12-2015

3D reconstruction of close range objects using free and open source software and Raspberry Pi technologies

Juan Lorenzo Monrreal
The University of Texas Rio Grande Valley

3D RECONSTRUCTION OF CLOSE RANGE OBJECTS USING FREE AND OPEN SOURCE SOFTWARE AND RASPBERRY PI TECHNOLOGIES

A Thesis

by

JUAN LORENZO MONRREAL

Submitted to the Graduate College of
The University of Texas Rio Grande Valley
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2015

Major Subject: Computer Science

3D RECONSTRUCTION OF CLOSE RANGE OBJECTS USING FREE AND OPEN SOURCE SOFTWARE AND RASPBERRY PI TECHNOLOGIES

A Thesis
by
JUAN LORENZO MONRREAL

COMMITTEE MEMBERS

Dr. John Abraham
Chair of Committee

Dr. Bin Fu
Committee Member

Dr. Emmett Tomai
Committee Member

Dr. Timothy Wylie
Committee Member

December 2015

ABSTRACT

Monrreal, Juan Lorenzo, 3D Reconstruction of Close Range Objects Using Free and Open Source Software and Raspberry Pi Technologies. Master of Science (MS), December, 2015, 72 pp., 2 tables, 37 figures, references, 29 titles.

Existing 3D rendering open source software along with Raspberry Pi technology can be used to create an affordable method and workflow for time efficient, accurate and quality scans for 3D printing. The emergence of technology spurs a technological community working to progress in a collaborative effort. This brings a potential to the possibility of efficient and economical solutions to emerging problems, in this case, the ability to render three dimensional scans using free and open source software as well as Raspberry Pi technology. The focus of this paper will be divided into three different aspects including the background needed to present the role that photogrammetry plays, the development of a software using Python along with open source software, and finally its collaboration with Raspberry Pi using networking techniques to create the final 3D render. As with all technology, the possibility for improvement will be discussed.

DEDICATION

The completion of my master studies would not have been possible without the love and support of my family. My father, Lorenzo Monrreal Jr., my mother, Eva Monrreal, and my siblings, Becky, Adan and Mina who always saw something in me that pushed me to prove it to myself. Thank you for your love and support.

ACKNOWLEGEMENTS

I will always be grateful to Dr. Abraham, chair of my thesis committee, for all his mentoring and advice not only within the scope of this thesis but the scope of my Computer Science career. From believing in my proposal's inception to the intermediate ventures while completing it, his input has been invaluable. I would also like to thank my thesis committee members whom took the time to overlook and advise on my proceedings.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

The field of 3D printing is quickly growing and with it the methods of scanning objects for replication or modification. Existing 3D scanners are not very efficient because of limitation of range and time of scanning. To try and alleviate this problem we may look to the field of photogrammetry as an alternative method. In order understand its possible contribution, we must first look at what photogrammetry constitutes.

The fundamental principle used by photogrammetry is triangulation. By taking photographs from different locations, so-called "lines of sight" can be developed from each camera to points on the object. The intersection of the different lines of sight creates points in space called a “point cloud” which outlines the three dimensional object. By using this method, we are able to cut down on scan time by using a multitude of cameras. Once the images have been acquired from different angles, we are then able to stich them together using previously developed software that looks at similarities in the photographs to accurately stitch the photographs together. One of the more popular software currently in existence is 123Catch from Autodesk. This software allows you to upload a number of photos then uses cloud computing to generate a three dimensional stich of the photographs. Due to the fact that it is not open source, there is only so much you can do before you run into limitations that restrict the ability for an accurate replication (Bartos, Pukanska, Sarova 6). The same goes for other similar software. It

is at this point that we look to open source software to give us structure and guidance without limitation.

Once the point cloud has been acquired, the points must be accurately connected to create a mesh that correlates with the original object. At this point we may use software such as MeshLab. MeshLab is an open source system for the processing and editing of unstructured 3D triangular meshes. The system is aimed at helping the processing of the typical not-so-small unstructured models arising in 3D scanning. It provides a set of tools for editing, cleaning, healing, inspecting, rendering and converting these kind of meshes (MeshLab 3).

The final aspect of creating the three dimensional object is the texturizing and coloring for accurately replication. Blender will cover this aspect of the workflow. Blender is a free and open source 3D animation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation (Beginning Blender). Users who may reach an advanced level of comfort with the software use Blender's API for Python scripting to customize the application and write specialized tools.

Once the technical aspect of how photogrammetry and open source software will be used in conjunction is understood, it may be worthwhile to look at the hardware or physical aspect. In order to reduce the cost of a multi-camera system, Raspberry Pi cameras will be networked to simultaneously capture images from different angles of the object. The idea for Raspberry Pi first came around with the motivation of trying to teach computers to students without needing to use entire computer systems or traditional settings. This correlates very much with the motivation behind this thesis as the whole point is to find an affordable and efficient method.

CHAPTER II

REVIEW OF LITERATURE

Vectors and Space

In order to understand many of the fundamental mathematical concepts which encapsulate the field of photogrammetry, it will be necessary to take a step back into more basic concepts of linear algebra within the scope of matrices for the most part. We will later see how matrices will play a huge part in developing the code as well.

Since we will be dealing within the scope of a two dimensional as well as three dimensional analysis, it is important to understand the notation that will be used from this point on as well. $\mathbf{R}^2$ will be used to represent the two dimensional real coordinate space. This refers to all the possible real-valued 2-tuple ordered numbers which exist within this space. Within this same context, R^3 will be used to represent three dimensional space. To represent vectors within these spaces we will be using the notation $\vec{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ where a,b and c represent some constant used to show the magnitude of the vector in the different dimensions. It will become important later to understand different representations of a vector such as with the use of unit vectors. For example the previous vector, x, may also be shown using the representation $\vec{x} = a\hat{\imath} + b\hat{\jmath} + c\hat{k}$. The question may become, why so many different representations? The answer is that it really just depends on the application. For example, within photogrammetry we will be dealing with

vector space a lot. Due to this, we will have to find a way to represent lines within this space much more effectively. This is where line parametrization comes into play. From earlier or more basic math we've always known that a line can take the form y=mx + b, but what about when dealing with more than two dimensions. In this case the parametric form allows us to do this. Let us take the next example to show this. Let's say that we want to draw the line that goes through the points represented by to location vectors where $\overrightarrow{P1} = \begin{bmatrix} -1 \\ 2 \\ 7 \end{bmatrix}$ and $\overrightarrow{P2} = \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}$. In two dimensions this may have been very simple as you would only have x and y coordinates to deal with and you would find the difference between the two points to eventually find the slope and so on. Now with three dimensions it is a bit more complicated yet there is a form to do this. By using the following form $L = \{\overrightarrow{P1} + t(\overrightarrow{P1} - \overrightarrow{P2}) | t \in R\}$ we are saying that the line can be found by finding the different between point 1 and two and multiplying by some scalar t and adding one of the point vectors, as long as t belongs to the set of real numbers. This may be difficult to see unless we substitute the original vectors into this form. So if we use the form and substitute our points we get $L = \{\begin{bmatrix} -1 \\ 2 \\ 7 \end{bmatrix} + t \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix} | t \in R\}$, already having subtracted P2 form P1. This then produces the parametric form of the line x=-1-t, y=2-t, and z=7+3t which allows us to find points along this line in three dimensions.

As we move further towards some of the more specific aspects matrix manipulation and characteristics with respect to photogrammetry we must also understand the subject of linear subspaces. We may start this discussion by first understanding what a subspace is. We may look for example at how a V may be subspace of R^n. This is a way of generalizing that V belongs to the set of real numbers in n number of dimensions. There are three basic conditions that must be met in order for V to be a subspace of R^n. The first is that V must contain a zero vector. The

second condition says that for any vector in V we should be able to multiply it by some scalar and get another vector in V. This is known as closure under scalar multiplication. The final condition says that the sum of any two vectors in V should be another vector in V. This is known as closure under addition. These conditions prove that V really is a subset as a subset should be bound by certain rules that are true for all values within that set. This will come into play further when we look at finding epipolar lines which are bound by common factors. It is important to note that spans can also be subspaces of the defined space as long as they follow the three previously defined rules.

At this point it may be worth reviewing some of the basic mathematical operations as later concepts will depend upon understanding of this. The first is understanding the dot product of two vectors. We can generalize the definition by stating that the dot product of any two vectors will be equal to a scalar. So we can say that $\vec{a} \cdot \vec{b} = \begin{bmatrix} a1 \\ \vdots \\ an \end{bmatrix} \cdot \begin{bmatrix} b1 \\ \vdots \\ bn \end{bmatrix} = a1b1 + \cdots + anbn$ such that this equals a scalar. To further understand this and other operations the length of a vector can be generalized as $\left|\left|\vec{a}\right|\right| = +\sqrt{a1^2 + a2^2 + \cdots + an^2}$.

It is important to now take some time to talk about the angle between different vectors as this will become important when describing the angle between different line describing the same object from different angles. By doing some vector manipulation along with using some of the inequality concepts learned from a previous basis it is found that $\vec{a} \cdot \vec{b} = \left|\left|\vec{a}\right|\right| \left|\left|\vec{b}\right|\right| \cos\theta$ such that Θ is the angle between vectors a and b.

As the program will not only deal with vectors as a means of defining our images but the multiple view, it is important to also be able to define planes mathematically. We can start by first referring back to what we have always understood to be the algebraic equation of a plane

which is Ax+By+Cz=D where any point (x,y,z) that satisfies the condition lies on the plane. But what if we wanted to define it more generally in terms of vectors and space. In this case we will look at how we can define it in terms of a point and normal vector to the plane. In order to understand this it must be understood that for a vector to be normal to the plane, it must be orthogonal to all other vectors lying on that plane. So if we think about this, any vector on the plane may be found by using any other two points found on the plane. So for example if we take the point x defined by the position vector (x,y,z) and we subtract the point x_0 defined by the positional vector (x_0, y_0, z_0) we will define the new vector found on the plane as $(x - x_0, y - y_{0,} z - z_0)$. So by definition the dot product of the normal vector with this vector should be equal to 0, or in other words $\vec{n} \cdot (x - x_0, y - y_{0,} z - z_0) = 0$. By then distributing this using normal vector we get the final equation of the plane to be $n_1(x - x_0) + n_2(y - y_0) + n_3(z - z_0) = 0$. By then using any normal vector and point we can now find the equation of the plane that makes those two conditions true. One thing that may not be obvious right away but can be found is that the normal vector can also be found from the plane equation. So where the plane equation is given by Ax+By+Cz=D, the normal vector is equal to $A\hat{\imath} + B\hat{\jmath} + C\hat{k}$.

It is then important to make sure and review the cross product as compared to the dot product of vectors. One of the major differences is that the cross product is only defined in R^3. This becomes very important as that is the space for the most part that we will be dealing with. Also, different from the dot product, is the fact that the cross product produces another vector, which is orthogonal to the two vectors in question. For example $\vec{a} \times \vec{b}$, where $\vec{a} = \begin{bmatrix} a1 \\ a2 \\ a3 \end{bmatrix}$ and $\vec{b} = \begin{bmatrix} b1 \\ b2 \\ b3 \end{bmatrix}$, would take the form of $\begin{bmatrix} a2b3 - a3b2 \\ a3b1 - a1b3 \\ a1b2 - a2b1 \end{bmatrix}$. One thing that might prove interesting by

this definition is that according to our previous definition of the dot product, the dot product of $\vec{a} \times \vec{b}$ with either $\vec{a}$ or $\vec{b}$ should be equal to zero since they are orthogonal.

At this point we'd also like to make sure we understand that there is a relationship between the angle between two vectors and their cross product in the case that this is what we have to work with. From proof we find that the result is that $\left|\left|\vec{a} \times \vec{b}\right|\right| = \left|\left|\vec{a} \times \vec{b}\right|\right| sin\theta$. This proves to be very interesting as it is very similar to the dot product yet we use sine instead of cosine which also makes for a good way to remember them.

We will now start to connect all the previous relationships to produce some other useful aspects to know. It is important to always keep in mind that this will be projected onto the topic of photogrammetry as it applies to three dimensional space. Let us for example look at the distance between any given point and a plane. Now, it is important to keep in mind that although an infinite number of distances exist due to the distance to any part of the plane, for this purpose we are talking about the minimum distance. As previously discussed this would be found by the perpendicular vector off the plane connecting to that point. So we can start by either identifying a point by its coordinates as $(x_{0,}y_0, z_0)$ or even by its positional vector as $x\hat{\imath} + y\hat{\jmath} + z\hat{k}$. Now if we identify another point that is on the plane we may start working towards a formula for the solution. So if we have another point whose coordinates are (x_1, y_1, z_1), the vector between these two points can be found by subtracting their two vectors as previously discussed. So this new vector may be defined as $\vec{f} = (x_0 - x_1)\hat{\imath} + (y_0 - y_1)\hat{\jmath} + (z_0 - z_1)\hat{k}$. The question then becomes how does this factors in? Well if we used this vector in conjunction with the distance vector formed by the point in question and the plane this would form a right triangle. The formula for the distance would then become $d = \left|\vec{f}\right| cos\theta$. This then points to one more question, since we don't know θ, how do we approach a closer solution. It is here that we start

to tie all previous relationships. If we think about it, the angle between this new vector and $\vec{f}$ is the same as the angle between the normal vector. By doing a little manipulation, as follows $d = \frac{\vec{n}\vec{f}cos\theta}{|\vec{n}|}$, we end up with a very familiar relationship. The numerator is really just the dot product of the normal vector and this new vector, so we can rewrite the formula as $= \frac{\vec{n}\cdot\vec{f}}{|\vec{n}|}$. If we then proceed one last step and actually follow through with the dot product and magnitude of these vectors we get $d = \frac{Ax_0 - Ax_1 + By_0 - By_1 + Cx_0 - Cy_1}{\sqrt{A^2+B^2+C^2}}$. Looking closely at the previous formula, we finally see that the relationship goes right back to the equation of a plane. So the shortest distance between any point and a plane is given by $d = \frac{Ax_0 + By_0 + Cx_0 - D}{\sqrt{A^2+B^2+C^2}}$.

All of this then brings us to the topic of matrices. We will be dealing with many systems of equations and need to understand where matrices play a part in optimizing the process for solving these systems. One such optimization is known as Reduced Row Echelon Form. There are a few basic credentials to reducing a system of equations to this form as will be discussed.

Let's take for example the following system of equations, $\begin{matrix} x_1 + 2x_2 + x_3 + x_4 = 7 \\ x_1 + 2x_2 + 2x_3 - x_4 = 12 \\ 2x_1 + 4x_2 + 6x_4 = 4 \end{matrix}$. This system of equations can easily be represented as a matrix by taking into account their coefficients as well as their equal values. So we may start setting up a matrix without losing any of the information about the system in the following form, $A = \left[\begin{matrix} 1 & 2 & 1 & 1 \\ 1 & 2 & 2 & -1 \\ 2 & 4 & 0 & 6 \end{matrix} \middle| \begin{matrix} 7 \\ 12 \\ 4 \end{matrix}\right]$. Without going to in depth, and taking into account that we can manipulate each row and add or subtract from other rows just as we could with a system of equations we may re duce the matrix to Reduced Row Echelon Form which will produce the following $\text{rref}(A) = \left[\begin{matrix} 1 & 2 & 0 & 3 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 \end{matrix} \middle| \begin{matrix} 2 \\ 5 \\ 0 \end{matrix}\right]$.

