findContours() and moments(). Table 2 below gives a list of steps to instantiate the moments class, along with a source code example in Appendix A.

Consider the task of finding the location of an apple within an image using the OpenCV moment method for object recognition. It is assumed that a vector with the apple's contours have already been determined. To extract the apple's coordinates, a pointer to a vector of contour points should be created. If there are multiple contours within the multidimensional contours vector, a looping mechanism should be implemented to extract all of the various contours from the vector, and the iterator should be incremented with each loop. A moment object must be declared using the notation from step 2 using a matrix cast within the parameter for the moments function. Once casted, the function cv::moments() will convert a casted Mat to a moment object. The moment class is significant since it has important member properties m00, m01, and m10,

which are used extensively to find an object's area and center. Table 2 below depicts how to use the moment class to find area and center of an object.

Table 2: List of Operations for Converting Contours

1. Instantiate iterator pointing to contours	vector<vector<Point> >:: iterator it = contours.begin()
2. Declare Moment Object:	Moments M = moments((Mat)*it)
3. Find object area	double area = M.m00
4. Store object center in a point object (x,y)	Point center =Point(M.m10/M.m00,M.m01/m00)

The *Rect* template class stores rectangular coordinates through use of its constructor and member properties. The Rect class is widely used as a method to crop one image into an area of interest for the user or program. Image coordinates are saved into a rectangle object, and used as a parameter in the Mat class constructor. Figure 4 below introduces how an image matrix can be cropped into a smaller image matrix using the ROI technique.

```
// Crop Region of Interest
Mat ROI = userXmasTreeImg(Rect(Point(leftPoint, topPoint), Point(rightPoint, bottomPoint)));
```

Figure 4: Using the cv::Rect Class to Crop Image

The *Point* class is a simple yet useful template class for holding x and y image points. The class has two properties 'x' and 'y' that are used for storing x and y image values, as seen in the above ROI example.

The *Scalar* class is a simple template class which can hold up to a maximum of 4 values. The Scalar class is generally used along with Hue Saturation Value (HSV) and RGB morphological operations to conduct image binarization. Scalars are used as a means of

transporting the upper and lower cutoff values that allows an image to be binarized in OpenCV. Image binarization is an extremely important concept that will be discussed in Chapter 3.

Color space conversion and image filtering plays a pivotal role in AOI algorithms. One of the most common techniques for conducting color-based object detection in OpenCV is through the function cv::cvtColor(). The function allows a user to switch an images color space from one type to another. Color based object detection using the Hue Saturation Value color space was an important method for locating objects for this research project. Figure 5 below illustrates the 3D representation of a HSV color-wheel. The HSV color space is a cylindrical coordinate representation of points in an RGB color model [5]. The HSV color space is preferable to using image based analysis within the RGB color space since it aligns more closely with how people experience color. In an HSV color space, the 'Hue' of the color refers to the pure color of the object. For example, even if an object has a light red color, its hue value will correspond to a perfect red nevertheless.

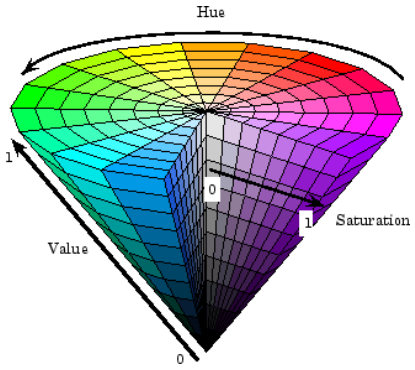


Figure 5: Visualization of the HSV Color Space

Hue is described on its color wheel by a rotation to the number that represents the position of the color [6]. In OpenCV, the values for Hue are from 0-180. Saturation represents a radial move outwards on the color wheel. Colors that have full saturation are located further radially outwards than colors with less saturation [6]. In OpenCV, saturation values range from 0

(white) to 255 (fully saturated), respectively. Value described how light the color is, and refers to a vertical translation on the HSV color wheel. Values range from 0 (black) to 255 (white) in OpenCV. Figure 6 below illustrates how to navigate the HSV color space.

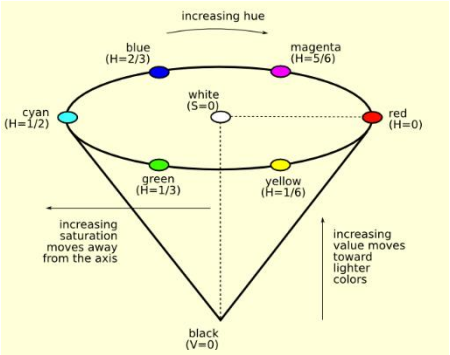


Figure 6: Navigating the HSV Color Space

Since HSV is simply a transformation of the RGB color space, the HSV values found after conversion will always depend upon the initial red, green and blue values of the image [7]. While using images converted into the HSV color space is useful for computer vision tasks, great care must be taken to properly filter out substantial noise from the image. OpenCV morphological functions such as thresholding, dilation, blurring, and erosion are employed to smooth data. The end goal of using the HSV color space is to take a read in RGB image and transform it into a binarized thresholded image with only the object of interest remaining within the image. This process is covered in great detail within the next chapter and used extensively within each object recognition algorithm.

Introduction to the C++ Standard Template Library (STL):

Lastly, a brief overview of the STL will be mentioned as it is used in many computer vision source code examples. OpenCV follows the C++ standard to create a seamless integration for all CV functions and STL library containers. This is exceptionally important for image processing within stochastic environments like apple orchards. Within each image, it can never be precisely predicted exactly how many apples, blossoms, trees, or rows will be found. Therefore, it is impossible to pre-allocate how large an array should be, or how many objects will be found within a given image. Furthermore, if objects are accidentally filtered out, predicting quantities of objects is further muddied. Luckily, due to STL's dynamic memory containers, this does not pose any problem, as containers can grow and shrink as more or less elements need to be added and removed.

A formal definition of *template* may be needed at this time. A template can be thought of like a placeholder. Similar to how a school lunch box can contain any type of food, a template class can contain any type of parameter it is initialized with. Instead of creating one type of class that uses an integer, and another similar class that uses a double, a template class can be created which can be instantiated with any type of parameter (integer, double, or even a string). This type parameterization allows for great code re-usability and faster code creation. Templates can be used to define both classes and functions, making it possible to enact powerful sorting and numerical operations on a data container [5].

The vector class meets the need of a dynamic memory container for computer vision tasks. Instead of declaring an array with an initial quantity of elements, a template container known as a vector can be initialized. The vector class is a sequence container capable of random

memory access with dynamic memory allocation. Table 3 below illustrates commonly used member functions of the vector class.

Table 3: Member Functions Commonly Used in the std::Vector Class

begin()	Function returns an iterator pointing to the first element of the container
end()	Function returns an iterator pointing to one past the last data value in the array. The contents of end() is undefined and should not be accessed.
size()	Function returns the current size of the container
erase()	Erases elements within the container. Commonly used as part of the remove-erase idiom as discussed below
push_back()	Places an item into the vector container
pop_back()	Deletes the last element.

Not only does the vector class allow for template use, but also has a special pointer that can access its data members. An iterator is a special data access tool inherent to each of the main sequence container classes. The iterator serves as a pointer to a specific data member within the container, but can be incremented or decremented to loop through the whole data set. The vector functions begin() and end() are used to return the start and end memory locations of the vector, and used extensively for computer vision data storage.

Sorting data is another crucial task in computer vision. The STL library uses a template sorting function sort(), which sorts data within an STL container such as vector, according to a user-provided sorting predicate. STL sorting predicates may be fairly esoteric to the inexperienced C++ programmer, but serve as powerful tools for writing single line code. Sorting predicates even work with large multidimensional arrays of data in a single line call to sort.