From the previous we can now speak about the conventions of reduced row echelon form. The last row has been reduced to all zeros, while the preceding rows have leading one coefficients with the rest of their columns filled with zeros. For convention the leading one's are further trailing as we go down the matrix. So the question then becomes, how does this help us? Well aside from reducing our system of equations we can rewrite them in another form of the matrix. So for the previous example we can say that the remaining system of equations

$x_1 = 2 - 2x_2 - 3x_3$
$x_3 = 5 + 2x_4$, can be written as $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 5 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} -3 \\ 0 \\ 2 \\ 1 \end{bmatrix}$. This will eventually lead us to see that the solution for this system of equations is actually a plane defined within their space, which in this case is four dimensional. The same would apply for three dimensional space and may become even easier as we have less to deal with. It is important to note that we may easily use this reduction to completely solve the system for the values of x, y and z with respect to three dimensional space.

Multiple View Geometry Fundamentals

From understanding the fundamentals we transition into understanding where matrix manipulation comes into play. Although we will be looking at how to recreate a figure by using n-views eventually it will serve us best to first understand how we can do so from the perspective of two images.

For most of the algorithms that follow, the main consideration between two images are their point correspondences. So we consider the fact that there exists $x_i \leftrightarrow x_i'$ in two different images. From this we also assume that there exits some sort of camera matrix P and P' which share some sort of correspondence between the common three dimensional points Xi. This is to

say in other words that there exists some PXi=x_i and P'Xi=x_i'. As a result we find that there exists a set of points Xi that projects to the initial given data points. The problem then becomes that we do not know the data points nor do we know the camera matrix and so this becomes problem under question (Zisserman, 11)

Looking at the worst case scenario of not knowing the calibration of the cameras we will fall into the topic of projective transformations. The tool that we will be looking at as it is the most crucial in the reconstruction of two views is the fundamental matrix. We can think of the fundamental matrix as the constraint to which points in different planes find a correspondence. This constraint comes about as a product of the camera centers, the image points and the space point of the two views all being on the same plane, or being coplanar (Zisserman, 11). With this being said we can then say that ${x'_i}^T F x_i = 0$ should be satisfied. As will be discussed in more detail later, F is 3x3 matrix with a rank of 2.

Zisserman discusses the method by which we use the fundamental matrix to reconstruct a scene is consisting of the following steps:

1. Given several point correspondences across two view, we will form linear equations on the entries of F such that ${x'_i}^T F x_i = 0$ is satisfied
2. Find F as the solution to a set of linear equations
3. The camera matrices will be computed from F
4. Once the two camera matrices are known as well as the corresponding image points, find the three dimensional point that corresponds to the image points. This solution is known as the triangulation of the points

Epipolar Geometry Towards the Fundamental and Camera Matrix

The geometry that represents the relationship between two different views is known as epipolar geometry. It is only dependent on the cameras' internal parameters which will discussed in further detail later. Even further, it is the fundamental matrix which is denoted by F that encapsulates this geometry. F is a 3x3 matrix of rank 2. This basically means that if we denote a point as x in one view of the object and x' in a second view of the object, then these points and F should satisfy the relation $x'^T Fx = 0$.

These being the basis it will then be important to understand epipolar geometry and then learn to derive the fundamental matrix as this will be essential in 3 dimensional reconstruction. The ultimate factor is to show that the cameras can be retrieved using F.

We will first take a look at how epipolar geometry works and how it is used. In essence it refers to the intersection of the image planes having the baseline as their axis. When we speak of the baseline we refer to the line joining the two camera centers. If we look at the following figure let's consider a few parameters. Let's denote X as some point in space which may really represent some point on an object. This point may then be denoted as x and x' on two corresponding image views respectfully. We may then say that these three points are coplanar and denote this as plane π.

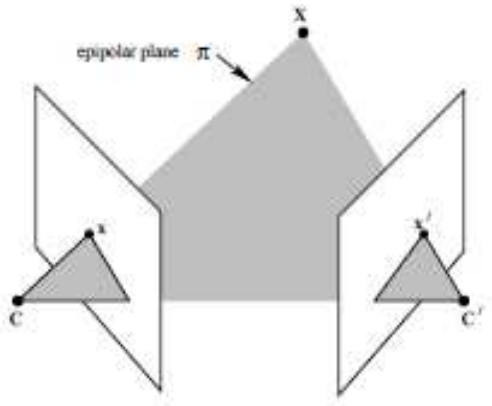


Figure 1. *Coplanar Point Across Image Planes.* (Zisserman, 240)

The question then becomes, how is one image point geometrically related to the other? So we can look at it in terms of the fact that the plane π is created by the baseline and the ray defined by the point x of the first view, which we will pretend to be the point that we do know. As far as the second point, x', all we know is that it lies on the line l' which is created by the intersection of the second image plane and the common plane π as shown by the following figure.

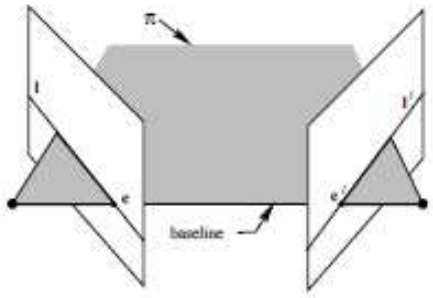


Figure 2. *Common Plane Intersection.* (Zisserman, 240)

To further define the relationship, this newly created line l' is really just the image in the second view of the ray created by x and the camera center c in the first view. This known as the epipolar line corresponding to this x. What this means is that in order to find the corresponding point x' to x, we would not need to search the entire second image plane but rather just the line l'.

There are three basic terms to consider at this point. The epipole is the point of intersection of the baseline with the image plane. The epipolar plane basically is basically just the plane or planes containing the baseline. Therefore there is an infinite number of epipolar planes. Finally, the epipolar line is the intersection of the epipolar plane with the image plane. By these definitions, all epipolar lines intersect at the epipole and therefore corresponding epipolar lines on two image planes will help us find corresponding image points.

This then brings us to the fundamental matrix which we earlier denoted by F. The following will look at how F is derived by mapping a point to its epipolar line. As seen earlier we know that given a pair of images, for a point x in one image there exists an epipolar line in the other image. We also know that the corresponding point x' in the second image must lie on this epipolar line. The fundamental matrix will then show that there is a mapping between these points and their corresponding epipolar lines l' (Zisserman, 241).

We will look at the following figure to understand the first step in creating this fundamental matrix. We can reduce the mapping of a point in one plane to the epipolar line in two steps. The first would be to map the point to another on the second plane which should lie on the epipolar line. Once we have decided that this point be a candidate, the same epipolar line should connect this candidate point to the epipolar line connecting the two camera centers.

Let it be considered that there exists some plane π which does not intersect the camera centers at all. The ray created by the first camera center and x intersects the plane at point X as shown. If this point X is then transferred to the second image plane through the second camera center it is seen that it intersects at a point x' which now has some sort of relationship to the original image point. This is known as transfer via a plane and thus shows us that for all points x_i in the first plane there exists points x_i' in the second image plane which are equivalent through projection. This shows that there is a 2D homography H_π that maps these corresponding image points on two different image planes.

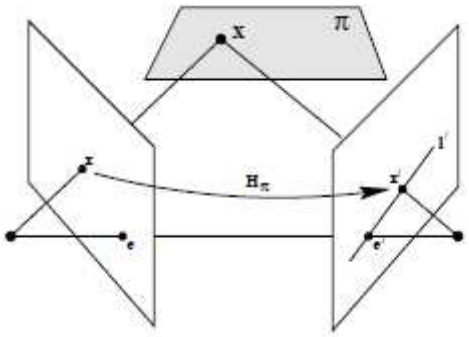


Figure 3. *2D Homography* (Zisserman, 243)

The second step then shows us how finding the epipolar line will help us derive the fundamental matrix. From the previous it is seen that the epipolar line l' can be defined as the cross product of the epipole and the image point x' shown as $l' = e' \times x'$. By using some simple derivations we can also denote the previous as $l' = [e']_x x'$. Finally since we earlier showed that x' can be written as $H_\pi x$ therefore we can write the final derivation as $l' = [e']_x H_\pi x$=Fx where F is the fundamental matrix. Therefore from all of this we get F=$[e']_x H_\pi$ (Zisserman, 243).

Zisserman describes the Fundamental matrix as having certain properties as outlined by the following table. For our purposes will further investigate how the fundamental matrix is then computed as well as the computation of camera matrices. It is important for now to note that from the fundamental matrix we should be able to extract the camera matrices of the two views. In fact if we look at the table as provided by Zisserman, the fundamental matrix corresponding to a pair cameras P=[I|0] and P'=[M|m] is equal to [m]xM (Zisserman, 254).

A Method Towards 3D Reconstruction

In order to proceed with the reconstruction of an object from two views, a few assumptions have to be made. First is that the two views do contain points correlating to the same point in space. Second, for our purposes we do not know where these points lie in space nor do we know the position, orientation or calibration of the cameras. The task at hand then

becomes to find these camera matrices P and P' as well as the 3D points such that the product of the camera matrices against the 3D points should give us their corresponding image points for all image points.

- F is a rank 2 homogeneous matrix with 7 degrees of freedom.
- **Point correspondence**: If $\mathbf{x}$ and $\mathbf{x}'$ are corresponding image points, then $\mathbf{x}'^{\mathsf{T}}\mathbf{F}\mathbf{x} = 0$.
- **Epipolar lines**:
  - $\mathbf{l}' = \mathbf{F}\mathbf{x}$ is the epipolar line corresponding to $\mathbf{x}$.
  - $\mathbf{l} = \mathbf{F}^{\mathsf{T}}\mathbf{x}'$ is the epipolar line corresponding to $\mathbf{x}'$.
- **Epipoles**:
  - $\mathbf{F}\mathbf{e} = \mathbf{0}$.
  - $\mathbf{F}^{\mathsf{T}}\mathbf{e}' = \mathbf{0}$.
- **Computation from camera matrices** $\mathtt{P}, \mathtt{P}'$:
  - General cameras, $\mathtt{F} = [\mathbf{e}']_\times \mathtt{P}'\mathtt{P}^+$, where $\mathtt{P}^+$ is the pseudo-inverse of $\mathtt{P}$, and $\mathbf{e}' = \mathtt{P}'\mathbf{C}$, with $\mathtt{P}\mathbf{C} = \mathbf{0}$.
  - Canonical cameras, $\mathtt{P} = [\mathtt{I} \mid \mathbf{0}]$, $\mathtt{P}' = [\mathtt{M} \mid \mathbf{m}]$, $\mathtt{F} = [\mathbf{e}']_\times \mathtt{M} = \mathtt{M}^{-\mathsf{T}}[\mathbf{e}]_\times$, where $\mathbf{e}' = \mathbf{m}$ and $\mathbf{e} = \mathtt{M}^{-1}\mathbf{m}$.
  - Cameras not at infinity $\mathtt{P} = \mathtt{K}[\mathtt{I} \mid \mathbf{0}]$, $\mathtt{P}' = \mathtt{K}'[\mathtt{R} \mid \mathbf{t}]$, $\mathtt{F} = \mathtt{K}'^{-\mathsf{T}}[\mathbf{t}]_\times \mathtt{R}\mathtt{K}^{-1} = [\mathtt{K}'\mathbf{t}]_\times \mathtt{K}'\mathtt{R}\mathtt{K}^{-1} = \mathtt{K}'^{-\mathsf{T}}\mathtt{R}\mathtt{K}^{\mathsf{T}}[\mathtt{K}\mathtt{R}^{\mathsf{T}}\mathbf{t}]_\times$.

Table 1. *Summary of Fundamental Matrix Properties.* (Zisserman, 246)

We will assume for our purposes that we have an abundance of points that correlate to each other as should be the case if properly capturing images from two or more angles. The reconstruction method can then be reduce to first computing the fundamental matrix. Second we should be able to compute the camera matrices from the fundamental matrix. Finally, for all point correspondences we should be able to compute a related point in space. Although this is a simplification of the process, it will hold true for all cases.

Computing the Fundamental Matrix

Computation of the fundamental matrix comes down to the culmination of several algorithms which we will superficially discuss. The main note to keep in mind is that many of these methods include estimation by using a set of point correspondences between two images. The main method we will be using is known is the 8-point algorithm for computation of the fundamental matrix.

The algorithm actually gets its name from how it works. Since the fundamental matrix is 3x3 with a determined up to an arbitrary scale factor, we will actually be using 8 equations to obtain the solution. The matrix basically works by using the equation of the form [xx' yx' x' xy' yy' y' x y 1]f = 0. With the assumption that we have two points which we may describe as x=[xy1]^T and x' = [x'y'1]^T and a vector f =[F11,F12,F13,F21,F22,F23,F31,F32,F33] which holds all the elements of the fundamental matrix F, by stacking eight of the equations in a matrix A we get Af = 0. We are then able to use Singular Value Decomposition (SVD) to solve the system of equations (Hartley, 581).

Auto-Calibration of Cameras

A topic of great importance which umbrellas over what has previously been discussed is that of camera calibration. Many methods of three dimensional reconstruction call for the initial calibration of cameras by using special methods or objects for calibration. Although this may prove to be very accurate, it may prove to be inefficient especially when having already acquired the images without known the camera parameters or in moments where time was of the essence. We will be looking at how calibration can be done simply by using an image sequence rather than any tedious methods.

For our purposes, of importance is the fact that the internal parameters of the camera or parameter matrix K is the same but unknown for each view. Each camera in this case can then be decomposed as $P^i = K^i[R^i|t^i]$, where the calibration matrix will differ for each view. The essence will be the use then of a homography H as previously discussed to find these correlations. Since the cameras for our purposes all have fixed internal parameters then we can say that the cameras decompose to $P^iH = K^iR^i[I|t^i]$. The approach can then be defined under two steps. The first step in so obtain a projective reconstruction $\{P^i, X_j\}$. The second step is to then determine a homography H form auto-calibration constraints and transform to a metric reconstruction $\{P^iH, H^{-1}X_j\}$ (Zisserman, 459).

Principal Component Analysis

Principal Component Analysis or PCA as we will call it from now on is a very useful tool when we are looking at an immense number of data. For our purposes we have to realize this becomes very useful as the point as to get as a sense a cloud as possible yet without losing quality. So in essence we want both quantity and quality. The main purpose of PCA is to find the principal components of data. So with respect to our three dimensional cloud we can use it to get rid of anomalies or data with too much deviation.

So although we may be used to measuring data conventionally with regards to their correlation from the x and y axis, we now find that we can instead reference their principal components. This makes sense as we will be dealing with three dimensions and not necessarily consistency. So better put, principal components are the directions in which the data is the most spread out towards or concentrated. A good example is to imagine data set out in the shape of an oval. Depending on how we draw a line across this data we will either have data with more or

less of a spread. So in essence if we draw a line from the two furthest points of the oval we will capture more of a spread of the data, making this the principal component.