Figure 7 below depicts a *functor*, which can be used as a Boolean sorting predicate by overloading the operator(). The functor is then used as an instruction set for sorting the content

of a vector container. Functors serves as quick methods to instruct the sort function how to arrange the data members.

```
// Sorting Predicate: Sorts a vector container of pair<> in ascending order
struct sort_pred_double_ascending {
    bool operator()(const pair<double, Point2i> &left, const pair<double, Point2i> &right)
    {
        return left.first < right.first; // first value is the smallest
    }
};
```

Figure 7: Example of a Functor Used with Standard Template Library Sorting

Figure 8 below depicts how this functor was implemented as a means of sorting a vector of data. The sort function is given the range of data value memory addresses that should be sorted, along with specific sorting instructions provided by the functor.

```
// Sort Results by Area - Ascending
std::sort(objectContainer.begin(), objectContainer.end(), sort_pred_double_ascending());
```

Figure 8: Sorting an Object Container Using a Sorting Functor

Many times in CV applications, data points need to be added and removed from a vector continuously. For example, if a program required all repeat values within a vector to be removed, one technique could be to loop through the array and delete repeated data points through random iterator access. However, this may result in undefined program behavior. Instead, safe STL functions can be used to safely remove any items from a vector. Figure 9 below shows how elements were safely found, stored, and then removed from the desired vector through using the ‘erase remove‘ idiom, which assures safe element removal from an STL container.

```
// 12) Use erase remove idiom to find and remove all of the objects within the container that are undesirable
for (vector <pair < pair< double, Point2i >, vector < Point> > >::iterator itF = eraseObjects.begin(); itF != eraseObjects.end(); ++itF)
{
    treeAreasAndCoordinates.erase(remove(treeAreasAndCoordinates.begin(), treeAreasAndCoordinates.end(), *itF), treeAreasAndCoordinates.end());

}
```

Figure 9: Erase-Remove Idiom Used for Safe Data Removal from Container

With a basic understanding of the main components that make C++ computer vision possible, a discussion of how the UAV's images were analyzed is provided in the next chapter. Visual Studios and OpenCV 3 were used to create an image analysis environment (IAE) where unique properties of each orchard and field image could be discovered and analyzed. The environment employed a graphical user interface (GUI) for user interaction, tracker bars and sliders for testing morphological operations, as well as mouse and keyboard events to run binarization and AOI algorithms on the image sets. The image analysis environment (IAE) was crucial to discovering specific image characteristics for AOI. In addition to a discussion IAE creation, important OpenCV morphological operations, filtering algorithms, binarization and thresholding techniques will also be discussed. See Appendix A or https://github.com/natsnyder1/AppleOrchard for the full IAE source code.

Chapter 3: The Image Analysis Environment

All source code and examples were created through using Visual Studio 2013 and OpenCV 3. Visual Studio provides an ideal environment to run, debug, and edit source code for machine vision projects. OpenCV 3 fits very well into the Visual Studio framework after the correct libraries are linked to the project. Visual Studio 2013 and OpenCV 3 are a recommended pairing for beginner CV applications.

A basic image analysis environment can be composed of three parts. Code is first called to read a selected image into memory and set image variables. Next, functions that set up a graphical user interface (GUI) are called to allow interaction with the captured image. This is particularly useful, since a user can see how a certain morphological operation affects their image, and simply slide a tracker bar back to undo the transform. Lastly, mouse and keyboard events are used to call and execute automated object inspection (AOI) algorithms on the image. Mouse and keyboard events are very useful for deciding which operations to run on an image at any single instant. OpenCV provides the keyboard function “cv::waitKey()” which returns an ASCII integer corresponding to the user’s pressed computer key . The function will return -1 if no keyboard key is pressed. Users should implement keyboard, mouse, and user interface events to make their IAE more robust.

Crafting computer vision code is tedious however. The ‘Image Analysis Environment’ is the basic code skeleton beginning image processors would form to start analyzing their images. The main purpose of creating the IAE is to build interactive image software to test morphological operations for each of the blossom, apple, and Christmas tree image AOI. By creating a well-structured, reusable GUI equipped with functionality to adjust image values and

toggle image operations on and off, finding important image parameters becomes drastically less time consuming.

OpenCV erosion, dilation, blur, color space conversion, binarization and thresholding functions proved significant for accurate AOI. Each of the apple, blossom, and Christmas tree images were subjected to such morphological transformations to convert their color space, filter, and binarize. Each concept is explained through the image analysis example provided in Figure 10.

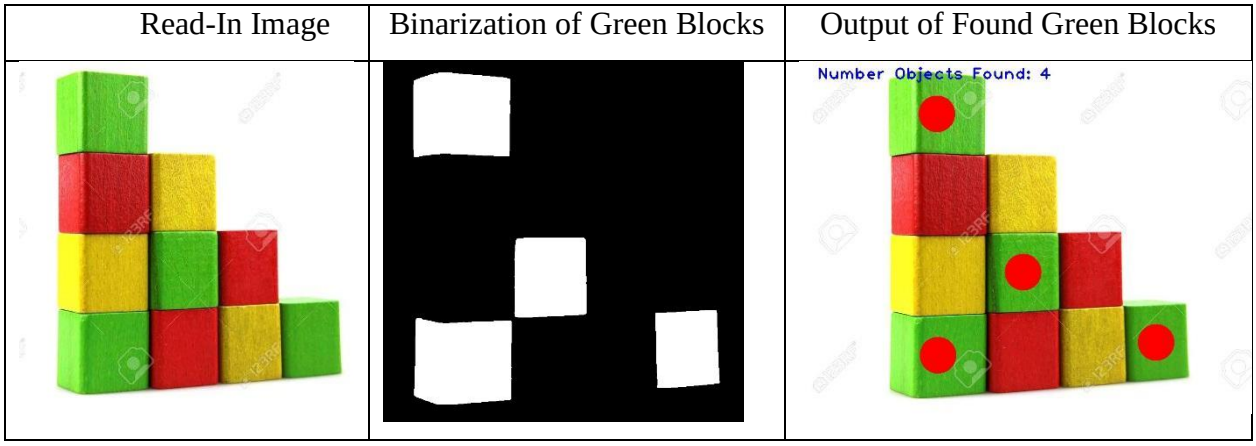


Figure 10: Example of Automated Object Inspection Process

Suppose a user wishes to develop a script to automatically count how many green objects are in the 'Read-in' Image above. To accomplish the task, he or she would need to develop a program closely resembling one a typical image analysis environment would employ. The block example serves as a simple introduction to understand the functions and theory that make computer vision for agricultural applications possible, and will be used to introduce these concepts.

For CV operations involving object recognition, an RGB to HSV color space transform is typically first performed. As previously mentioned, the HSV color space is particularly useful for image analysis and object recognition. Once an image is converted into the HSV color space, an

upper and lower range of hue, saturation, and value points should be determined to filter out colors and objects not of interest. All pixel values that fall outside of the range will be filtered out, leaving a binarized image remaining as seen in the middle section of Table 5. Any pixel value that falls within the specified range is given a value of 1, whereas any pixel that falls outside of the range is given a value of 0. Therefore, if a HSV range of values that solely correspond to green are found, what remains is a binarized image with only the green objects of interest in white. By converting the image to the HSV color space, a specified range of HSV values can be found to threshold the image into a foreground and background.

However, not all thresholding works exactly as planned. The right picture in Figure 11 depicts a common issue when binarizing an image using HSV color ranges. If two objects are close to one another, and both have the same color, the binarized image may appear as though the two objects are joined. In Figure 11, the upper right corner of the middle block and lower left corner of the right-most block are connected by a thin line. This is problematic if the objective of a program is to count objects or determine coordinate locations. While the human eye can clearly tell the two objects are not joined together, a computer does not possess the same cognitive reasoning skills.

The answer to this dilemma lies within the OpenCV dilate, erode, and blur functions. Dilatation is used to increase the size of a binarized object. Applying a dilation function will in essence wrap a thin layer of pixels around the contour of an object. The erode function on the other hand is used in essence to peel a layer off of an object. If the object is small enough, the erode function will destroy the object. Erosion is a powerful filtering tool in image processing, as unsmoothed data can lead to undesirable results and inaccurate findings. Using a

simple erosion function with a large kernel could solve the problem for the red blocks binarization image in Figure 11.

Blurring an image is another common pre-processing function used for filtering noisy image data. Blurring smooths the high frequency noise, and is commonly used for pre-processing an image before finding contours of an object so noisy pixel data will not affect contour locations. While blurring is a very useful filtering tool, erosion is typically the agent used to separate overlapping objects as seen in Figure 11. A more mathematically dense description of each function can be found on the OpenCV filtering reference page. Once an image's color space is converted, binarized, and filtered, objects remaining in the foreground can be tested for certain criteria and classified.

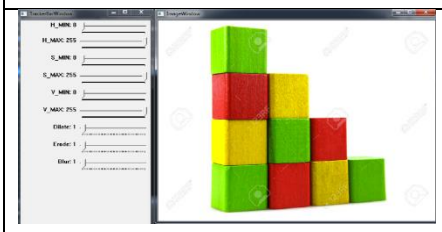
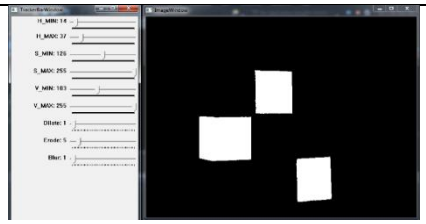
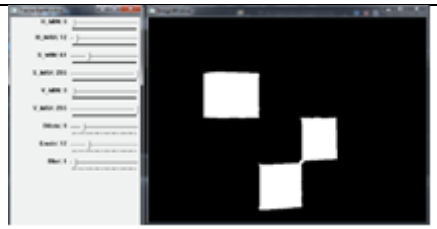
RGB Image Without HSV Conversion	Yellow Block Binarization	Red Block Binarization
		

Figure 11: Binarization of the Yellow and Red Blocks

Tracker bars and GUI events can be useful for determining the HSV range of values for filtering out background objects. To initialize a tracker bar object in OpenCV, a call-back function must be provided. The call-back function is executed each time the position of a tracker bar is changed. Figure 11 above shows how changing the range of HSV max and min values leads to different colored blocks in the foreground of the binarized image. By using both the tracker bars to manipulate the upper and lower HSV values as well as the function cv::inRange() to binarize the image pixels using those values, the user can visually see which objects from the

RGB image appear in the foreground of the threshold image. Since the binarized image contains the pixel locations of each foreground object, the objects can be analyzed to determine whether or not it classifies as an object of interest. In our case, the binarized objects have all the properties of the colored blocks, and would be considered an identified object.

Thresholding an image into a foreground and background makes identifying objects and their classifying their properties possible when using a color-based approach for image processing [8]. Once an object's threshold image is found, its area, center point, and shape can be determined. Checking object properties after binarizing an image is very important. With any AOI algorithm, some noisy image data will inevitably slip through. Having a set of conditions each foreground object must meet will help filter out skewed data, or at least alert the program that an object does not meet the proper specifications. Figure 12 below illustrates how several objects within a UAV image of an apple orchard should be excluded.

Figure 12: Unintended Binarization of Cinder Blocks and Posts in Blossom Foreground

If the objective is to count individual apple blossoms and determine the average blossom size, the inclusion of objects such as the cinder blocks, post, or white pipe would drastically skew the results. Since some objects, like the cinder blocks and piping, appear to have a similar color to the blossoms, they will appear in the foreground of the binarized image. It then becomes the programmer's job to use various conditions in order to filter out these objects from the results.

Although a finalized AOI script will not contain slider bars for users to manually adjust HSV max-min values, the image analysis environment is critical in helping find what image attributes are significant for a given task. By setting up the IAE with mouse events, UI features, and other morphological operations, various CV functions can first be tested before being implemented. In the blocks example above, the same set of image operations were applied to colored blocks for object recognition; color space conversion, morphological operation, filtering, binarization, and finally output. This technique was used repeatedly for each set of UAV agricultural images. Now that a basic introduction to computer vision theory and programming is covered, the next three chapters will discuss the results of the UAV AOI algorithms. Apple orchard images are discussed first, followed by a discussion of Christmas tree counting. The full source code to set up an IAE for Visual Studios 2013 with OpenCV 3 can be found on https://github.com/natsnyder1/AppleOrchard**.**

Chapter 4: Image Processing Blossoming Apple Trees

Consider Figure 13 below. This is a typical UAV image taken from the DJI Phantom III quadcopter from approximately 30 feet altitude with a camera pitch of 45 degrees down. To collect data for image processing, the drone was manually flown down the center of each orchard row. Images were taken every six feet by the camera attached to the underside of the drone.

Figure 13: Illustration of Blossoms Taken From UAV

Computer vision AOI algorithms were created to process various characteristics of the apple orchard trees. ‘Blossom canopy coverage’ is a commonly used term by orchard farmers to describe how dense a tree is covered with blossoms. Depending on the quantity of blossoms an apple tree has, its crop yield will vary. If a tree is seen to have a high blossom canopy coverage, a farmer would then expect the tree to produce a larger quantity of slightly smaller apples. If a tree has a low blossom canopy coverage, a farmer would then expect the tree to produce a smaller quantity of slightly larger apples. Depending on whether the farmer wishes to harvest the apples for juicing or the fresh fruit market, a certain type of apple is preferable. Apples with smaller sizes tend to be used for juicing, whereas larger, more aesthetically pleasing apples are brought to market. Managing blossom coverage becomes important then for determining whether

a tree requires blossom thinning. Because apple trees can grow up to 30 feet tall, seeing a full view of the tree canopy is sometimes not possible from the ground.

A UAV was used to test whether aerial AOI with OpenCV could identify the same traits and objects a farmer would during an orchard inspection. The UAV was manually flown between each row to capture low altitude images as seen in Figure 13. Computer vision scripts would then extract the physical characteristics of the image and analyze the orchard data. Realistically, in order for a farmer to implement this strategy over a physical inspection themselves, the UAV must perform the flight mission autonomously. Autonomous orchard missions requires GPS slightly more precise than what is currently available today. GPS drift exacerbates UA flight accuracy due to day-to-day longitude and latitude position changes. Maneuvering through orchard rows at low altitude requires exact flight planning. Further sensors and external control units would be required to assure precise flight of the drone, which was outside the scope of the research.

For blossom image processing, several goals must be accomplished for apple orchard AOI. Each row within the image first must be identified. Since only the trees within the first row of each image are of importance, the first row should be copied into a new image matrix. Next, each tree is identified and their coordinates are stored, respectively. Once tree locations are found, blossom binarization and object filtering is required to remove any objects that are not blossoms. The blossom canopy coverage of the trees can then be found, along with statistics of the blossom canopy coverage for individual trees. Each step will be covered in detail below.

The C++ class ‘BlossomOrchard’ was used for all blossom image processing. Each image was read into a CV script to initialize a BlossomOrchard object. The object would then subsequently call a set of functions on its image to determine characteristics such as number of

rows, number of trees, and blossom statistics per image. Finding the first row of orchard trees was the first major task for the computer vision script. The task was accomplished through using the classes member function ‘findRowLocations’. Figure 14 below illustrates what a binarized image of the rows looked like.