The focus then becomes on how to use math to find this principal component. When we get a set of data points they can be deconstructed into eigenvectors and eigenvalues. Eigenvectors and values exist in pairs: every eigenvector has a corresponding eigenvalue. An eigenvector is basically a directional vector as the principal component is as well. An eigenvalue is a number that tells us the amount of variance that there is in the direction of that directional vector. By this logic, the eigenvector with the highest eigenvalue is therefore the principal component.

Very important now is to realize that there is not an infinite number of eigenvectors or eigenvalues for that matter. In fact the amount of eigenvectors and values that exist equals the number of dimensions the data set has. It should be noted that we should keep in mind that we will be working in three dimensions. The following example will make more senses of this.

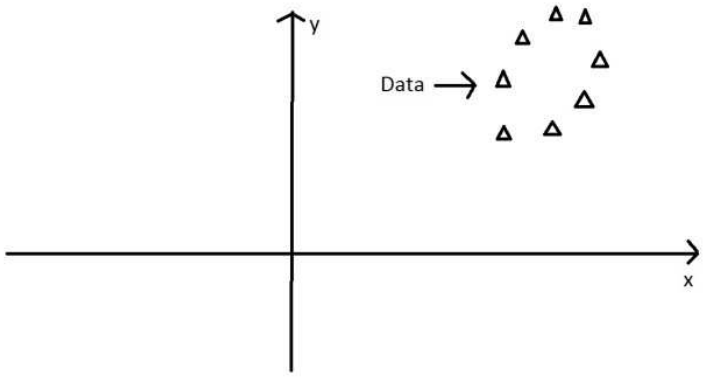


Figure 4. *Example Data Layout*

At the moment the oval is on an x-y axis. x could be one variable of a data set and y the other. These are the two dimensions that the data set currently exists in. From previously we need to remember that visually the principal component would be the line splitting this data on its longest side as shown by the following:

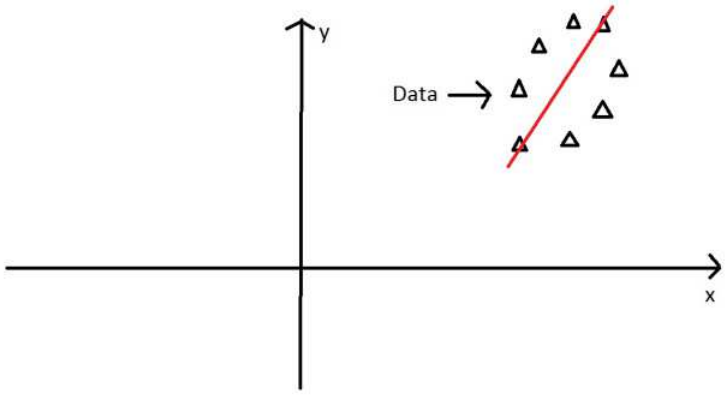


Figure 5. *First Principal Component*

Due to previous logic, since there is only one other dimension that means there is only one other principal component, in this case the line perpendicular to the first as shown. This why the x and y axis are orthogonal to each other in the first place. So the second eigenvector would look like this:

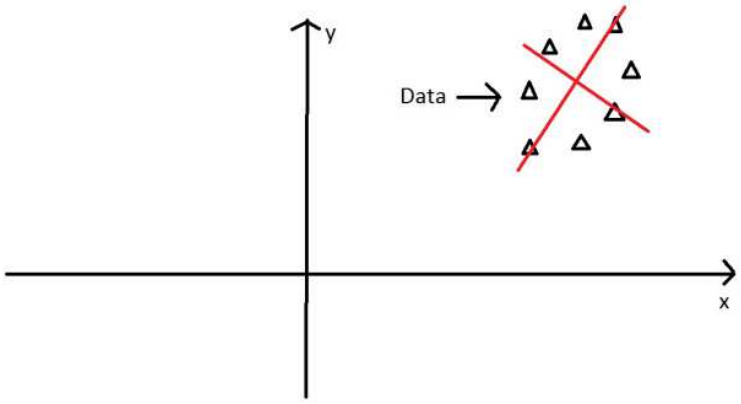


Figure 6. *Perpendicular Principal Components*

The eigenvectors have given us a much more useful axis to frame the data in. We can now re-frame the data in these new dimensions which would show the following:

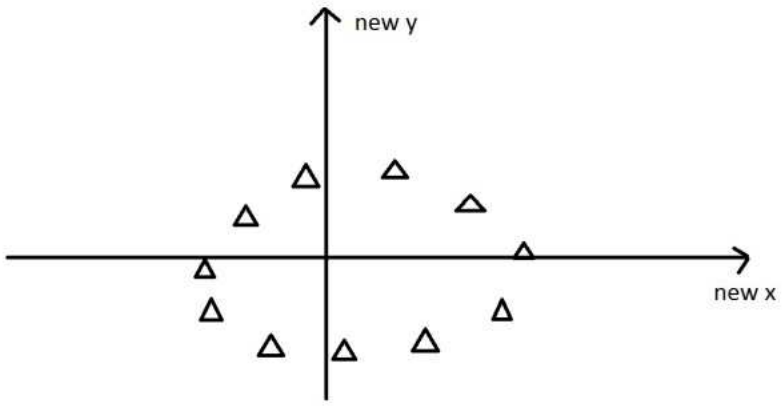


Figure 7. *New Eigenvector Axis*

The question then becomes, how is this useful? Although the new axis better concentrates our data, there is no real logic to what each axis represents. Rather its use will be in manipulating the data to better suit our needs. One such use especially with huge amounts of data is dimension reduction as will be talked about.

Harris Corner Detection

In order to construct the fundamental matrix we need the help of the program itself to find points of interest as well. There are several methods towards finding these. One of the more popular ones is the Harris corner detector as presented by Harris and Stephens in 1988. The basic principal plays on the shifting changes around a point in order to detect whether it is a flat, edge or a corner. The derivation is as follows as presented by Konstantinos.

We look at a specific point as (x,y) and consider $(\Delta x, \Delta y)$ as shift in this point. The auto-correlation function can then be defined as,

$$c(x,y) = \sum_W [I(x_i, y_i) - I(x_i + \Delta x, y_i + \Delta y)]\text{^}2$$

Where I refers to the image function and (x_i, y_i) refer to the points in the Gaussian window. A Taylor expansion is then used to approximate the image as follows

$$I(x_i + \Delta x, y_i + \Delta y) \approx I(x_i, y_i) + [I_x(x_i, y_i) I_y(x_i, y_i)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Where Ix and Iy refer to the partial derivatives of x and y. By then substituting the equations into each other we then get the following

$$c(x, y) = \sum_W (I(x_i, y_i) - I(x_i, y_i) + [I_x(x_i, y_i) I_y(x_i, y_i)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix})^2$$

$$= \sum_W (- [I_x(x_i, y_i) I_y(x_i, y_i)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix})^2$$

$$= \sum_W ([I_x(x_i, y_i) I_y(x_i, y_i)] \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix})^2$$

$$= [\Delta x \Delta y] \begin{bmatrix} \sum_W (I_x(x_i, y_i))^2 & \sum_W I_x(x_i, y_i) I_y(x_i, y_i) \\ \sum_W I_x(x_i, y_i) I_y(x_i, y_i) & \sum_W (I_y(x_i, y_i))^2 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

$$= [\Delta x \Delta y] C(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

where the matrix C(x,y) captures the intensity of the surrounding members of that point. If we then let λ_1 and λ_2 be the eigenvalues of matrix C, they will form a rotational description. From this three cases are considered. If the values are small then that means it is a flat region indicating there is little change in direction. If one value is low and the other is high then there is a high contrast indicating that we are looking at an edge. Finally, if both values are high, this indicates there is much change in every direction indicating an edge.

Scale Invariant Feature Transform

The problem with the Harris corner detector is that it does not consider the possibility of change in scale, which drives a need for another algorithm. Commonly known as SIFT, Scale Invariant Feature Transform is an algorithm developed by David Lowe in 1999, used to detect and describe features within images. Some of the more common applications include object recognition, image stitching, and video tracking. The basis of this algorithm is that we are able

to extract interest points form an object in an image that is able to provide a feature description of the object. These features should then allow us to identify this object in another image even when there are other objects. Some of the driving mechanisms behind the algorithm push for being to identify the object regardless of changes in scale, noise and lighting. To do so, these interest points usually lie in areas of extreme contrast such as would be expected for edges.

The SIFT method starts by first extracting key points from a set of images. These images should all hold some information about the object in question. In other words, for our purposes, all images should contain the objects we are trying to recreating from one angle or another. The extracted points are all stored in a central database. Theoretically, the object in question should be found in another image by comparing each of its features to the database features. This is done by first finding candidate features using the Euclidean distance of their feature vectors. From this set of potential matches, a filters is used by using location, scale, and orientation to reduce the number another subset. After all this and the discard of outliers, object matches can be agreed upon with a pretty high confidence.

There are about six steps to the algorithm each with methods about doing this as well aas their advantages over many other algorithms. The first is about finding interest points regardless of their scale or rotation. In this case the DoG, or Difference of Gaussian scale space function is used. The key factors behind this are that you can shoot for accuracy and stability while scale and rotation do not factor too much into the equation. The second step speaks to the possibility of geometric distortion. At this point we are able to bring up the subject of past discussion of Gaussian blurring. The basis of this is that you can use this to first blur the image then clean it up for resampling. So, a little distortion should not have an effect on the cleaned up samples. The advantage behind this is the ability to avoid overlooking matches due to a little bit of change

in geometry. The next step is key in that we really focus on indexing and matching the key points. A well-known algorithm, nearest neighbor is used. At this point we will also introduce the Best Bin First search algorithm which will be talked about in more detail later in in this paper. These algorithms both play upon their efficiency and speed. Once we've got a good idea of which points match, we can begin to cluster them to points of similar interest. The algorithm used for this is known as Hough Transform voting which we will also go further into detail later. The final steps speak to the ability to clean up the points by getting rid of outliers as well as define our acceptance of the left points. This is done by first imposing the linear least squares method for cleanup and then reducing our success based on Bayesian Probability analysis (Lowe).

CHAPTER III

METHODOLOGY AND FINDINGS

Raspberry Pi Set Up and Configuration

The initial set up of the Raspberry Pi first required that all necessary hardware be bought in order to properly operate the system. In this case the model to be used would be the Raspberry Pi 2 Model B as it is the newest and fastest model. The following are the specs of the model as provided by MCM Electronics which is where the model was specifically ordered from.

- 900 MHz quad-core Arm Cortex-A7 CPU
- 1 GB RAM
- 4 USB Ports
- Full HDMI port
- Ethernet port
- 3.5mm audio jack and composite video
- Camera Interface
- Display Interface
- Micro SD card slot
- VideoCore IV 3D graphics core

Figure 8. *Raspberry Pi Specs.* Raspberry Pi 2. Digital image. MCM Electronics. Web

The basic hardware needed to appropriately operate the Raspberry Pi would be a keyboard, mouse, Ethernet cable, HDMI cable, power supply and a Micro SD card. For the

purpose of my initiative I also ordered a camera module made specifically for this product. It is important to note the price of these items, as the affordability of the overall product is an important factor. Every camera module would require a Raspberry Pi and Micro SD card, so these would really be the required items needed for each camera in use. The Raspberry Pi cost $35, the Micro SD card cost $10 and the camera module cost $25 for a total of $70 for the entire setup.

There are a few important notes to make about the Raspberry Pi which were quickly brought to my attention in my initial attempt to set it up. The first is that there is no pre-loaded software on the device itself therefore it has no operating system. This is where the Micro SD card comes into play. In order to make sure that the OS safely loaded onto the SD card and for there to still be enough space for extra programs the minimum space required would be 8GB. The second is the required power in order to operate the device. A 5V 1A supply should be enough. It should be noted though that due to the high draw of current from multiple devices being connected to the Raspberry Pi, this may not meet the necessities as was the case initially for me. The Raspberry Pi worked best at 5V and 2A.

Once everything was set, it was time to load the OS onto the Micro SD card. In order to do so the card first had to be formatted. The recommended formatting tool was the SD Formatter for Windows as provided at www.sdcard.org. The following screenshot shows the format settings as required.

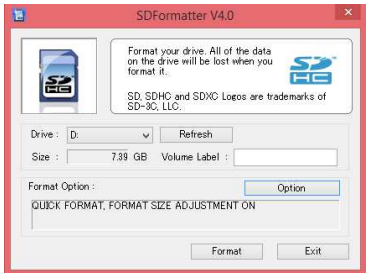


Figure 9. *Format Settings.*

In order to image the SD card, the Raspberry Pi website provides a package that takes care of this installation while also providing optional extra programs including several programming IDE's. The latest version of this is NOOBS 1.4.0 which is what I decided to go with for this device. Once all the files were loaded onto the SD card the device was powered on. It is important to note that the Raspberry Pi has no actual on/off button. It turns on as soon as the power supply is connected. Everything must be connected before the power supply in order to ensure successful booting. There are really only two indicators to look for in order to keep track of the initial status of the boot. The red LED indicates that there is indeed power reaching the device. It must be noted though that this does not necessarily mean that the appropriate amount is being supplied as was the original case with me. The green LED is the main indicator of the processor working as it should. Constant blinking of the LED indicates that the processor is actually reading the SD card. The first time the system is booted we get the following screen which gives us the option for what to install. In this case I decided to install Raspbian which is the OS most recommended and already comes with Python pre-installed. It is important to note that since the Raspberry Pi is connected to its own LCD screen, all images must be acquired externally.

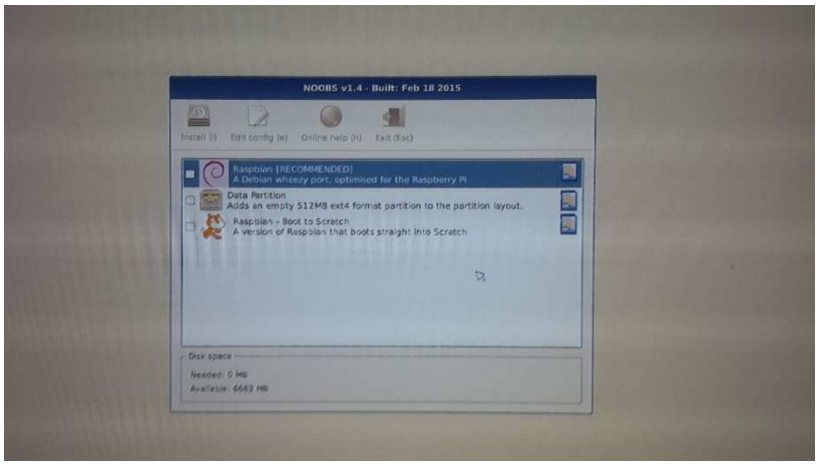


Figure 10. *Raspbian Image Selection*

Once the Raspbian image is selected, the process of installation takes about half an hour. The following shows the screen as it goes through the operation of installation.

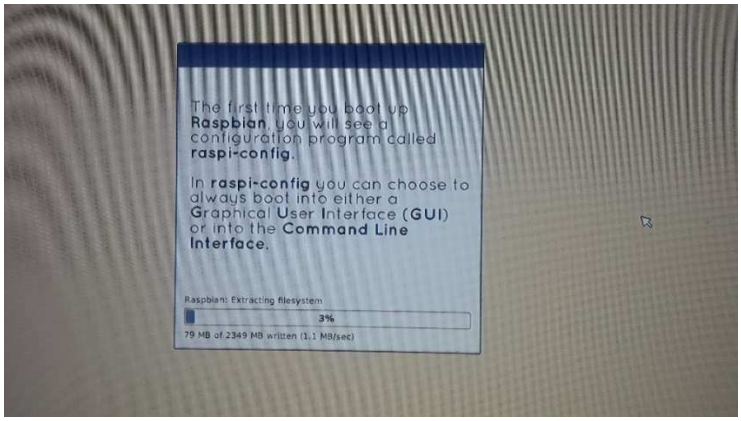


Figure 11. *Raspbian Image Installation*

The first time that the set up was tried, the Raspberry Pi never seemed to boot after the OS installation. After trying everything from switching SD cards to better power supplies, the final conclusion was that the device was defective. Another Raspberry Pi was sent as a replacement and the device worked fine on the first try. The following screen shows the Raspi-Config tool which is the first configuration tool to pop up as soon as the OS installation finishes.

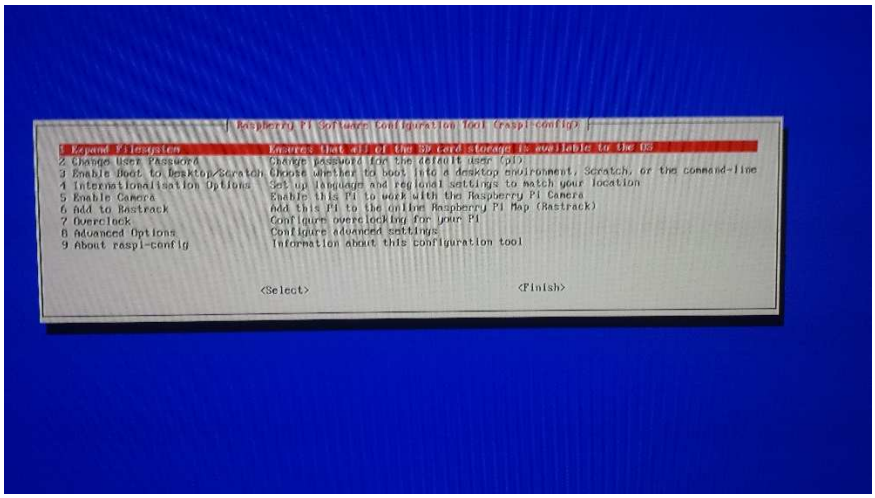


Figure 12. *Raspberry Configuration Tool*

The first configuration would be to automatically log in with the default username and password into the GUI. This should give a screen similar to what we see in Windows.

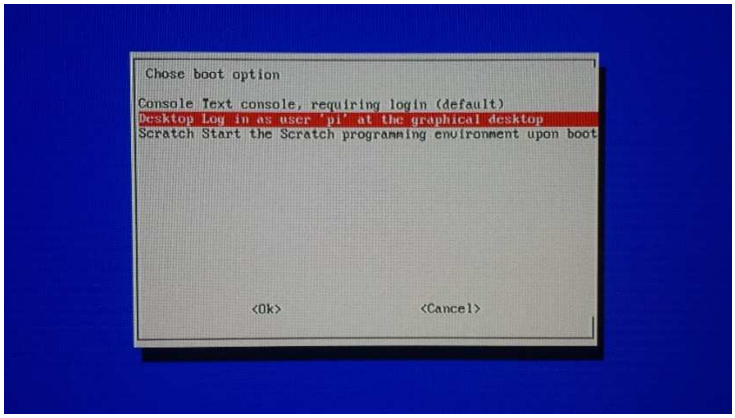


Figure 13. *Login Configuration*

Since I would eventually work with the camera module, it was important to enable camera support for the device. The following screen shows this step of the procedure.

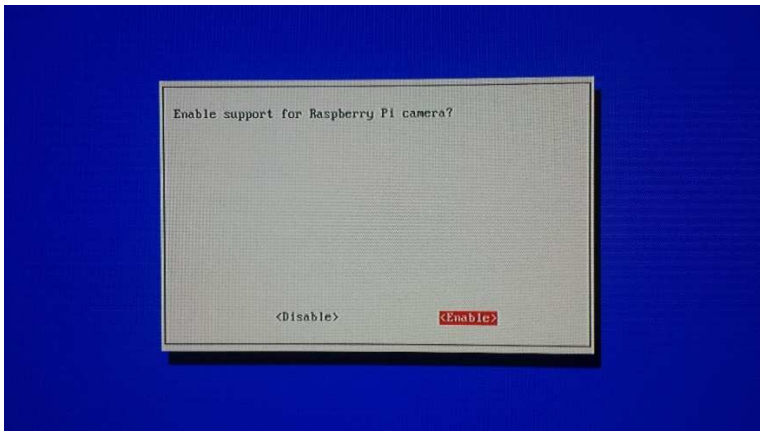


Figure 14. *Camera Enable*

After camera support was enabled, there would be three other items that would also have to be enabled. These will later play a part in being able to remotely connect and control the Raspberry Pi. These items include SSH, SPI and I2C as shown on the following screenshot. These options were found under the advanced configuration settings. The last thing to do was to update the tool to the latest version. This step probably took the longest at about half an hour.

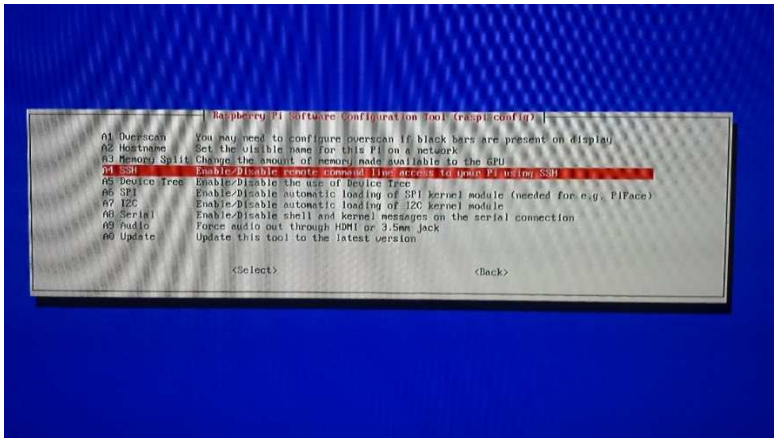


Figure 15. *SSH Enable*

Once all configurations were successfully set up, the following screen shows the GUI that is shown. It is very similar to the desktop GUI of a windows system.

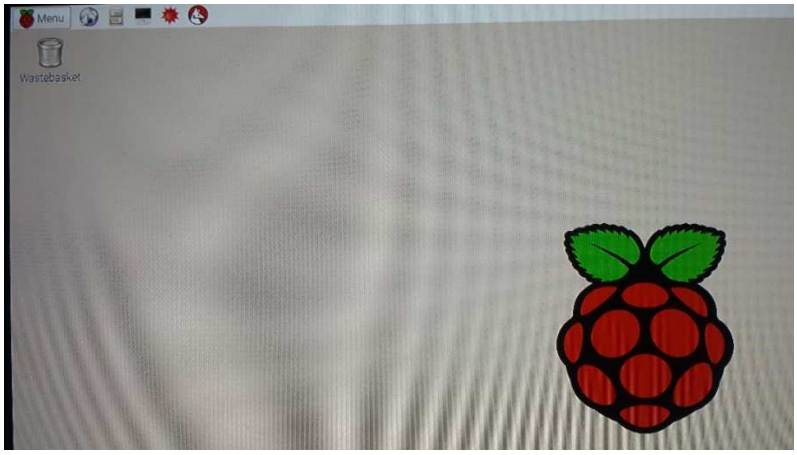


Figure 16. *Raspbian GUI*

Now that the system seemed to be configured and running as it should, the camera module would need to be installed and run to make sure it was properly working. The following shows the camera module. We can also see where the camera module connects to the Raspberry Pi.

Figure 17. *Raspberry Camera Module.* Cam Mod. Digital image. MCM Electronics. Web.

Once the camera was installed, the concentration would now turn to the actual networking of a multiple camera system. In order to do so all previous configuration steps will be applied to all future Raspberry Pi and camera module setups.

Multi-Camera System Construction and Network

In order to show case the intentions of the multi-camera system without spending too much, a minimalistic approach was taken to building system. The drive behind this would be the ability to take the multiple pictures of an object from different angles simultaneously. In order to do so without building a circular system and exhausting the number of needed cameras, a single angle system was designed as the following shows which would in turn take multi-level images while a rotating platform would allow for the variation in angles of the object under inspection. Materials used for the physical construction of the system would include PVC, Velcro and plastic ties in order to allow for quick setup and dismantling.

Other factors that quickly became apparent as playing a major role were lighting as well as background when it came to the topic of photography. In order to best deal with these obstacles, consistent lighting was also integrated into the system as well as a solid background in

order to keep background anomalies from playing any sort of factor in the schema of the setup. The following shows the design of the system from a top view.

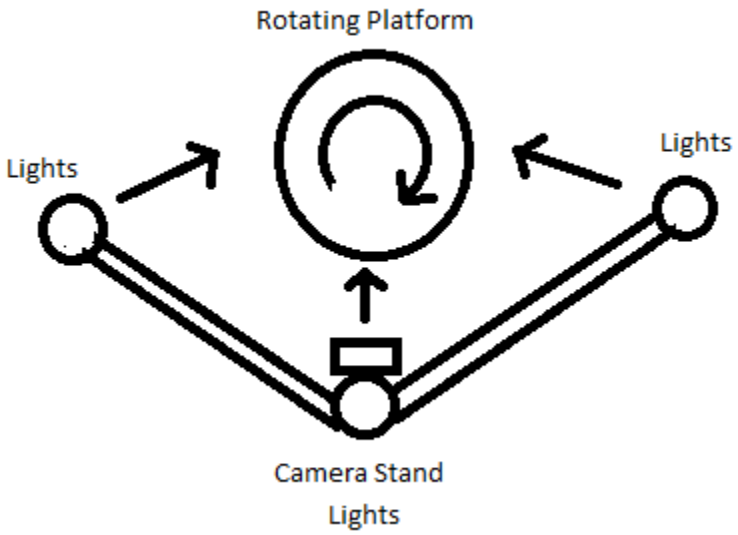


Figure 18. *Top View Structure Design*

The second physical aspect of the structure would entail the physical networking of all the cameras in order to properly receive feedback from them once all scripts were run. In order to allow for further ease other factors would also play a part such as the ability to remotely and wirelessly access the system. It is important to note at this time that although specific components may have been used for the purposes of this project there are no limitations set as it depends solely on the scale of the structure to be built. In order to allow all the cameras to speak to each other a multi-switch board would be used. Although it communication between the PC and the Raspberry Pi's could also have been accessed using the switch, adding a router would allow for remote communication creating for less clutter. Ethernet cables would be used to connect all components. Although it may seem like the most trivial aspect of all, powering the system would actually prove to be an important topic for discussion. Since the Raspberry Pi's do not have a specific specification but rather a minimum voltage and current supply it was

necessary to make sure that while finding ways to distribute power without using multiple power supplies, we did not cut the current distribution too low. For this reason when looking for a power supply that would allow for multiple outputs, the current draw from each output was very important. The Raspberry Pi seemed to work best with a minimum of at least 1A. So a powered hub that would supply 5V while still draw at least 1A was used for these purposes. Again it should be noted that there are multiple methods of doing this depending on the scale of the project. The following figure shows the physical configuration of the network using two Raspberry Pi's as an example.

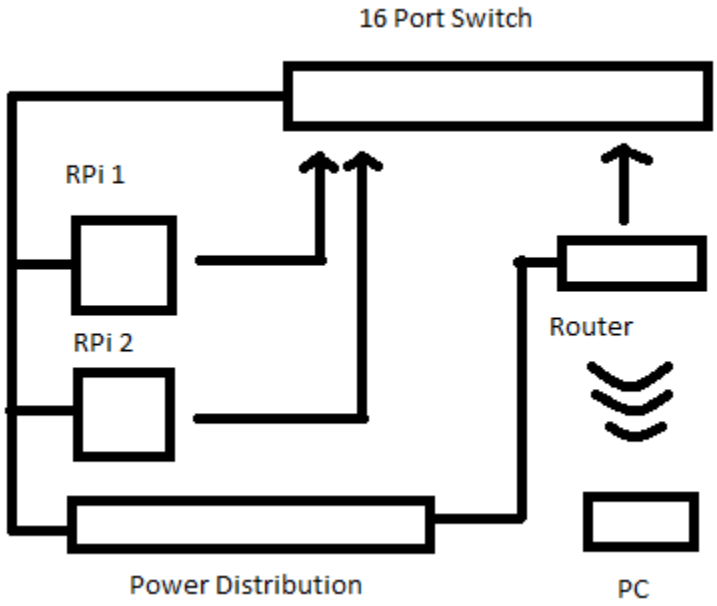


Figure 19. *Physical Network Setup*

Network Configuration

Once the design of both the network and well multi-camera structure was ensured to be accurate the structure was then actually built as shown in the following figure. It is important to note that everything was kept to dismantle as well as be built as quickly and efficiently as possible in order to allow for easy mobilization.