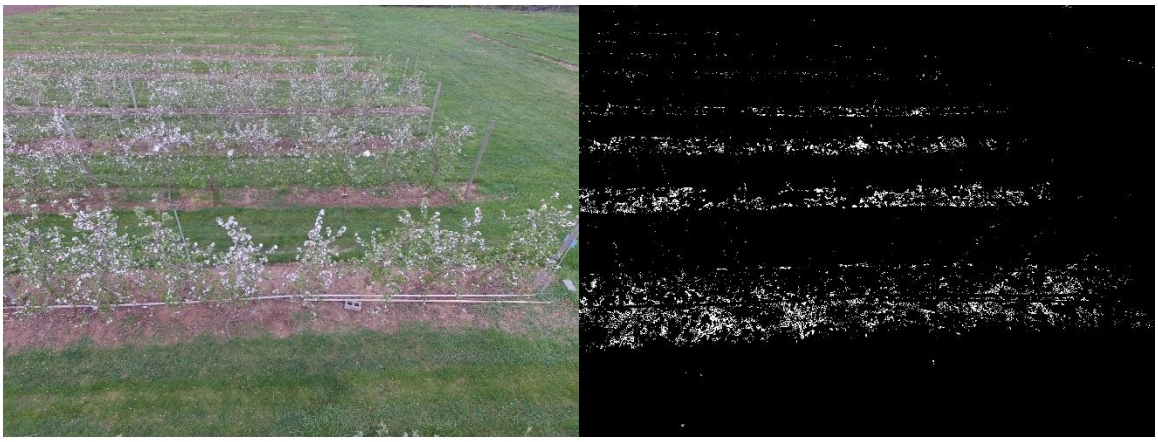


Figure 14: Binarization of Blossom Orchard UAV Image

As discussed in Chapter 3, the RGB image taken from the UAV was converted into the HSV color space. Since identifying rows is the first main objective, the image must be searched for characteristics corresponding to the row locations. Large patches of dirt between grass rows are characteristic of ground. Therefore, the image can be searched for HSV values corresponding to brown. A pre-determined HSV range of values was selected using the image analysis environment for all row binarization for blossom images. The filtered and binarized foreground on the right of Figure 14 resembles areas of the image that correspond to HSV values within the specified range. Unlike the block example in chapter three, the binarized image appears to have thousands of small objects instead of a couple well defined ones. This presents a challenge for AOI, as the aggregated grouping and spacing of each object is what is important, not individual pixels.

Figure 15 below depicts two strategies that can be employed to tackle this problem. One solution is to apply a heavy dilation to the binarized image, if a high enough kernel value is used, then the majority of pixels will merge together as a single object. Sorting out which objects are rows, and which smaller pixels groups are noise becomes relatively easy if the coordinate locations and areas of the objects are tested. The second method is to use a horizontal binning strategy to compare areas with high and low foreground density. Since the foreground pixels have large spacing between the rows, finding where one row starts and ends becomes a matter of testing how dense a row of image pixels is when compared to the average.

The first strategy is that shown in the left path of Figure 15. While a machine learning approach could be taken, the strategy used here tested how well the images content matched a predefined range of HSV values. After the RGB image was read into program memory, it was converted into the HSV color space. Once converted, the pixels were tested against a range of HSV values resembling light to dark brown.

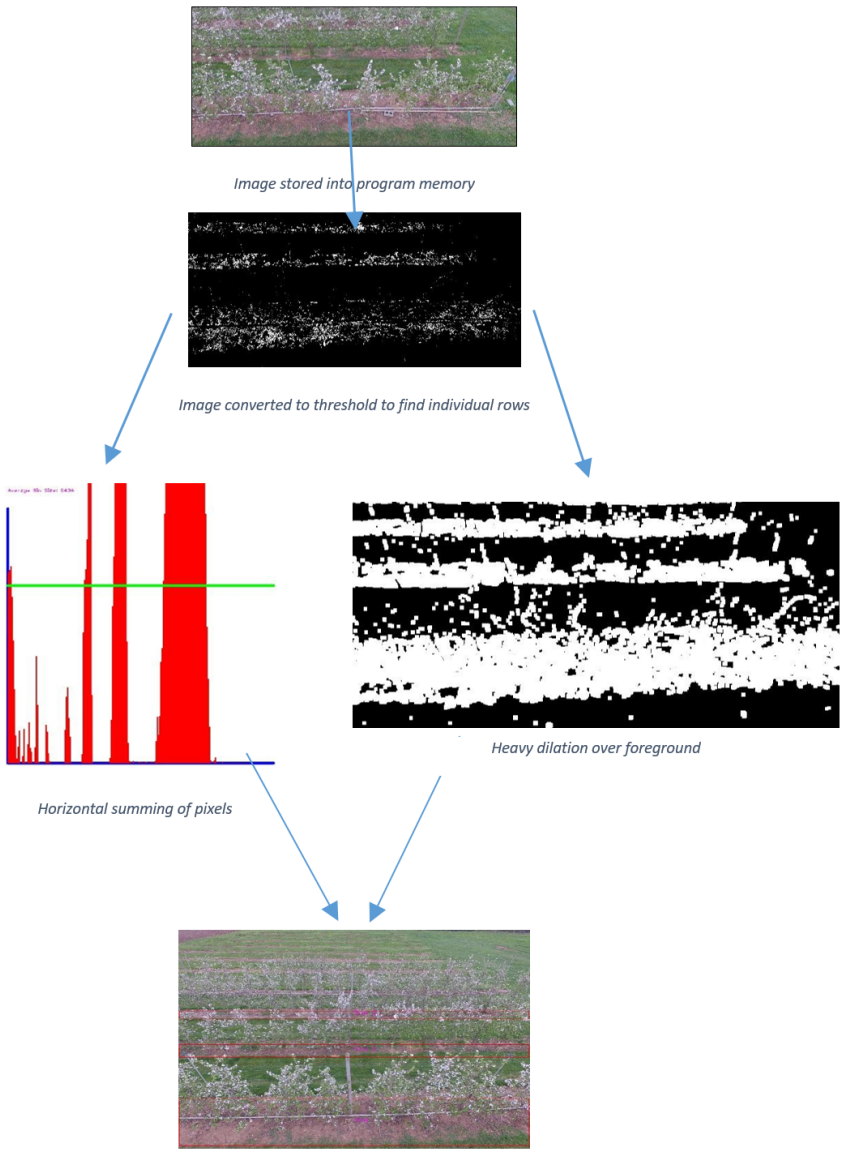


Figure 15: Two Separate Methods Used to Locate First Three Rows

The desired outcome was to filter out any green and keep earth shades in the image. The first strategy took advantage of a heavy dilation to merge neighboring pixels together. The OpenCV functions cv::dilate(), cv::erode(), and cv::medianBlur() were used in combination to remove smaller objects. Each remaining object was then sorted to see whether or not it met the predetermined characteristics of a row, and then tested for if it was the first row. If the object did

not meet the criteria of a row then it was removed from the queue. Once the first row was found, the image was cropped and analyzed for specific tree attributes.

The second strategy took advantage of the relative spacing of the pixels as seen in Figure 15. Rather than focusing on converging neighboring pixels together, the image was analyzed to sum how many white pixels were within each region. Each image was 4000 pixels wide by 3000 pixels tall. The image was sliced into 200 horizontal bins, leaving 15 rows of pixels in each bin. Each of the 15*4000 pixel bins were counted to see whether the pixel value was part of the binary foreground. If it was, the pixel was counted. The total count of each of the 200 foreground bins was stored for comparison. Figure 16 illustrates a clear pattern seen after the binning is complete. It becomes fairly recognizable that the areas corresponding to a higher-than-average pixel count (green line) are rows. Moreover, the closest row to the UAV always corresponds to the right-most output of the horizontal binning.

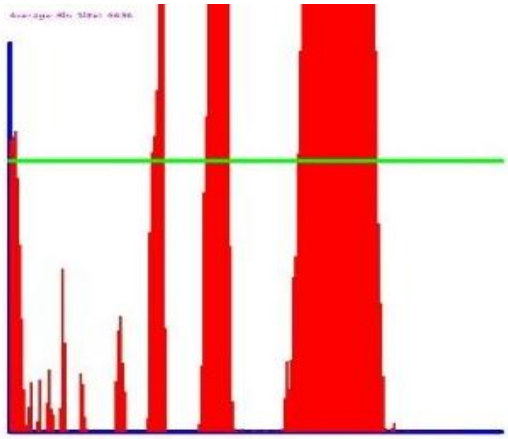


Figure 16: Horizontal Binning Image Output

Using horizontal binning for row recognition proved to be a valuable and highly accurate technique. Sorting the image foreground based on areas worked well for all blossom images, but

failed to report accurate results for apple orchards. The main difference between the two algorithms was their response to disorder within an image. The apple orchard images were not maintained with the same level of care the blossom images were. The multiple ground patches within the binarized foreground image of trick the algorithm into thinking there are multiple first rows. This leads the algorithm into thinking the image contained bad data, and therefore no decisions for row location will be made. The horizontal binning technique is much more forgiving, since only the relative vertical locations of the foreground objects are of concern. The horizontal binning method was chosen for all row recognition purposes, and was seen to report highly accurate results. In most cases, the findRowLocations function was able to find the first three or even four rows. The image was next cropped to show only the first row and processed to find how many trees were in the first row.

Discovering individual trees proved to be quite challenging, and had mixed results. While the majority of blossom images could be processed for tree identification, apple images at harvest had several traits that made tree inspection difficult. Removing the background proved challenging since both the tree's canopy and surrounding grass had almost identical colors. Tree bunching made autonomously finding where one tree ended, and another began almost impossible for some images. Figure 17 below illustrates when tree bunching made discovering individual trees impossible. Even the human eye has trouble identifying where one tree ends and the next begins.

Figure 17: The Effects of Tree Bunching for AOI

The algorithms findTreesAreaMethod and findTreesTrunkMethod in the BlossomOrchard class were used to identify two apple tree traits; the tree trunk coordinates and the outer edge of the tree's canopy. Used together, the AOI algorithms were able to locate where one tree began and the next ended. In cases where the edge of one tree's canopy drastically intersected with the edge of its neighbor, the algorithms would report the pair as a single tree. Figures 18 and 19 below depicts the output from the BlossomOrchard's AOI algorithms.

Figure 18: Row Detected by the Blossom Orchard Member Function findRowLocations()

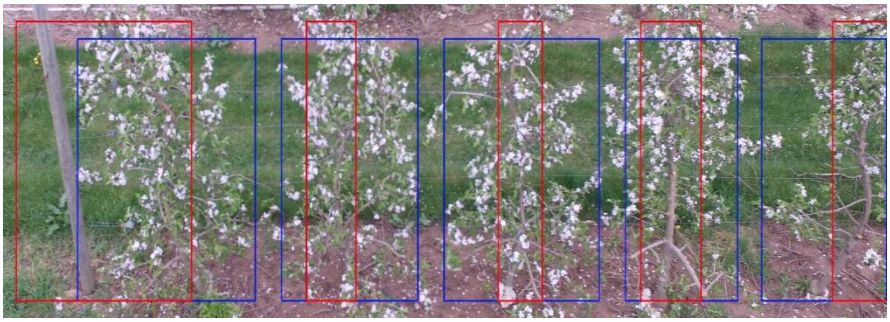


Figure 19: AOI for Individual Blossoming Apple Trees

When used in conjunction, the ‘findTree’ functions listed above were able to accurately locate trees. Identifying the start and end coordinates of a tree allowed the program to collect individual tree characteristics such as blossom canopy coverage and mean blossom size. However, having the AOI algorithms search for individual trees was found to be highly inaccurate in some cases. Figure 20 below depicts the output only from the findTreesAreaMethod, and how the algorithm grouped the data from its search as two large trees, not multiple smaller ones.

Figure 20: The Effect of Tree Bunching for Individual Tree Inspection

After individual tree locations were mapped (or attempted), the last task for the BlossomClass was to find the location, area and count of how many blossoms were on each tree. This last set of algorithms served to test whether the photos from the UAV could locate blossoms and predict whether a certain tree would need thinning. Due to the complications that tree bunching raises, the locations, areas, and count of blossoms were reported on a per row basis, not per tree. Figure 21 below illustrates the output from the findBlossomCount function. Figure 21 depicts the count on a per row basis, not per tree.

Figure 21: Identification of 1074 Blossoms in Blossoming Apple Image

Mean blossom size and standard deviation was calculated to identify areas of the image with large bunches of blossoms. If a blossom was reported as being over two standard deviations above the mean, then it was safe to assume the neighboring blossoms were bunched together. This property was easily detected, and was used to determine whether the trees within a given image needed thinning or not. In the case where individual tree identification was possible, the function could predict blossom bunching on a per tree basis instead.

One interesting attribute of the blossom images was the presence of other agricultural equipment. In many cases of blossom binarization, objects such as posts, poles, and tubes would sometimes be identified incorrectly. Figure 22 below depicts a binarized blossom image where three cinder blocks, a pole and post slipped into the foreground.

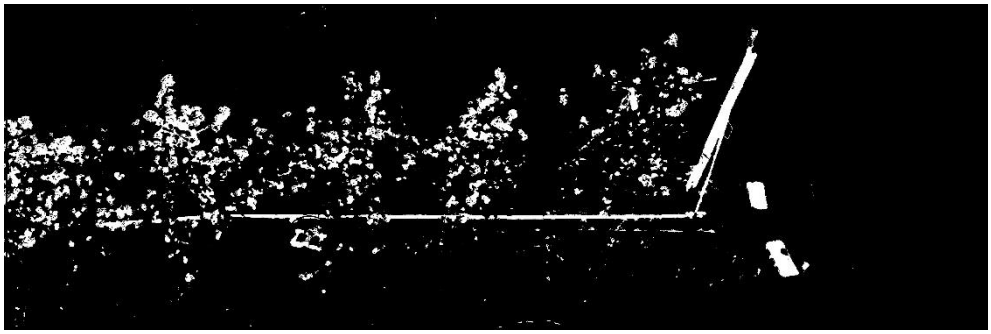


Figure 22: Unwanted objects Within Foreground of Binarized Image

The undesired farm equipment has larger object areas than the individual blossoms. Finding the undesired objects was done by iterating through the blossom vector array and comparing the object's size against two standard deviations above the interquartile mean. Once the objects were removed from the binarized image, blossom statistics could be gathered.

Chapter 5: Image Processing Apple Trees During Harvest

Apple orchard image processing was conducted similar to blossom processing. A UAV was flown above the tree canopies and between each orchard row. The UAV would take photos every six feet with a camera pitch of 45 degrees downwards. Figure 23 below depicts a typical image taken from the Phantom III for apple tree processing.

Figure 23: UAV Image Capture of Apple Orchard at Harvest

Apples grow differently based on their tree's characteristics. If a tree has many blossoms, the tree will tend to grow a large quantity of smaller apples. Conversely, if a tree has fewer blossoms, the tree will tend to grow a smaller quantity of larger apples. Depending on what purpose the farmer has for the apples, the apple trees must be cared for differently. The goal for the AppleOrchard class of functions was to determine how accurately each image could be processed using AOI algorithms. The class was assigned three main objectives: identify each of the rows in the image, identify the individual trees, and lastly count all the apples within the image. The AppleOrchard class followed a similar pattern to that of the BlossomOrchard class.

Each image was first analyzed for row locations. Once row locations were discovered, individual trees were identified. Lastly, each image was counted for the number of apples that it contained.