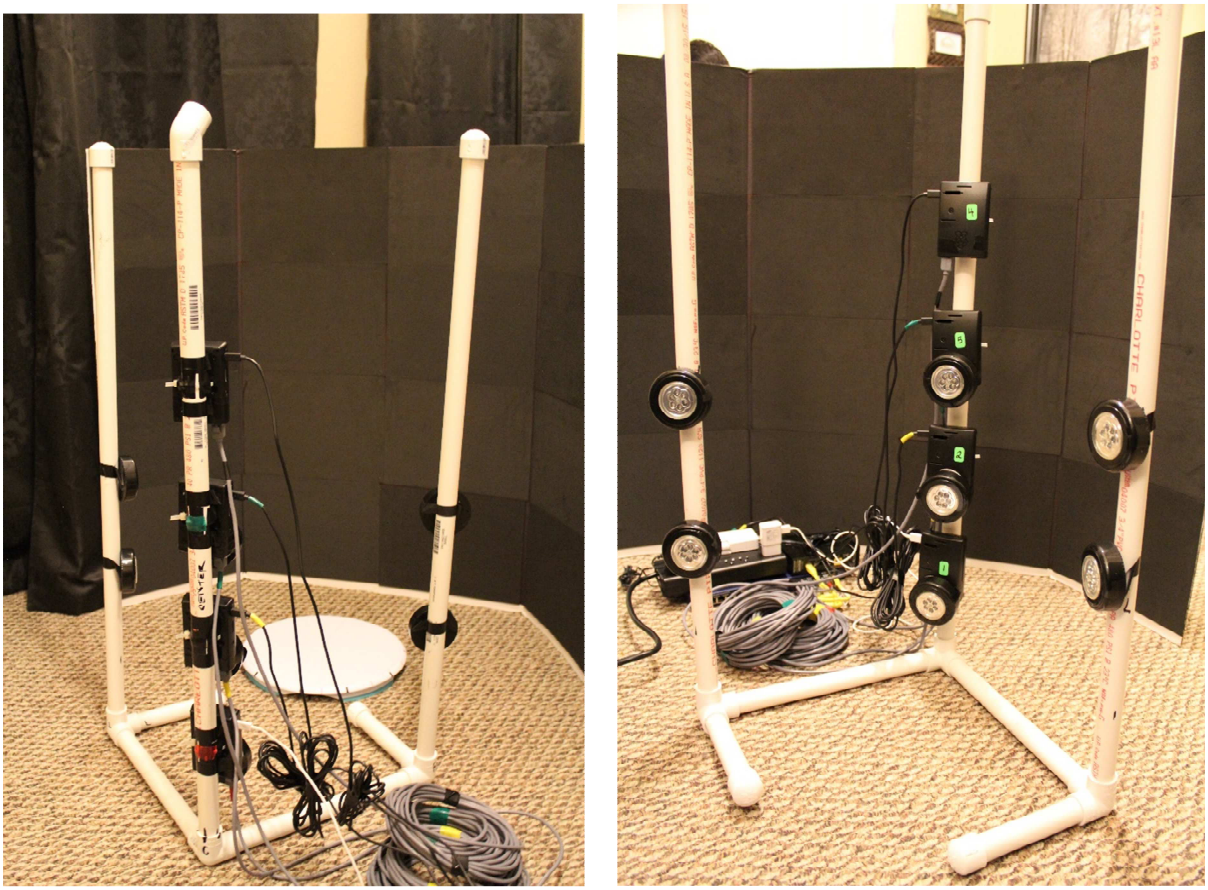


Figure 20. *Multi-Camera System Built*

Once everything had been physically set up to work within the realms of a network, it would become necessary to make sure and allow for the devices to speak to each other by way of scripts as well as commands from the user. Before any of this was done some major considerations had to be taken. Since future modifications and updates would only prove to be very tedious when using multiple Raspberry Pi's, there had to be a way to do so simultaneously. Also, eventually the system should work to take an image from each Raspberry Pi and send it to one central location for further processing. All this pointed to the need for a central file server. There were several options to proceed with the previous but the best proved to be Samba which seemed to be the best option for not only a Linux based system but specifically the Raspberry Pi. Before any of this could be done we had to consider the possibility of IP addresses updating

themselves in the future. This would prove chaotic as this is the main method of addressing individuals Raspberry Pi's. In order to fix this problem a few different steps were undertaken. The first would be to modify the host name of each Raspberry Pi for simplification. The second procedure called for the modification of each Raspberry Pi's network configuration file in order to update the IP address to remain static rather than use DHCP. The following Table 1 shows the settings used for the 4 Raspberry Pi's used for the purposes of this setup.

Host Name	**IP Address**
Pi01	192.168.1.18
Pi02	192.168.1.22
Pi03	192.168.1.10
Pi04	192.168.1.24

Table 2. *Raspberry Pi Static Configurations*

Once the addresses were set to static, it allowed for many other abilities which would simplify the communication process. We would need a tool in order to communicate with the Raspberry Pi's without having to switch monitors each time. The initial enabling of SSH as well as the conversion to a static IP open up this line of communication. The following figure shows a snapshot of Putty, a tool which would be used in order to SSH into the Raspberry Pi from now on.

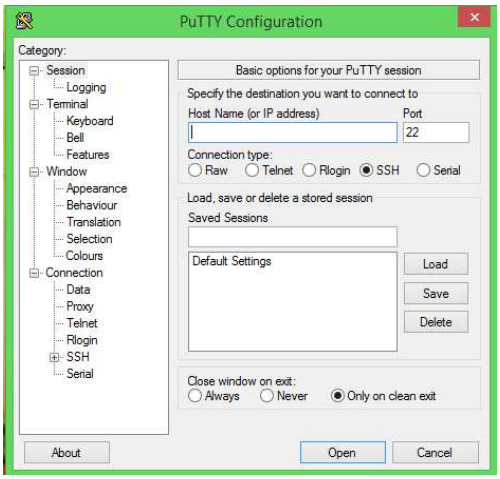


Figure 21. *Putty Configuration*

The next would be to actually make a shareable folder as well as allow for the rest of the Raspberry Pi's to view it. In order to do so, Pi01 was made to be the file server while the rest would be clients of this server. It is important to note that the PC itself would also be a client. So a folder was created on the home directory of Pi01 which would eventually hold all the images. One objective that quickly became apparent was that although a folder may become visible by other clients, the client may not have automatic access to it once the system was restarted. For this reason it became necessary to mount the folder so that it was always available. This worked by writing the mount command onto the boot profile of the clients as well as pointing to the shared folder by using the complete address including the IP of the server. At boot up all Raspberry Pi's would now have access to the shared folder as well as the PC.

Networking Scripts

For our purposes the scripts under discussion may be found in Appendix C. In order to capture simultaneous images, all Raspberry Pi's would have to receive the same message at the same time with as little if possible no lag. Since all had access to the shared folder it became apparent that the folder itself could become the access point for any scripts. In this manner, one script could be updated and affect all systems. With this in mind, in theory the system would

work by using the server to cast out a simultaneous message to all Raspberry Pi's telling them to take an image and save it to the central server. Once each Pi took the image, the central server would wait for indication form each Pi that the image had been taken. At this point a few other factors would come into play. In order to know which Pi's were responding they would each append their hostname to the replied message as well as the images.

There were two scripts that would undertake the previous procedures. First we will discuss the send script as this is tied to the server. The server would first open up a socket in order to allow for communication. Once this was done, input was awaited from the user in order to know what the next step would entail. If it entailed taking the image then the system would take an image while also simultaneously multicasting the message to the other Raspberry Pi's. The server would then listen back to from the other Raspberry Pi's to receive confirmation that the image was taken. If at any point the message was given to exit, then then socket would be closed as well as simultaneously relaying the message to the others in order to also close their sockets.

On the receiving end, a listening script was written. The difference in this case would be that aside from first opening up a socket, it would be bound to an address in order to continue to communicate consistently through the same port. No action would be taken beyond this until it received a message indicating what it should do. In the case of taking an image, it would save the image by appending its hostname as well as replying to the server that it had followed through with its hostname as well. Each time it would continue to listen as well as update the count in order to not overwrite the previous images taken. We now had a manner of taking simultaneous images saved to a central location for processing. The following shows the setup for taking the image of an inanimate object, in this case a cat.

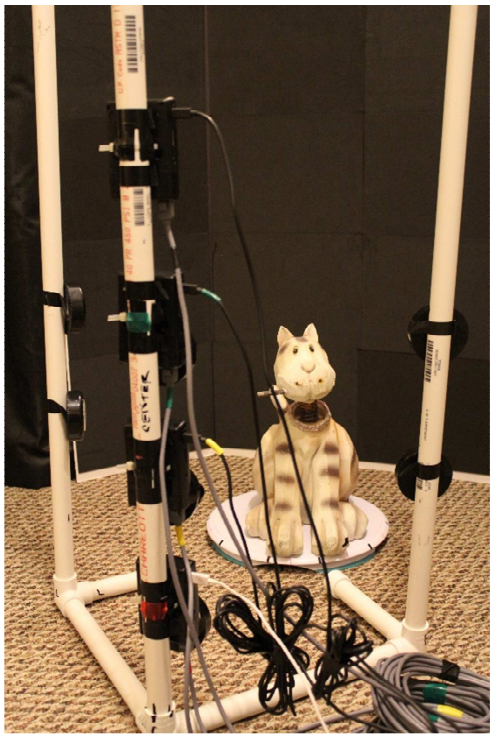


Figure 22. *Experimental Setup*

The following shows the images as taken from different angles of a cat statue. It should be noted that there were a total of 20 angles were taken of the cat in order to increase the number of eventual cloud points. This means that the rotating platform was rotated by 18 degrees in order to capture each image.

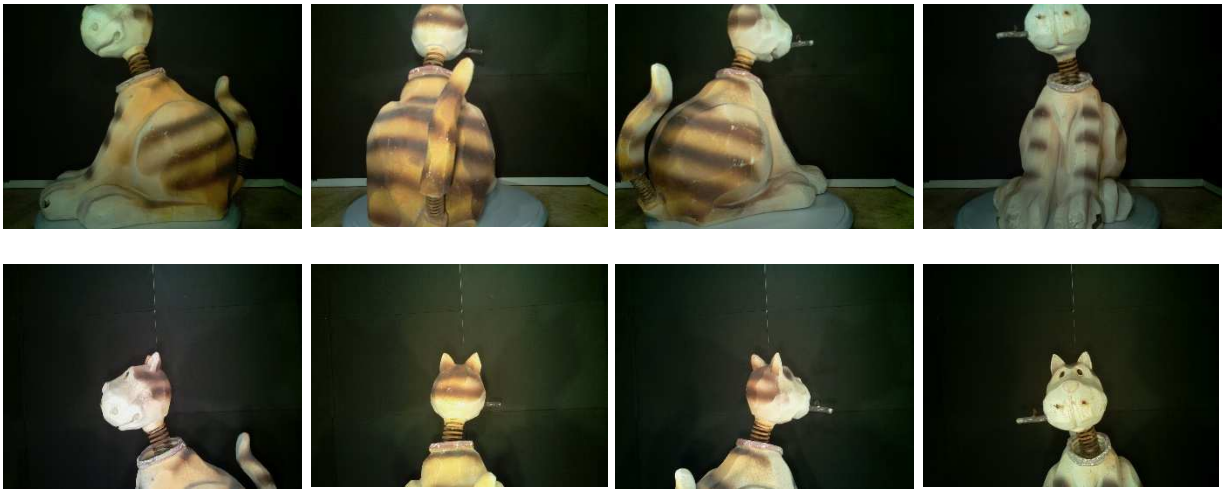


Figure 23. *Sample Raspberry Pi Images*

Python Vision Fundamentals

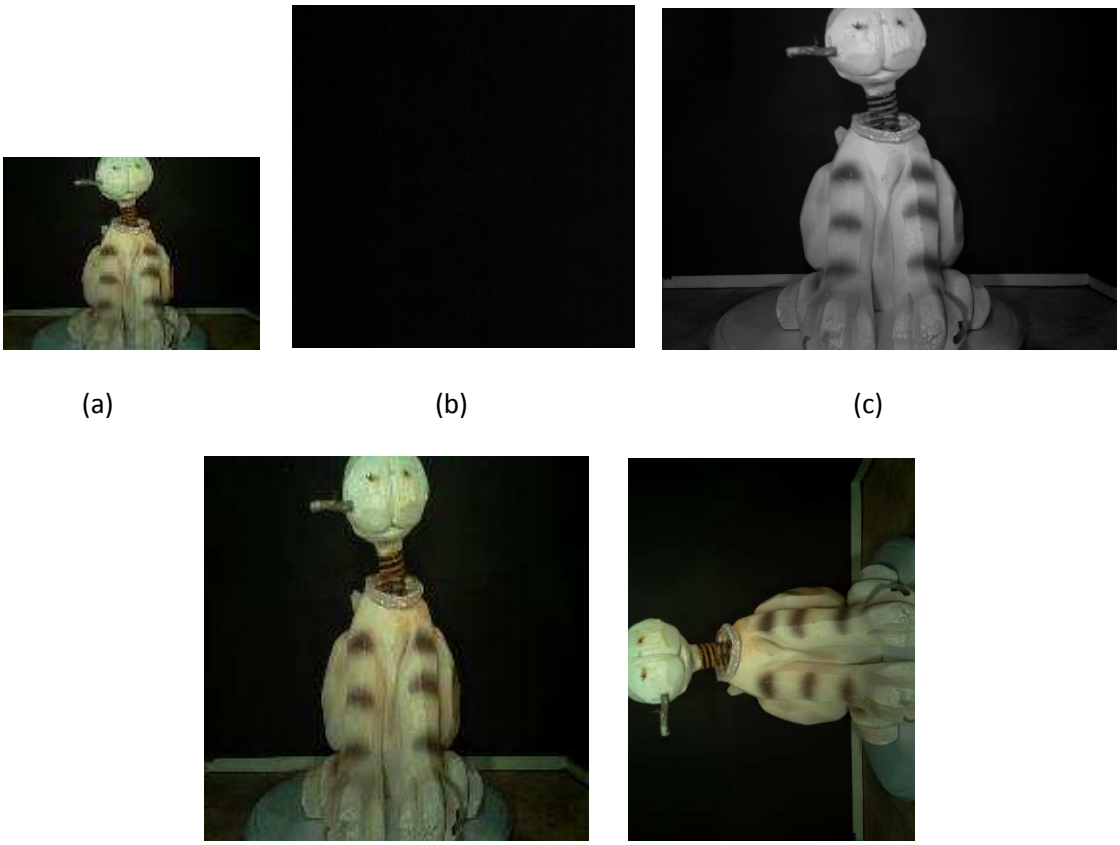
For the purpose of this application, I decided to use Jan Erik Solem's book, "Programming Computer Vision with Python", as a guide to learning and applying some of the fundamental principles which would be built on for the final reconstruction. Only the most important of modules as well as techniques will be discussed.

The image is first imported into a new variable as is possible with the PIL library. The image is further processed by being converted to into greyscale. This will prove to be very important throughout the rest of this process as we are merely concerned for the variation in color intensity rather than the color itself in order to improve feature detection. The following screenshots show the results of before and after the transformation.

Figure 24. *Greyscale Transformation*

Manipulation of the images became very important in not only implementing some of the functionality of the program but in improving efficiency as well. For example, the resizing of an image may reduce the quality of the photograph but will ultimately allow for faster processing as there would be less pixels to deal with. This would prove to be extremely import in future feature detection implementation. Five different operations were run as shown by the following

figure. The image was first resized to thumbnail size as shown by (a). Parts (b) and (c) then show the ability to both crop a portion of the image and paste it in a different such as rotated which was the case for these. Finally, parts (d) and (e) show the ability to both resize by different proportions as well as rotate the image if necessary.



(a) (b) (c)

(d) (e)

Figure 25. *Image Manipulation.* (a) Image thumbnail (b) Image region crop (c) Cropped region image paste (d) Image resize unproportioned (e) Image rotate.

For the remainder of image processing the matplotlib library would become very important as it allowed for the ability to not only convert an image to an array representation but would allow for better understanding of the underlying mechanisms in using those values. The following figure shows an example of the ability to import an image as an array and then use the plotting features to cater to exact points.