As discussed in Chapter 4, the findRowLocations function was used for both image classes to identify where individual rows were located. Rows were found to be identified very accurately for this set of images as well. Figure 24 below depicts the results of row recognition for the AppleOrchard class (left) and the BlossomOrchard class (right). Notice how even though the orchard is much less maintained when compared to the BlossomOrchard images, the algorithms are still able to find rows. This indicates the algorithm can be used in multiple environments with similar success.

Figure 24: Rows Identification for Apples at Harvest (left) and Blossom Apples (right)

Finding individual trees did not work as well for the AppleOrchard photos as it had for the BlossomOrchard set of images. While locating individual trees may appear an easy task for humans, the task proves much more daunting for a computer to solve. Since no two trees look identical, having an algorithm test for a specific property or color generally yields unfavorable results. While accurately identifying individual apple trees would be helpful in collecting individual tree statistics, finding the exact count of apples per tree is not completely necessary. Since farmers weigh apples by the row, knowing total apple count per tree would not be as useful

as knowing the total apple count per row. Therefore, the strategy to find each tree and the count of apples per tree was abandoned. Instead, only the first row would be found and the count of apples per whole image was determined. Furthermore, since the apple images contained more tree bunching per image than the blossom images, finding the apple count per row was a more accurate statistic to gather as well.

Figure 25 below depicts the typical output of an apple image when counting the number of apples per image. The strategy here is to apply a HSV transformation to the RGB image and binarize the result to check whether or not the image has yellow and red objects within it. If the objects turn out to have small circular red or yellow objects within it, than the image is known to contain red or yellow apples, respectively.

Figure 25: Apple Identified Through HSV Color-Space Conversion and Binarization

Statistics such as mean, median, and standard deviation were gathered per apple image to help identify apple traits per image. Figure 25 illustrates how individual tree identification is nearly impossible for both AOI algorithms as well as the human eye when trees are heavily bunched together. After the count of apples is determined, the AppleOrchard class will return the count, mean apple size, apple standard deviation, and row coordinates to the user, per image. One area that still remains to be improved is accurately counting all apples in a row through still

images. Since some apples are captured by multiple images, they are counted additional times. Different flight or image processing techniques could be developed during future research (as discussed in Chapter 7) to combat this set back.

Chapter 6: Image Processing Christmas Trees Captured by UAV

Christmas tree counting used the Phantom III flown at approximately 200 feet altitude to capture birds-eye-view images of Christmas tree farms. The UAV was flown autonomously to gather data for analysis. The goal of high altitude Christmas tree image capture was to provide farmers alternative methods to more traditional land maintenance and surveillance techniques. Manually counting the number of trees in a certain section will certainly prove a lengthy task if the farm is large. Instead, employing a UAV that can fly missions of 200 feet in altitude would enable a farmer to view his or her farm from an aerial perspective, removing the need to manually inspect a section of his farm to count quantity of trees. If the UAV was paired with computer vision algorithms that could tell the farmer the exact count of trees within a certain area of the field, then the automated process could end up saving the farmer many tedious hours of manually counting trees.

To test this process, a set of computer vision classes and functions were created using OpenCV. A GUI was created to allow a user to select certain areas of the image that he or she wished to find the total quantity of trees within. By allowing the farmer to interact with the images of his field, he or she would be able to map how many trees are in each block of their fields with relatively little effort of their part.

The ‘ChristmasTree’ C++ class was designed to process Christmas tree images taken from high altitude UAV flights. The class had three main tasks. First, the class would read a

user's image of choice into the analysis environment and display it on screen. Next, the class would use drawing functions to allow the user to select on their image exactly the area they would like the program to count trees within. Once the user selected the area he or she wished to have counted, the program would then crop the image to the user's specifications, threshold the content, and count the quantity of trees within that region. The function returned the original image as well as the processed image with red dots over all of the trees counted. The total count was displayed in the top left-hand corner for the user to view. Figure 26 below depicts how a user would select the region of their field they wanted counted.

Figure 26: User's Selection of Region to be Counted (red circles)

The ChristmasTree class was the only one of the three classes that incorporated graphical user interfaces for the user. In the case of apple orchards, using a GUI to inspect each image would not make any sense, as viewing each image would take just as long for a farmer as walking through the orchard themselves. The Christmas tree counting mission does not require low-altitude flight to inspect individual fruit, however. Using a GUI along with overhead images allowed for a user to quickly find the count of trees within a segment of their image, as well as receive visual confirmation that each tree was counted as expected.

Though the tree counting algorithms were able to identify and count individual trees, there were some complications with tree bunching and mistakenly filtering smaller trees. Similar to individual tree recognition for in season apple orchard images, Christmas trees that were very close together posed a problem for the computer vision algorithms. Objects that touched or overlapped were found to be recognized as a single object, rather than two. To combat this, an erosion operation was applied to the image to help detach overlapping objects. While this technique was met with some success, one major complication arose if a heavy erosion was applied. Due to the various shades of green inherent to different species of trees, some trees did not threshold as strong as others. When applying an erosion to the image, smaller objects and less defined trees were occasionally filtered. Figure 27 below illustrates this phenomenon, and depicts how some trees are partially filtered when the image is thresholded.

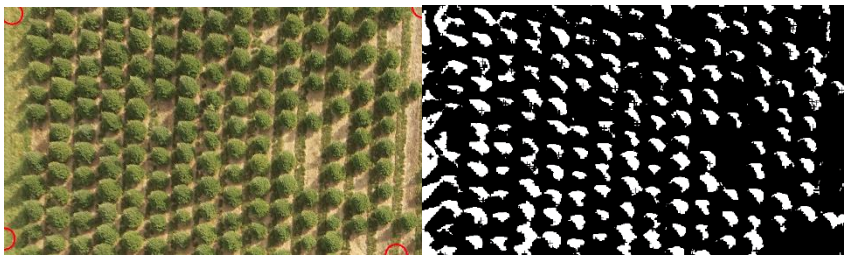


Figure 27: Thresholding of High Altitude Christmas Tree Farm

Another issue was the similar foreground and background colors within the image. Since the grass behind the trees was a similar shade of green, distinguishing between the object of interest and the background proved difficult at times. In some cases, patches of grass that were especially close in color and shape were mistakenly categorized as trees. Figure 28 below demonstrates how the threshold image depicts patches of grass, which in turn are mistakenly added to the total count of objects identified.

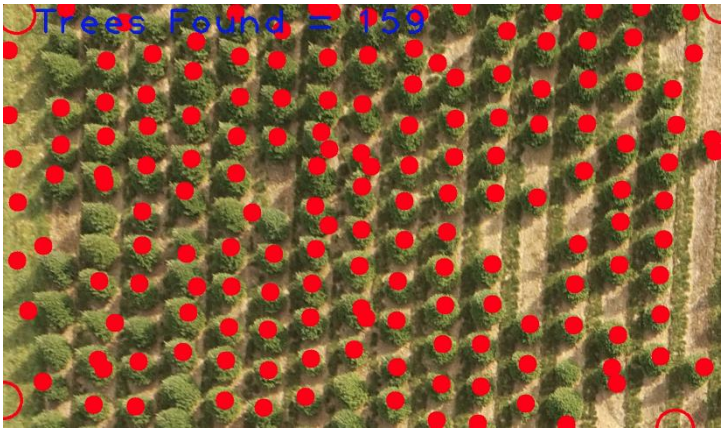


Figure 28: Output of 'ChristmasTree' Class Tree-Counting Algorithm

Despite these setbacks, most trees were accurately processed using the ChristmasTree class. When bunched up trees appeared as one large object in the threshold image, and the area was two standard deviations above the mean, a function was called to automatically divide the object into the correct number of underlying trees. In Figure 28, there are 154 trees within the image. The algorithm found there to be 159, leaving an error of 3.25% between estimated and actual. Potential solutions to the problems seen within Christmas tree image processing as well as apple orchard images will be discussed in the next chapter. While the results of the computer vision algorithms show promise for autonomously processing the contents of agricultural images, much work is still to be done to bring this type of service to market. Algorithm speed, accuracy, as well as efficiency can all be improved with further research.