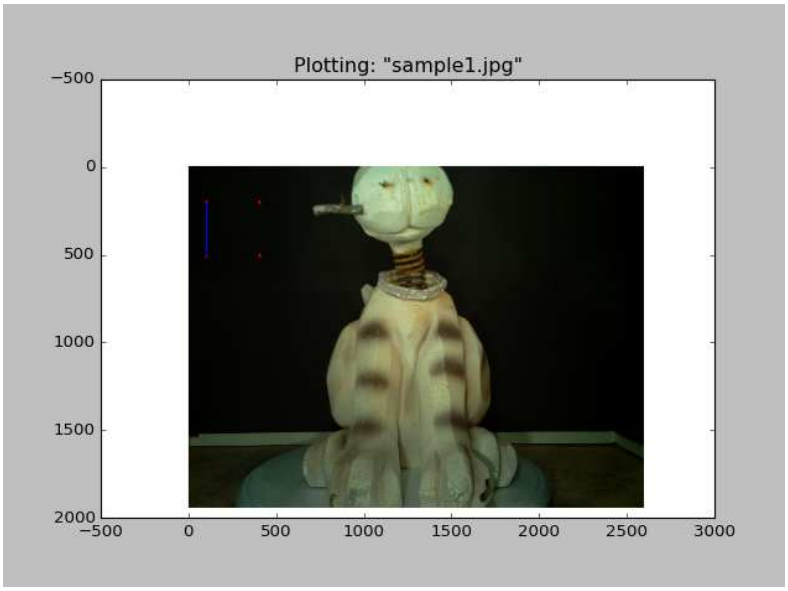


Figure 26. *Image Plot.*

One of the features that immediately popped out as important is the ability to extract contours from an image. This would prove to be critical as point matching from different images will depend on the programs ability to define important variations within an image. The following shows how we may do this by using the pylab module's contour method. One important note to make is the need to first convert the image to grayscale in order to get a better feel for the contour. The following images show the change from the original image to the one who shows the true contour of the image. One important aspect to notice is the definition of the lines around areas that change in color. Already it could be seen where this may come in handy by possibly using the contour option to match exact points on different images, then basically doing a layover of the original image to work with the true color images.

Figure 27. *Image contour.*

Once the image is converted into an array, we can manipulate it running the values through anything from very simple mathematical operations to complex algorithms. The following shows an example of what simple manipulations will do to the image. It is important to note that the already converted grayscale image is being used do many of these operations. If we realize that values of greyscale range from 0-255 we can use this knowledge to make some simple conversions. In the first conversion we see that all the values will be subtracted from 255. This should have the effect of basically inverting all the values to their opposite on the greyscale. The second manipulation of the image clamps the image values to anything from 100-200. This means that the values will be converted to a scaled version of themselves only within this range. The final manipulation allows us to basically apply a quadratic function to the values of the image. This should in turn set all the values to a darker level. The following shows the three resulting images after each of the mathematical manipulations of the original greyscale image array.

(a) (b) (c)

Figure 28. *Mathematical Manipulations.* (a) Values subtracted from 255 (b) Values restricted to interval 100-200 (c) Quadratic function over values

A very useful example of a graylevel transform is histogram equalization. This transform flattens the graylevel histogram of an image so that all intensities are as equally common as possible. This is often a good way to normalize image intensity before further processing and also a way to increase image contrast. The transform function is in this case a cumulative distribution function (cdf) of the pixel values in the image.

It is important to note that on the technical side, at this point a script has been created which will house important functions which may not exist as libraries but will become useful in the further manipulation of images. The script is saved as imtools and is easily used by calling it with the import command.

The function takes a grayscale image and the number of bins to use in the histogram as input and returns an image with equalized histogram together with the cumulative distribution function used to do the mapping of pixel values. It is important to note the use of the last element (index -1) of the cdf to normalize it between 0 and 1.

Figure 29. *Grey Level Transformation.*

To show how the function has actually worked, we are able to print out the function. The way this works is by displaying the pixel values as a function of the range within which they fall within the bin value. So for example in this case since we are displaying the images in grayscale the value will be anything from 0 to 255. The following plots show the before and after of the histogram equalization. It is immediately noticeable that there seems to be a more equal after. Although this may take away from the originality of the image, it may allow for better contrast which will ultimately allow for better point identification in different image views of the same scene.

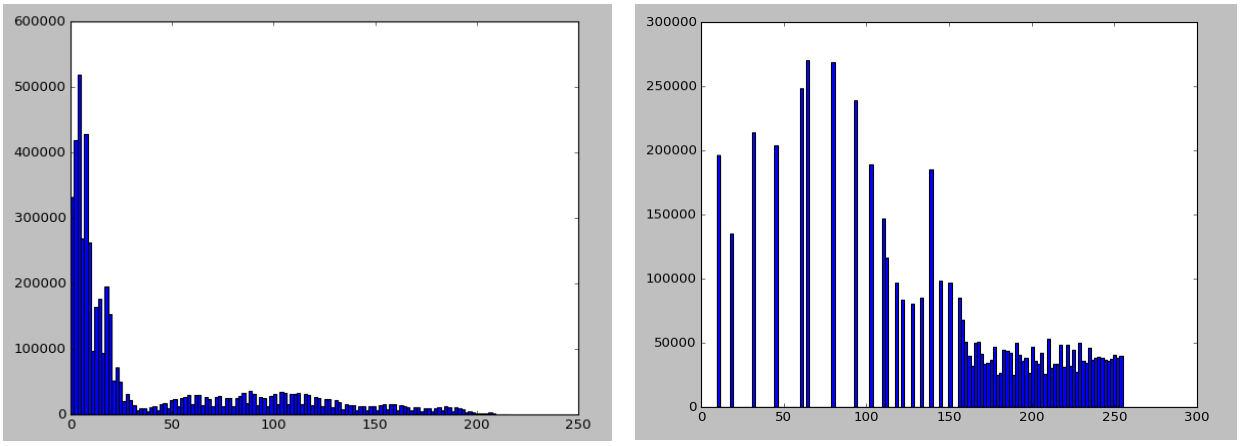


Figure 30. *Histogram Before and After.*

In continuation, other libraries that were built upon the previous yet may be more specific or advanced in their computations were implemented. One of these is SciPy, which is based on NumPy. One ability for example is Gaussian blurring. In essence what is being done is a convolution on the image with a Gaussian filter. The intensity is based on the deviation of the Gaussian effect. The formula used to represent this may be seen as $I_\sigma = I * G\sigma$ where I is the greyscale image and $G\sigma$ is the Gaussian filter with a standard deviation of σ (Solem, 31).

The following images show what happens when an image is convolved with a Gaussian filter of different deviation values, using the following code. Notice the second parameter of the filter is a number. This is the standard deviation. The images show standard deviations of 5, 10 and 15 respectively.

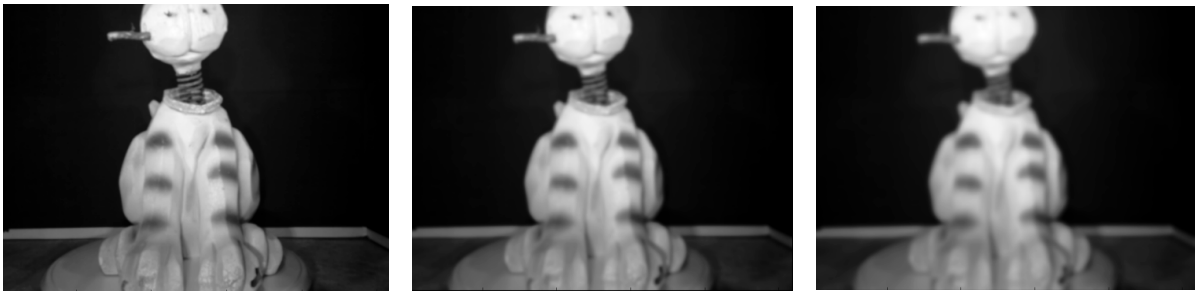


Figure 31. *Gaussian Blurring.*

Although the reason may not have been apparent immediately, Gaussian blurring may provide a quick way to provide points of interest. Let's say an image has too much detail and we only wanted to capture the superficial essence of its surface. We may then use Gaussian blurring to first provide us with the most intense points to create that surface which we may then build off of.

We can use the derivatives to really see how intensity changes within the image. In other words we may be able to better detect edges. We can really see the intensities by using filter to find the x and y derivatives of the image. We can use the following to understand this. If we

think of the gradient of an image it is comprised by the following, $\nabla I = |I_x I_y|^T$, where I_x and I_y are the x and y derivatives respectively. From this there is some important information we can acquire. The gradient magnitude $|\nabla I| = \sqrt{{I_x}^2 + {I_y}^2}$ describes the strength of the intensity, and the gradient angle $\alpha = \arctan(Ix, Iy)$ describes the direction of the intensity at each point or pixel in the image (Solem, 33). The best way to attain the derivatives may be by using filters. The following shows the convolution Sobel filters on the image respectively. We see the original

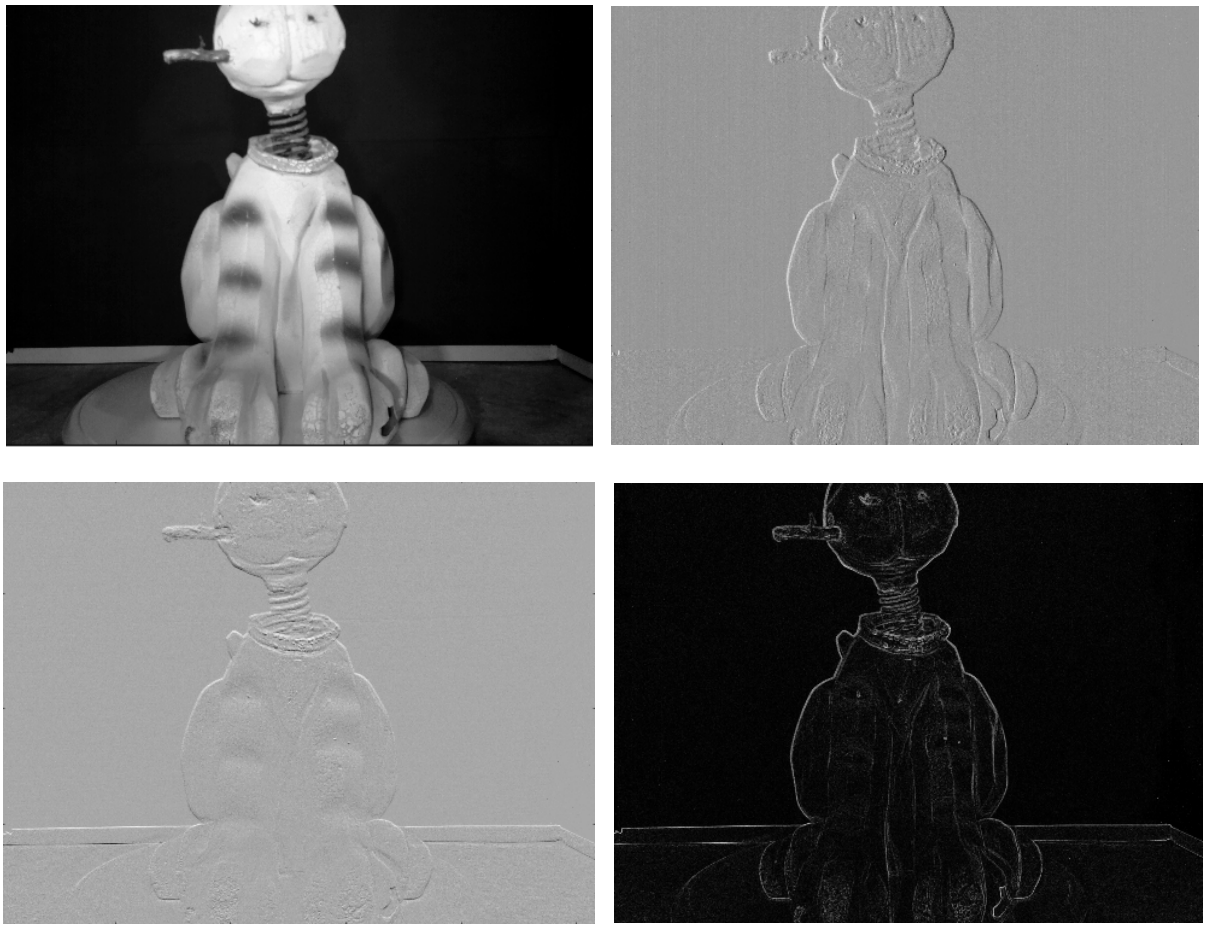


Figure 32. *Image Derivatives.*

At this point previous methods were used to actually find mathcing points between different images. Although the code and implementation will not be elaborated on here, the literature review goes into the theory behind the subject matter. The first of these methods is

known as the Harris corner detection algorithm which really plays on previous startegies such as the gaussian filtering and such. The basics is that it looks for points which have several lines connected to it which would indicate some sort of corner. The following shows the resulting image.

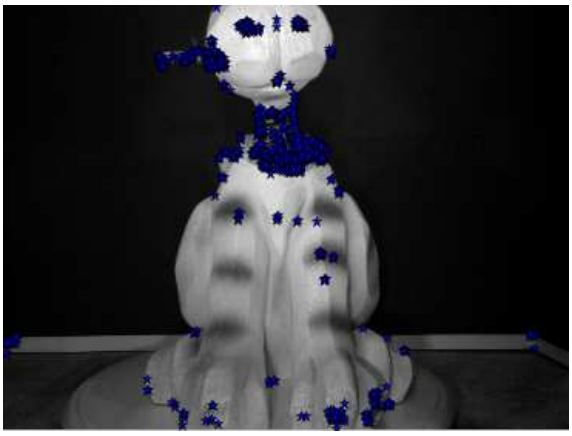


Figure 33. *Harris Corner Detector.*

Although finding points of interest is a great start to 3D reconstrcution, it did not yet give us any information as far as correlation between images. It is at his point that the code was implemented to add descriptors to each of the points. As described in the literature review, descriptors allow for a method of describing a point by way of its surrounding factors. In this method we would then be able to find a good correlation between the two images. The following figure finally shows the impplementation of mathing descriptiors and the output. Lines are plotted to show the match across the two images. It is important to note that there are some errors, which would be delt with in another fashion.