Chapter 7: Future Work and Research

Though the AOI algorithms developed using OpenCV show promise for autonomously recognizing agricultural attributes such as orchard rows, individual trees and their fruit, there is still much work needed before this small scale autonomous UAV system would be ready for commercial use. While most image characteristics were determined accurately, each set of images contained obstacles that future research should explore. This section will highlight the shortcomings of each C++ class, and where room is left for future improvements to be made.

Most notably, color-based image processing was used as a means of transforming, thresholding, and identifying objects. Color-based image processing is one of two largely explored methods when image processing is the topic of discussion. As mentioned, Machine Learning (ML) is largely used to teach a computer to recognize a pixel-based pattern by exposing the algorithm to a large set of training data. With more training data, the algorithm gains more exposure to the different variations the object can have. In the case of a picture with the word “Hello” inside it, the algorithm would test each letter to see which characters A-Z the image’s pixels best represent. The algorithm would then take its best guess, basing its decision off of the previous data it was fed. Machine learning, while more complex, could prove useful for agricultural images where certain characteristics repeat from one image to another. High altitude Christmas tree recognition would benefit from machine learning algorithms to better determine locations of trees and help reduce miscounting sections of bunched or overlapping trees. Conversely, the stochastic nature of the apple orchard may prevent machine learning from being useful for object recognition here. Due to the large amount of variability seen in apple

images, ML algorithms may not be able to find patterns amongst the low-altitude images as easily as high altitude images.

Selecting accurate flight paths for the UAV will play a critical role in moving this technology forwards. Though the UAV was able to capture crisp images of the orchard at both low and high altitudes, these images were not captured autonomously. Image capture missions must be flown autonomously for the UAV-AOI algorithm combination to be most useful to farmers. The problem currently arises with flying low-altitude missions. Since the UAV relies on GPS to navigate itself when set in its autonomous mode, the GPS must be highly accurate to fly precision missions. Because obtaining accurate images is contingent upon an unhindered flight path, if the associated GPS drift offsets the drone several feet in either direction, the drone could mistakenly fly into an orchard tree. For low-altitude missions to become autonomous, a real time sensory system could be added to detect whether the UAV is headed towards a collision or not. Alternatively, if GPS accuracy improves to just a couple of feet, then low-altitude missions will be one step closer to autonomy without a sensory system.

The most accurate data was gathered from the blossom images. Rows, individual trees, and blossom density could all be found. Blossom images yield better results due to their even spacing between rows and trees. Unlike the in-season apple images, the tree's white blossoms stood out and made identifying where the canopy of one tree ended and the next started a simple color transformation task. Even when neighboring tree canopies overlapped, individual trees could still be found through searching for the tree's trunk and then comparing it to the location of its neighbor's trunks. If excessive bunching between trees is observed, then the results appeared similar to Figure 29 below. Better methods must be developed to deal with image analysis for overlapping trees before this system could be implemented.

Figure 29: Incorrect Identification of Individual Christmas Trees

The in-season apple orchard images proved to be the most difficult for AOI. Unlike the blossoming apple trees, the in-season apple trees had large sections of bunching, further exacerbated by poorly maintained rows. This additional clutter took away from being able to identify the start and end of single trees. While rows could still be found with high levels of precision, accurately counting apples within a row through still images proved problematic. Since images were only taken six feet laterally apart, many apples appear as repeats in their neighbor's images. A possible alternate method could be video capture instead of discrete image capture for the apple images, repeated apples would not appear within the count.

Lastly, high altitude Christmas tree counting could be improved two fold. Since the objective of tree counting was to allow a user the ability to count trees within a certain region of their farm, better GUI functions to zoom in and scroll over the image would help the farmer navigate through their image more accurately. Better methods to de-bunch cluttered trees still remains for improvement as well. Machine learning methods should be explored to predict tree count as well, but may be problematic due to limited training data and large variability of field layout.

Overall, this research project aimed to develop methods for autonomous object inspection. The findings have shown that the UAV-AOI system has the potential to find image characteristics even in very stochastic of environments. This system could be built to have full

autonomy, allowing for minimal effort on the farmer's part. The UI could allow novice computer users to take advantage of the output from the high level computer vision algorithms created with no programming knowledge. Future work could include aiming to have the UAV make a full autonomous low-altitude flight and then send the data to the terminal computer for image processing. A database could also be developed to store the information found within each of the images, not just for the program's execution. For easy access, an online SQL database could store the information and act as the login point for farmers to see their field's analysis. Using Python based OpenCV with SQL for a database would be an effective combination, and easily ported from the C++ source code created for this research project.

Over 2500 lines of code were developed within the BlossomOrchard, AppleOrchard, and ChristmasTree classes. Full source code for the 'Image Analysis Environment' can be found in full in Appendix A. The three constructed C++ classes, examples, and source photos can all be found on https://github.com/natsnyder1/AppleOrchard.