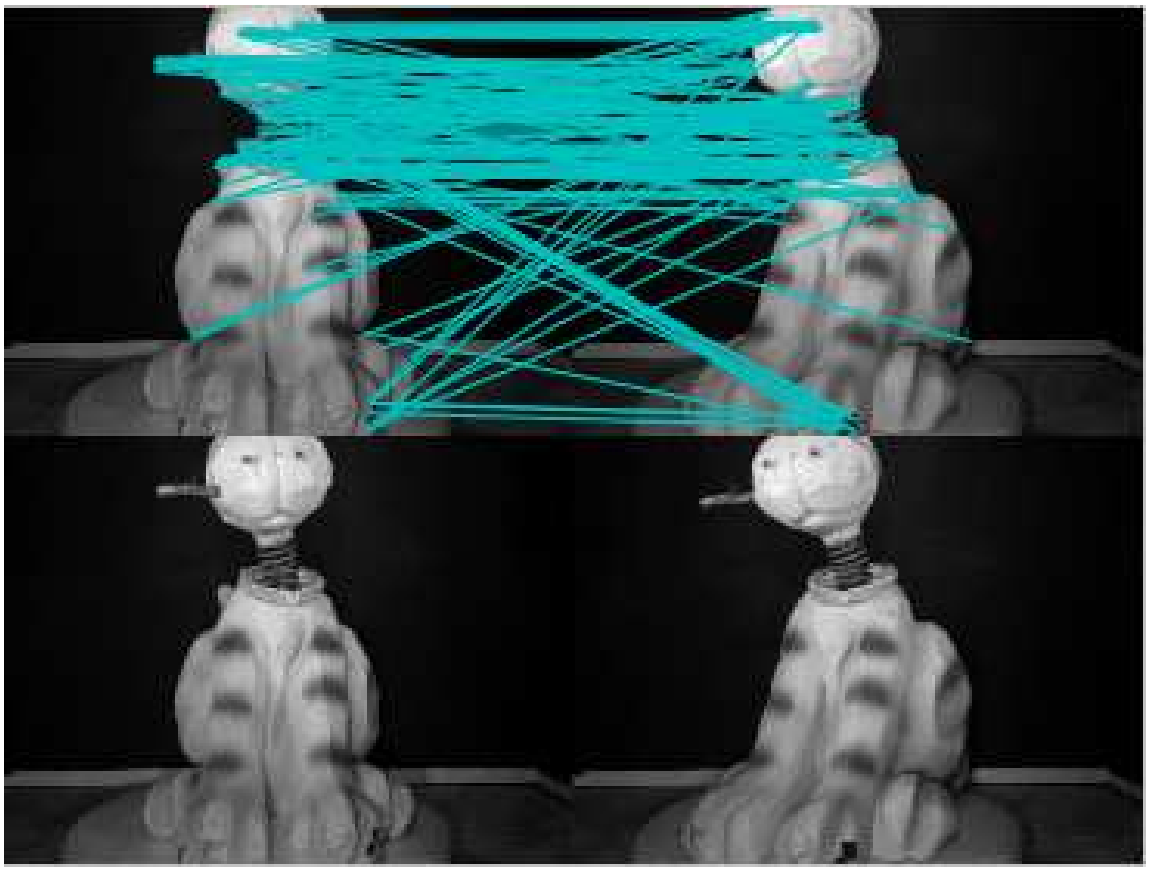


Figure 34. *Feature Matching*

Image Processing

Once all images were acquired, the next step would be to process them. The Python Photogrammetry Toolbox became very apparent as the easiest tool to use in order to extrapolate all the data that would be needed in order to later reconstruct the object form this point cloud. After all the proper configuration and installation procedures the GUI was opened as shown by the following figure. The photographs were run past three basic steps as the theory previously explained. The first would be the calibration of the camera. This would be done by first checking the camera database for the proper camera setting depending on the width of the camera. This was found to be .25 in for the Raspberry Pi camera module. The camera was

found within the parameters. The second step would include running bundler which would call upon the previously discussed algorithms in order to extrapolate common data points across the images. Finally CMVS/PMVS were run to create an even denser cloud from the previously extrapolated points. It is important to note that it may initially be useful to scale down the images so as to cut down on run time and the possibility of a crash due to low resources.

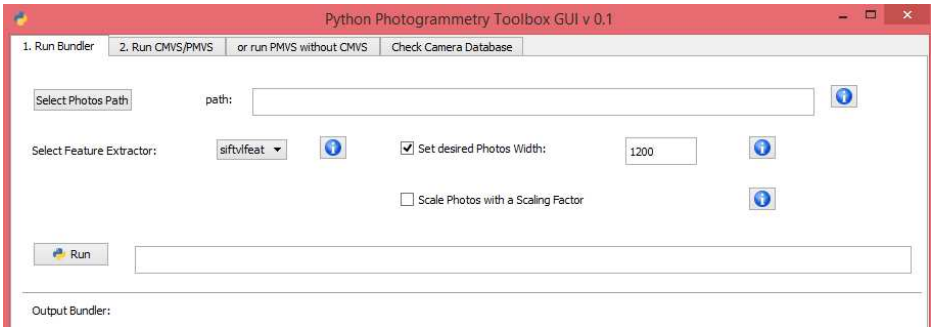


Figure 35. *Python Photogrammetry Toolbox GUI*

The toolbox's ultimate goal is to output a .ply file which can be read by other open source programs in order to then reconstruct and refine the point cloud. At this point MeshLab comes into play as a very user friendly as well as efficient tool. In order to open the point cloud it was as simple as importing the .ply file as produced by the previous toolbox in MeshLab. The following figure shows the point cloud produced as opened without any editing in MeshLab.

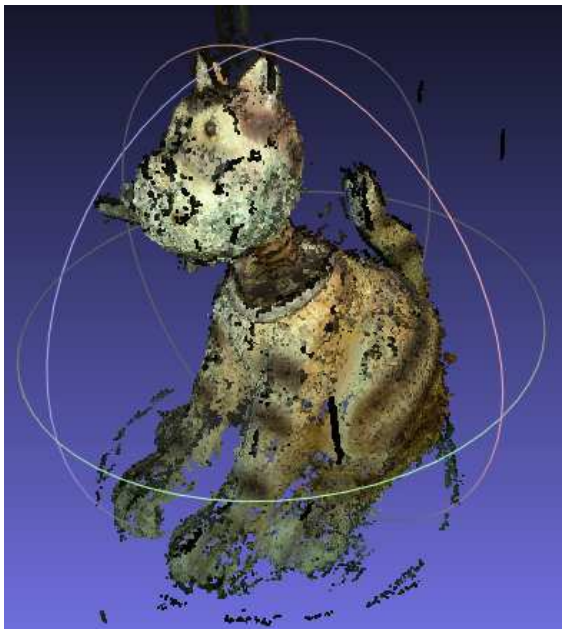


Figure 36. *MeshLab Point Cloud*

As noted, the previous point cloud was a bit rough around the edges but for the most part captures the major details of the object under test including color. At this point MeshLab introduces some techniques to smooth out some of the outlying data as well as close the gaps. Deleting outlying points was as easy as selecting and pressing delete. In order to further smooth the figure a Poisson filter was used. Once this was done the MeshLab could much easily create a mesh closing all the gaps on the object. The following figure shows the resulting object after these actions were taken. From this point the object could easily be exported for further processing or even 3D printing as MeshLab creates acceptable file extensions to do so.

Figure 37. *Poisson Filter Resulting Object*

CHAPTER IV

SUMMARY AND CONCLUSION

A finalization of the process of creating 3D scans of close range objects entailed the culmination of many fields of study with an emphasis on computer science. It is necessary to give credit where due though so that the complete package is understood. To begin, it was necessary to envision how multiple views of an image would be captured. In order to understand this, the field of photogrammetry gave some insight into some of the requirements this would entail. This pointed to the clear fact that each of the views should share common points as from this we would need to extract information in order to pin point their location in 3D space.

The outcome of the previous understanding was a 4-camera rig and a rotating platform in order to allow for consistency when capturing views at different degrees of rotation without changing the object. In order to keep within the realm efficiency, this was built out of Raspberry Pi cameras which would allow for good quality images while still promoting both cost efficiency and ease of use. In order to safely operate the entire networked system, a minor electrical background played a part in ensuring proper power distribution.

Although, the cameras were simple enough to physically setup, networking skills played an integral part to allowing for communication between the cameras as well as the user. The subjects of server/client relationships, socket programming, and multicasting as well as network configuration were all a recurring theme in order to acquire the images.

Finally, the integration of multiple libraries into one central location in order to process the images played a vital part in acquiring the cloud points which would be used for the final reconstruction of the object. Although the ability to write such a program could have been the purpose of its own thesis, understanding of the subject led to the incorporation of several pieces of a puzzle to a centralized program which would use all the previously discussed algorithms in order to extract all the data necessary to create a point cloud and construct a mesh which would ultimately define the remake of the original object.

In summary, as was initially stated, a process was created to scan 3D close range objects by using Raspberry Pi technologies as well as open source software. Specifically, the Raspberry Pi was programmed to work as a multi-camera system which would acquire simultaneous images and send them to a central location for processing. The Bundler, CMVS, PMVS and MeshLab software libraries and packages were then used to both process the images as well as reconstruct the 3D object.

Although the proposed objectives were met, there is still much room for improvement. The processing time was too long at an average of about 30 minutes. Although the intentions of this thesis were not to write a completely new software for 3D reconstruction, future work may look at this as a subject of interest in order to cut down on processing time. The second improvement entails the subject of accuracy. Since the previous algorithms were all based on auto-calibration of the cameras and only projective imaging, the resulting object is not an automatic replica of the original object. This means that although the points in space are proportionally accurate with respect to each other, they may not be accurate with respect to the outside world. Future improvements may entail adding distance parameters to the equation ro even automating this aspect with the use of distance sensors.

Although, I do have areas for improvement I feel that the proposed objectives were met and am glad to have undertaken a difficult yet interesting subject. I can only hope that research in this area continues as I can see the applications of this to be infinite. My dream is that one day we may evolve all current two dimensional platforms to three dimensional to provide a new realm of possibilities.

REFERENCES

Bartoš, Karol, Katarína Pukanská, and Janka Sabová. "Overview of available open-source photogrammetric software, its use and analysis."*International Journal for Innovation Education and Research* 2.4 (2014): 62-70.

Cignoni, Paolo, et al. "MeshLab: an Open-Source Mesh Processing Tool."*Eurographics Italian Chapter Conference*. Vol. 2008. 2008.

Flavell, Lance. *Beginning Blender: Open Source 3D Modeling, Animation, and Game Design*. Apress, 2010.

Solem, Jan Erik. *Programming Computer Vision with Python: Tools and algorithms for analyzing images*. " O'Reilly Media, Inc.", 2012.

Lindeberg, Tony. "Scale invariant feature transform." *Scholarpedia* 7.5 (2012): 10491.

Lowe, David G. "Object recognition from local scale-invariant features." *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee, 1999.

Lowe, David G. "Distinctive image features from scale-invariant keypoints."*International journal of computer vision* 60.2 (2004): 91-110.

Hartley, Richard, and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

Treleaven, Philip, and Jonathan Wells. "3D body scanning and healthcare applications." *Computer* 7 (2007): 28-34.

Guarnieri, Alberto, Francesco Pirotti, and Antonio Vettore. "Cultural heritage interactive 3D models on the web: An approach using open source and free software." *Journal of Cultural Heritage* 11.3 (2010): 350-353.

Falkingham, Peter L. "Acquisition of high resolution three-dimensional models using free, open-source, photogrammetric software." *Palaeontologia Electronica*15.1 (2012): 15.

Jazayeri, I., C. S. Fraser, and S. Cronk. "Automated 3D object reconstruction via multi-image close-range photogrammetry." *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci* 38 (2010): 305-310.

De La Torre García, Javier. "A platform for automatic 3D object reconstruction through multi-view stereo techniques for mobile devices." (2013).

Wulf, Oliver, and Bernardo Wagner. "Fast 3D scanning methods for laser measurement systems." *International conference on control systems and computer science (CSCS14)*. 2003.

Boehler, Wolfgang, and Andreas Marbs. "3D scanning instruments." *Proceedings of the CIPA WG 6 International Workshop on Scanning for Cultural Heritage Recording, Ziti, Thessaloniki*. 2002.

Bartoš, Karol, Katarína Pukanská, and Janka Sabová. "Overview of available open-source photogrammetric software, its use and analysis." *International Journal for Innovation Education and Research* 2.4 (2014): 62-70.

Beraldin, Jean-Angelo, et al. "Virtualizing a Byzantine crypt by combining high-resolution textures with laser scanner 3D data." (2002).

Seitz, Steven M., et al. "A comparison and evaluation of multi-view stereo reconstruction algorithms." *Computer vision and pattern recognition, 2006 IEEE Computer Society Conference on*. Vol. 1. IEEE, 2006.

Thorne, Brian. "Introduction to Computer Vision in Python." *University of Canterbury. New Zealand* (2009).

Baltsavias, Emmanuel P. "A comparison between photogrammetry and laser scanning." *ISPRS Journal of photogrammetry and Remote Sensing* 54.2 (1999): 83-94.

Moulon, Pierre, and Alessandro Bezzi. "Python Photogrammetry Toolbox: A free solution for Three-Dimensional Documentation." *ArcheoFoss*. 2011.

Brooks, Michael J., et al. "Towards robust metric reconstruction via a dynamic uncalibrated stereo head." *Image and Vision Computing* 16.14 (1998): 989-1002.

Hartley, Richard. "In defense of the eight-point algorithm." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19.6 (1997): 580-593.

Derpanis, Konstantinos G. "The harris corner detector." *York University* (2004).

Snavely, Noah, Steven M. Seitz, and Richard Szeliski. "Photo tourism: exploring photo collections in 3D." *ACM transactions on graphics (TOG)*. Vol. 25. No. 3. ACM, 2006.

Furukawa, Yasutaka, and Jean Ponce. "Accurate, dense, and robust multiview stereopsis." *Pattern Analysis and Machine Intelligence, IEEE Transactions on*32.8 (2010): 1362-1376.

Furukawa, Yasutaka. "Clustering views for multi-view stereo (cmvs)." *Website-http://grail. cs. washington. edu/software/cmvs* 4 (2012).

Furukawa, Yasutaka, and Jean Ponce. "Patch-based Multi-view Stereo Software (PMVS-Version 2)." *PMVS2, University of Washington, Department of Computer Science and Engineering. Web. Downloaded from on May* 14 (2012).

Snavely, Noah. "Bundler: Structure from motion (SFM) for unordered image collections." *Available online: phototour. cs. washington. edu/bundler/(accessed on 12 July 2013)* (2010).

APPENDIX A

APPENDIX A

OPEN SOURCE LIBRARIES AND SOFTWARE

Python 2.7.10 for Windows 32-Bit
Source: https://www.python.org/download/releases/2.7/

Python Imaging Library (PIL 1.1.7) for Python 2.7
Source: http://www.pythonware.com/products/pil/

Matplotlib for Python 2.7
Source: http://matplotlib.sourceforge.net/

NumPy for Python 2.7
Source: http://www.numpy.org/

SciPy for Python 2.7
Source: http://www.scipy.org

OpenCV Python for Windows
Source: http://sourceforge.net

PyQT for Python 2.7
Source: http://sourceforge.net

Bundler:
Source: http://www.cs.cornell.edu/~snavely/bundler/

PMVS:
Source: http://www.di.ens.fr/pmvs/

CMVS:
Source: http://www.di.ens.fr/cmvs/

Pyhton Photogrammetry Toolbox
Source: http://184.106.205.13/arcteam/ppt.php

APPENDIX B

APPENDIX B

PYTHON FUNDAMENTALS SOURCE CODE

Image.py:

```
from PIL import Image
from pylab import *
from numpy import *
import imtools
from imtools import *
import harris
from scipy.ndimage import filters
import cv2
import numpy as np
import os

"""Start with one image file named sample1.jpg to test script"""

"""Opens an image, converts it then writes it out to a different image"""
"""pil_im=Image.open('sample1.jpg').convert('L')
pil_im.save('sample2.jpg')"""

"""Will create a thumbnail from an image"""
"""pil_im2=Image.open('sample1.jpg')
pil_im2.thumbnail((128,128))
pil_im2.save('sample3.jpg')"""

"""Will crop a certain region"""
"""box=(0,0,400,400)
region=pil_im.crop(box)
region.save('sample4.jpg')"""

"""Can rotate and re-paste using the following"""
"""region = region.transpose(Image.ROTATE_180)
pil_im.paste(region,box)
pil_im.save('sample5.jpg')"""

"""To resize an image"""
"""pil_resize=Image.open('sample1.jpg')
out=pil_resize.resize((200,200))
out.save('sample6.jpg')"""
```

```
"""To rotate an image"""
"""pil_rot=Image.open('sample1.jpg')
out=pil_rot.rotate(90)
out.save('sample7.jpg')"""

"""The following creates an array from an image then plots along with markers and a line
it is important to note the array is of the form im[m,n]"""
"""im = array(Image.open('sample1.jpg'))
imshow(im)                #Must be an mxn array
x = [100,100,400,400]
y = [200,500,200,500]
plot(x,y,'r*')
plot(x[0:2],y[0:2])
title('Plotting: "sample1.jpg"')
show()"""

"""The following creates an image contour"""
"""im = array(Image.open('sample1.jpg').convert('L'))
figure()
gray()
axis('equal')             #This allows for the axis to both
contour(im, origin='image')   # Secondary arguments relay to strting location and other features
axis('off')
show()"""

"""The following will create a histogram for the image"""
"""figure()
hist(im.flatten(),128)
show()"""


"""The following will display information about the array in the form of a tuple, with the
form (rows, columns, color channels).  The following string shows the datatype"""
"""im = array(Image.open('sample1.jpg'))
print im.shape, im.dtype
im = array(Image.open('sample1.jpg').convert('L'),'f')
print im.shape, im.dtype"""

"""The following form allows us to access a single value from the array.  It is important
to notice in the following we get back a single value  since it is in grayscale """
"""value=im[1,1]
print value"""

"""The following enables three different operations on the array of the image
creating different effects which may become useful"""
"""im = array(Image.open('sample1.jpg').convert('L'))
```

```
figure()
im2 = 255 - im #invert image
imshow(im2,cmap=cm.Greys_r)
show()
figure()
im3 = (100.0/255) * im + 100 #clamp to interval 100...200
imshow(im3,cmap=cm.Greys_r)
show()
figure()
im4 = 255.0 * (im/255.0)**2 #squared
imshow(im4,cmap=cm.Greys_r)
show()    """

"""The following will call on the histogram equalization tool and show its resulting
image and new histogram"""
"""im = array(Image.open('sample1.jpg').convert('L'))
figure()
imshow(im,cmap=cm.Greys_r)
figure()
hist(im.flatten(),128)
im2,cdf=imtools.histeq(im)
figure()
imshow(im2,cmap=cm.Greys_r)
figure()
hist(im2.flatten(),128)
show()"""

"""The following will perform a Gaussian blur on an image"""
"""im = array(Image.open('sample1.jpg').convert('L'))
im2 = filters.gaussian_filter(im,15)
imshow(im2,cmap=cm.Greys_r)
show()"""

"""The following will compute the convolution to derive the image derivatives"""
"""im = array(Image.open('sample1.jpg').convert('L'))
figure()
imshow(im,cmap=cm.Greys_r)

imx = zeros(im.shape)
filters.prewitt(im,1,imx)
figure()
imshow(imx,cmap=cm.Greys_r)

imy = zeros(im.shape)
filters.prewitt(im,0,imy)
figure()
```

```
imshow(imy,cmap=cm.Greys_r)

magnitude = sqrt(imx**2+imy**2)
figure()
imshow(magnitude,cmap=cm.Greys_r)
show()"""

"""This will use the Harris corner detection algorithm"""
"""im = array(Image.open('sample1.jpg').convert('L'))
harrisim = harris.compute_harris_response(im)
filtered_coords = harris.get_harris_points(harrisim,6)
harris.plot_harris_points(im, filtered_coords)"""

#This will compare two images using harris point descriptors
"""wid=5
im1 = array(Image.open('sample1.jpg').convert('L'))
im2 = array(Image.open('sample1turn.jpg').convert('L'))

# resize to make matching faster
im1 = imresize(im1,(im1.shape[1]/2,im1.shape[0]/2))
im2 = imresize(im2,(im2.shape[1]/2,im2.shape[0]/2))

harrisim = harris.compute_harris_response(im1,5)
filtered_coords1 = harris.get_harris_points(harrisim,wid+1)
d1 = harris.get_descriptors(im1,filtered_coords1,wid)


harrisim = harris.compute_harris_response(im2,5)
filtered_coords2 = harris.get_harris_points(harrisim,wid+1)
d2 = harris.get_descriptors(im2,filtered_coords2,wid)

print 'starting matching'
matches = harris.match_twosided(d1,d2)

figure()
gray()


harris.plot_matches(im1,im2,filtered_coords1,filtered_coords2,matches)
show()"""
```

Imtools.py:

```
from PIL import Image
from pylab import *
```

```
from numpy import *
import imtools
from scipy.ndimage import filters
import os

"""Will print out all the file names of the images in a given path"""
def get_imlist(path):
    return [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]
    print get_imlist('C:\Users\juan\Desktop\Photogrammetry\src')

""" Resize an image array using PIL. """
def imresize(im,sz):
    pil_im = Image.fromarray(uint8(im))
    return array(pil_im.resize(sz))

""" Histogram equalization of a grayscale image. """
def histeq(im,nbr_bins=256):
    # get image histogram
    imhist,bins = histogram(im.flatten(),nbr_bins,normed=True)
    cdf = imhist.cumsum() # cumulative distribution function
    cdf = 255 * cdf / cdf[-1] # normalize
    # use linear interpolation of cdf to find new pixel values
    im2 = interp(im.flatten(),bins[:-1],cdf)
    return im2.reshape(im.shape), cdf
def compute_average(imlist):
    """ Compute the average of a list of images. """
    # open first image and make into array of type float
    averageim = array(Image.open(imlist[0]), 'f')
    for imname in imlist[1:]:
        try:
            averageim += array(Image.open(imname))
        except:
            print imname + '...skipped'
    averageim /= len(imlist)
    # return average as uint8
    return array(averageim, 'uint8')

def pca(X):
    """ Principal Component Analysis
    input: X, matrix with training data stored as flattened arrays in rows
    return: projection matrix (with important dimensions first), variance and mean.
    """
    # get dimensions
    num_data,dim = X.shape
    # center data
    mean_X = X.mean(axis=0)
```

```
    X = X - mean_X
    if dim>num_data:
        # PCA - compact trick used
        M = dot(X,X.T) # covariance matrix
        e,EV = linalg.eigh(M) # eigenvalues and eigenvectors
        tmp = dot(X.T,EV).T # this is the compact trick
        V = tmp[::-1] # reverse since last eigenvectors are the ones we want
        S = sqrt(e)[::-1] # reverse since eigenvalues are in increasing order
        for i in range(V.shape[1]):
            V[:,i] /= S
    else:
        # PCA - SVD used
        U,S,V = linalg.svd(X)
        V = V[:num_data] # only makes sense to return the first num_data
    # return the projection matrix, the variance and the mean
    return V,S,mean_X
```

Harris.py:

```
from pylab import *
from numpy import *
from scipy.ndimage import filters


def compute_harris_response(im,sigma=3):
    """ Compute the Harris corner detector response function
        for each pixel in a graylevel image. """

    # derivatives
    imx = zeros(im.shape)
    filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)
    imy = zeros(im.shape)
    filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)

    # compute components of the Harris matrix
    Wxx = filters.gaussian_filter(imx*imx,sigma)
    Wxy = filters.gaussian_filter(imx*imy,sigma)
    Wyy = filters.gaussian_filter(imy*imy,sigma)

    # determinant and trace
    Wdet = Wxx*Wyy - Wxy**2
    Wtr = Wxx + Wyy

    return Wdet / Wtr
```

```
    def get_harris_points(harrisim,min_dist=10,threshold=0.1):
    """ Return corners from a Harris response image
        min_dist is the minimum number of pixels separating
        corners and image boundary. """

    # find top corner candidates above a threshold
    corner_threshold = harrisim.max() * threshold
    harrisim_t = (harrisim > corner_threshold) * 1

    # get coordinates of candidates
    coords = array(harrisim_t.nonzero()).T

    # ...and their values
    candidate_values = [harrisim[c[0],c[1]] for c in coords]

    # sort candidates (reverse to get descending order)
    index = argsort(candidate_values)[::-1]

    # store allowed point locations in array
    allowed_locations = zeros(harrisim.shape)
    allowed_locations[min_dist:-min_dist,min_dist:-min_dist] = 1

    # select the best points taking min_distance into account
    filtered_coords = []
    for i in index:
        if allowed_locations[coords[i,0],coords[i,1]] == 1:
            filtered_coords.append(coords[i])
            allowed_locations[(coords[i,0]-min_dist):(coords[i,0]+min_dist),
                    (coords[i,1]-min_dist):(coords[i,1]+min_dist)] = 0

    return filtered_coords


def plot_harris_points(image,filtered_coords):
    """ Plots corners found in image. """

    figure()
    gray()
    imshow(image)
    plot([p[1] for p in filtered_coords],
            [p[0] for p in filtered_coords],'*')
    axis('off')
    show()


def get_descriptors(image,filtered_coords,wid=5):
```

```
    """ For each point return pixel values around the point
        using a neighbourhood of width 2*wid+1. (Assume points are
        extracted with min_distance > wid). """

    desc = []
    for coords in filtered_coords:
        patch = image[coords[0]-wid:coords[0]+wid+1,
                      coords[1]-wid:coords[1]+wid+1].flatten()
        desc.append(patch)

    return desc


def match(desc1,desc2,threshold=0.5):
    """ For each corner point descriptor in the first image,
        select its match to second image using
        normalized cross correlation. """

    n = len(desc1[0])

    # pair-wise distances
    d = -ones((len(desc1),len(desc2)))
    for i in range(len(desc1)):
        for j in range(len(desc2)):
            d1 = (desc1[i] - mean(desc1[i])) / std(desc1[i])
            d2 = (desc2[j] - mean(desc2[j])) / std(desc2[j])
            ncc_value = sum(d1 * d2) / (n-1)
            if ncc_value > threshold:
                d[i,j] = ncc_value

    ndx = argsort(-d)
    matchscores = ndx[:,0]

    return matchscores


def match_twosided(desc1,desc2,threshold=0.5):
    """ Two-sided symmetric version of match(). """

    matches_12 = match(desc1,desc2,threshold)
    matches_21 = match(desc2,desc1,threshold)

    ndx_12 = where(matches_12 >= 0)[0]

    # remove matches that are not symmetric
    for n in ndx_12:
```

```
        if matches_21[matches_12[n]] != n:
            matches_12[n] = -1

    return matches_12


def appendimages(im1,im2):
    """ Return a new image that appends the two images side-by-side. """

    # select the image with the fewest rows and fill in enough empty rows
    rows1 = im1.shape[0]
    rows2 = im2.shape[0]

    if rows1 < rows2:
        im1 = concatenate((im1,zeros((rows2-rows1,im1.shape[1]))),axis=0)
    elif rows1 > rows2:
        im2 = concatenate((im2,zeros((rows1-rows2,im2.shape[1]))),axis=0)
    # if none of these cases they are equal, no filling needed.

    return concatenate((im1,im2), axis=1)


def plot_matches(im1,im2,locs1,locs2,matchscores,show_below=True):
    """ Show a figure with lines joining the accepted matches
        input: im1,im2 (images as arrays), locs1,locs2 (feature locations),
        matchscores (as output from 'match()'),
        show_below (if images should be shown below matches). """

    im3 = appendimages(im1,im2)
    if show_below:
        im3 = vstack((im3,im3))

    imshow(im3)

    cols1 = im1.shape[1]
    for i,m in enumerate(matchscores):
        if m>0:
            plot([locs1[i][1],locs2[m][1]+cols1],[locs1[i][0],locs2[m][0]],'c')
    axis('of
```

APPENDIX C

APPENDIX C

NETWORKING RASPBERRY PI SCRIPTS

Send.py:

```
multicast_group = ('224.3.29.71', 10000)
name=socket.gethostname()

# Create the datagram socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Set the time-to-live for messages to 1 so they do not go past the
# local network segment.
ttl = struct.pack('b', 1)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)
counter=1
while True:
   count=0
   message = raw_input('Press Enter to take picture or type Exit to quit: ')
   if( message=='Exit'):
      sent=sock.sendto(message, multicast_group)
      break

   # Send data to the multicast group
   print 'Sending...'
   sent = sock.sendto(message, multicast_group)

   # Server will take picture as well
   os.system('raspistill -n -q 100 -o '+name+'image'+str(counter)+'.jpg')
   counter+=1
   print name+' Complete!'

   # Look for responses from all recipients
   print 'Waiting to receive confirmation'
   while True:
      data, server = sock.recvfrom(16)
      print >>sys.stderr, 'received "%s"' %(data)
      count+=1
      if (count!=0 and count%2==0):
            break
```

```
print >>sys.stderr, 'closing socket'
sock.close()
```

Listen.py:

```
import socket
import struct
import sys
import os
import subprocess

multicast_group = '224.3.29.71'
server_address = ('', 10000)
name=socket.gethostname()
# Create the socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to the server address
sock.bind(server_address)

# Tell the operating system to add the socket to the multicast group
# on all interfaces.
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
counter=1
# Receive/respond loop
while True:
    print >>sys.stderr, '\nWaiting to receive message'
    data, address = sock.recvfrom(1024)
    if(data=='Exit'):
            break

    print >>sys.stderr, 'Taking picture...'
    print >>sys.stderr, data
    os.system('raspistill  -n -q 100 -o '+name+'image'+str(counter)+'.jpg')
    print >>sys.stderr, 'Done. Sending acknowledgement...'
    sock.sendto(name+' Done!', address)
    counter+=1

sock.close()
```

BIOGRAPHICAL SKETCH

Juan Lorenzo Monrreal was born on October 31, 1986 in McAllen, TX. He received the Bachelor of Science in Electrical Engineering from the University of Texas Pan American in 2009 and the Masters of Science in Computer Science from the University of Texas Rio Grande Valley in 2015. He has served as a Research Assistant in the Department of Electrical Engineering at the University of Texas Pan American. He has participated in internships for Ford Motor Company, Magic Valley Electric Coop and Raytheon. His professional experience has included teaching many subjects including Math, Concepts of Engineering, Electronics and Advanced Electronics. During his academic career he has been awarded honors such as being inducted into the Society of Hispanic Professional Engineers, the Engineering Honor Society, the Phi Kappa Phi Honor Society and the Golden Key Honor Society. His permanent mailing address is set at 1043 Plena Vista Dr. in Alamo, TX 78516.