Appendix A: Full Source Code for The Image Analysis Environment

```
// Function Prototypes
	Mat blurImage_Image_Analysis(Mat inputImg);
	Mat convertRGB2HSV_Image_Analysis(Mat sourceImg);
	Mat erodeImage_Image_Analysis(int HSV_ERODE, Mat sourceImg);
	Mat dilateImage_Image_Analysis(int HSV_DILATE, Mat sourceImg);
	void snagKeyboardEvents(int &keyboardChoice);
	void on_trackbar(int, void*);
	void setUpTrackerBars(void);
	void thresh_callback(int, void*);
	void calcBGRHisto(Mat& inputImg);
	void calcHSVHisto(Mat& bgrImg);

	// This Project is intended to be used to do soft coded image analysis. Desired
result is to find values and properties of images using a HSV color transformation

	// These are the changeable HSV Values used for Quick Image Analysis
	int H_LOW = 0;
	int H_HIGH = 255;
	int S_LOW = 0;
	int S_HIGH = 255;
	int V_LOW = 0;
	int V_HIGH = 255;
	int HSV_DILATE = 1;
	int HSV_ERODE = 1;
	int HSV_BLUR = 1;
	int key;
	int ESCAPE_KEY = 27;

	// Boolean Gates
	bool toggleHSV = false;
	bool toggleXmasTrees = false;
	bool countObjects = false;
	bool allowEnterance = false;

	// General Purpose Counter and Matrices
	int _counter = 0;
	Mat BGRImg,HSVImg,ThresholdImg;

	int main()
	{
		// Read in Image
		BGRImg =
imread("Z:\\Desktop\\Honors_Thesis\\ImageAnalysisLab\\Photos\\UnprocessedImages\\Picture2
.jpg");
		// Call the tracker bar function to initialize trackerbars
		setUpTrackerBars();
		// Create a named window to display the image on screen
		namedWindow("ImageWindow", CV_WINDOW_NORMAL); //this is to show image...we
want to output image into file

	while ((key = waitKey(50)) != ESCAPE_KEY)
```

```
	{
	snagKeyboardEvents(key);

			// We need to do a conversion between the BGR Full color image and the HSV image
			// Use HSV image and run through the inRange function to filter out unwanted pixels. Creates a binarized image
			ThresholdImg = convertRGB2HSV_Image_Analysis(BGRImg); // Returns the threshold image using the trackervalues we updated
			ThresholdImg = blurImage_Image_Analysis(ThresholdImg);
			// We need to check if the user wishes to toggle an event on or off with a key on his keyboard. Setup which keys youd like to use in snagKeyboardEvents function. This is simply a skeleton below where any keyboard event code can be inserted into
			if (toggleHSV)
			{
				imshow("ImageWindow", ThresholdImg);
				// do something

			}
			else
			{
				imshow("ImageWindow", BGRImg);
				// do something
			}
			if (countObjects)
			{
				countObjects = !countObjects;
				// do something
			}

			if (toggleXmasTrees)
			{
				toggleXmasTrees = !toggleXmasTrees;
				// do something
			}

	}
	// clean up
	destroyAllWindows();

	}

	Mat convertRGB2HSV_Image_Analysis(Mat sourceImg)
	{
		// HSV Values for Morphological Operations
		Mat Threshold, HSVImg, Threshold_Temp;
		cvtColor(BGRImg, HSVImg, CV_BGR2HSV);
		inRange(HSVImg, Scalar(H_LOW,S_LOW,V_LOW), Scalar(H_HIGH,S_HIGH,V_HIGH), Threshold_Temp); // turn hsv into threshold
		Threshold_Temp = dilateImage_Image_Analysis(HSV_DILATE, Threshold_Temp); // dilate
		Threshold = erodeImage_Image_Analysis(HSV_ERODE, Threshold_Temp);   // erode
		return Threshold;
	}
```

```
	Mat dilateImage_Image_Analysis(int HSV_DILATE, Mat sourceImg)
	{
		Mat outputImage;
		// Create the structuring element for dilation
		Mat element = getStructuringElement(0, Size(2 * HSV_DILATE + 1, 2 *
HSV_DILATE + 1),
			Point(HSV_DILATE, HSV_DILATE));
		dilate(sourceImg, outputImage, element);
		return outputImage;
	}

	Mat erodeImage_Image_Analysis(int HSV_ERODE, Mat sourceImg)
	{
		Mat outputImg;
		// Create the structuring element for erosion
		Mat element = getStructuringElement(1, Size(2 * HSV_ERODE + 1, 2 *
HSV_ERODE + 1),
			Point(HSV_ERODE, HSV_ERODE));
		// Erode the image using the structuring element
		erode(sourceImg, outputImg, element);
		return outputImg;
	}

	Mat blurImage_Image_Analysis(Mat inputImg)
	{
		// Simply blur the image to different values, which will be used to
determine if the blurred filter Image will
		// lead to better image recognition
		Mat blurredImage;
		medianBlur(inputImg, blurredImage, HSV_BLUR);
		return blurredImage;
	}

	void setUpTrackerBars(void)
	{
		// Need to call the window in the same function as the trackbars themselves
		namedWindow("TrackerBarWindow", CV_WINDOW_NORMAL);

		// HSV Values
		createTrackbar("H_MIN", "TrackerBarWindow", &H_LOW, H_HIGH, on_trackbar);
		createTrackbar("H_MAX", "TrackerBarWindow", &H_HIGH, H_HIGH, on_trackbar);
		createTrackbar("S_MIN", "TrackerBarWindow", &S_LOW, S_HIGH, on_trackbar);
		createTrackbar("S_MAX", "TrackerBarWindow", &S_HIGH, S_HIGH, on_trackbar);
		createTrackbar("V_MIN", "TrackerBarWindow", &V_LOW, V_HIGH, on_trackbar);
		createTrackbar("V_MAX", "TrackerBarWindow", &V_HIGH, V_HIGH, on_trackbar);

		//Dilate, Erode, and threshhold
		createTrackbar("Dilate", "TrackerBarWindow", &HSV_DILATE, 50, on_trackbar);
		createTrackbar("Erode", "TrackerBarWindow", &HSV_ERODE, 50, on_trackbar);
		createTrackbar("Blur", "TrackerBarWindow", &HSV_BLUR, 50, on_trackbar);
	}

	void on_trackbar(int, void*)
	{
		/*
		When you drag the trackerbar across the screen, the referenced value you
enter is used to gauge.
```

```
		*/
		// We want to make sure these values are never zero
		if (HSV_DILATE == 0)
			HSV_DILATE = 1;
		if (HSV_ERODE == 0)
			HSV_ERODE = 1;
		if (HSV_BLUR % 2 == 0)
			HSV_BLUR += 1;
	}

	void snagKeyboardEvents(int &keyboardChoice)
	{
		// hitting the keyboard key t will show the HSV image and not the regular
image
		if (keyboardChoice == 't')
		{
			toggleHSV = !toggleHSV;
		}
		// by toggling the keyboard key f we will run a full analysis on the
filefolder of images
		if (keyboardChoice == 'f')
		{
			toggleXmasTrees = !toggleXmasTrees;
		}
		// by pressing the letter c, count the objects
		if (keyboardChoice == 'c')
		{
			countObjects = !countObjects;
		}
	}
```

BIBLIOGRAPHY

[1]"Misconceptions about UAV-collected NDVI imagery and the Agribotix experience in ground truthing these images for agriculture." Agribotix. N.p., n.d. Web. 02 Mar. 2017. <http://agribotix.com/blog/2014/06/10/misconceptions-about-uav-collected-ndvi-imagery-and-the-agribotix-experience-in-ground-truthing-these-images-for-agriculture/>.

[2] Smith, Craig. "19 Interesting Drone Statistics and Facts." DMR. N.p., 02 Feb. 2017. Web. 2 Apr. 2017. <http://expandedramblings.com/index.php/drone-statistics/>.

[3] "Buy Phantom 3 Standard | DJI Store." CreateDJI. N.p., n.d. Web. 02 Mar. 2017. <http://store.dji.com/product/phantom-3-standard>.

[4] Joshi, Prateek, David Millán Escrivá, and Vinícius Godoy. OpenCV by example enhance your understanding of Computer Vision and image processing by developing real-world projects in OpenCV 3. Birmingham: Packt Publishing, 2016. Print.

[5] "Basic Structures." Basic Structures — OpenCV 2.4.10.0 documentation. N.p., n.d. Web. 02 Mar. 2017. <http://docs.opencv.org/2.4.10/modules/core/doc/basic_structures.html>.

[6] "HSL and HSV." Wikipedia. Wikimedia Foundation, 01 Mar. 2017. Web. 02 Mar. 2017. <https://en.wikipedia.org/wiki/HSL_and_HSV>.

[7] "HSL, HSB and HSV Color: Differences and Conversion." Codeitdown.com. N.p., 19 Feb. 2014. Web. 10 Apr. 2017. <http://codeitdown.com/hsl-hsb-hsv-color/>.

[8] "C Templates Tutorial." C Templates Tutorial. N.p., n.d. Web. 02 Mar. 2017. <http://users.cis.fiu.edu/~weiss/Deltoid/vcstl/templates>.

[9] "Basic Thresholding Operations." Basic Thresholding Operations — OpenCV 2.4.13.2 documentation. N.p., n.d. Web. 02 Mar. 2017. <http://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html>.

ACADEMIC VITA

Academic Vita of Nathaniel Snyder
Njs5324@psu.edu

Education: The Pennsylvania State University, Schreyer Honors College
Major: B.S in Mechanical Engineering
Honors: Mechanical Engineering

Thesis Title: Image Processing For Apple Orchards and Christmas Trees Using Unmanned Aircraft
Thesis Supervisor: Dr. H.J. Sommer III

Work Experience: Northrop Grumman Corporation
Date: May 31 2016 – August 8th 2016
Title: SIT&E Intern for Advanced Extremely High Frequency (AEHF) Satellite
Description: DoD Secret Security Clearance Position, Flight Six AEHF
Institution/Company: Redondo Beach, California
Supervisor's Name: Joe Fuller

Grants Received:
William and Wyllis Leonhard Engineering Scholars Program, 2014
William and Wyllis Leonhard Engineering Travel Grant, 2015
Schreyer Honors College Travel Grant, 2015

Awards:

President's Freshman Award

President Sparks Award

Arthur Wilcox Memorial Scholarship Award

Dean's List: 2013-2017

International Education:

The University of Auckland, New Zealand