

William H. Schultz  
Research in Image Based Modeling  
University of North Carolina at Charlotte  
December 2002

## TABLE OF CONTENTS

Abstract.....	3
Introduction.....	4
Theory of Program Design.....	5
CameraType Class .....	5
Camera Class .....	6
Image Class.....	6
ReferencePoint Class .....	7
ModelerDocument Class.....	8
General Classes.....	8
Presentation of Algorithms .....	10
Multi-dimensional Lagrange Interpolation.....	10
Camera Location Discovery .....	12
Look-At Vector Discovery .....	14
Up Vector Discovery .....	15
User Manual.....	16
System Requirements.....	19
Plans for Later Revisions.....	20
Listing of Appendices.....	21
Appendices.....	On Included CD

## ABSTRACT

This paper discusses the algorithms I devised to discover the location of the camera as well as the relative look-at direction and the relative up direction. Also, a discussion behind the logical arrangement of classes and objects is included. As a proof of the concept of being able to represent various arbitrary surfaces, a discussion of my multidimensional extension to the Lagrange Interpolation Algorithm is also included.

The goal of this senior project was to research and implement an image based modeling algorithm. The goal was to be able to take multiple photographs of arbitrary objects from different directions and to use this to create a three-dimensional model of the object. I did not reach this goal. In Paul Debevec's 1996 PhD Thesis (University of California at Berkeley), he outlines an algorithm that he designed to model simple architectural structures. My minimum hopes for the senior project were to implement this algorithm. However, in my research, I did not find algorithms to find the camera location. Without knowing the camera location, the look-at direction, and the relative up direction, no image based modeling can take place. As such, I have devised and implemented algorithms to find all of these based on user-defined points of reference inside the image.

## INTRODUCTION

After having successfully implemented a ray-tracing program for an undergraduate class, I had an idea that it should be possible to get computers to see in three dimensions, like humans do. This is something that is done instantaneously by the human brain as well as by many animal species. My thought was that it couldn't be so complex that an algorithm can't be written to do this.

The original goal of my senior project was to research Image Based Modeling systems and to implement a system. My idea of Image Based Modeling is to be able to take several photographs of an object from several different angles and to create a 3D model based on those images.

My hope is that the photographs can be taken from relatively arbitrary locations. If the program is to be useful in the future, the program must allow for hand-taken photographs, as opposed to tripod-mounted, measured camera locations. Before any form of triangulation and model building can occur, the camera location must be known. As it turned out, finding the camera location is a significantly more difficult problem than I had expected.

Early in the project, I decided that I needed some basic method of representing the surfaces that would be discovered based on other algorithms. In my high school Algebra II class, I learned about Lagrange Interpolation, and I decided that a multidimensional form of this could be used to represent many surfaces. As such, I also expanded the Lagrange Interpolation Algorithm to allow for higher dimensional surfaces.

## THEORY OF PROGRAM DESIGN

To successfully implement a program that is capable of Image Based Modeling, a basic class hierarchy would be required before a single line of code could be written.

Clearly, a document structure is required. The document needs to be able to handle multiple images (or at least multiple references to images). The program will require an easy method for the user to access multiple images, so some sort of visual image array will be required. In addition, the user will need to be able to open these images and view them independently.

We will require a type of point that is common between multiple images. This reference point will have a single location in real world coordinates as well as references to multiple images and a corresponding location in each of these images.

Several different cameras at several different settings may be used in the program. Each of these types of cameras can take multiple photographs. Each photograph is taken from a specific location with a specific orientation. With respect to the program, a camera type is defined as a single camera at a single setting; a camera is defined as a single position, orientation, setting, and photograph.

At this point, we are ready to create 3d models. We would like to allow for future expansion and the ability to define multiple algorithms for creation of 3d models. As such, we need a super class or protocol to define 3d models, renderers, and creators. Since no subclasses have been fully implemented, a full discussion of this has been omitted.

### CameraType class

This data structure is a relatively simple data structure. There are very few included functions, and most of the functions are simple in nature. The purpose of this class is to

represent a single camera at a single setting. This single camera could take multiple photographs, so it is referenced by other objects, instead of keeping a reference of those other objects.

The object keeps track of the brand and model number of the camera type. Each different camera or setting of the camera is given a different name. The resolution in pixels and the dpi are kept in the data type. The field of view angle for a default zoom is also kept.

A different zoom is not considered a different camera type, as the necessary information can easily be calculated, given an actual zoom.

#### Camera class

A single camera is tied to a single image. This is a simple correlation. A camera will not have more than one image tied to it. An image will not have more than one camera tied to it. Included in the data structure is the location of the camera as well as the direction the camera is looking and the up direction.

Even though the structure itself is quite simple, this object includes a significant amount of code. The camera is capable of finding its location based on several points in the image. In addition, once this is found, it will also find the direction it is looking and the up direction. Each of these is a simple algorithm with a lot of code.

#### Image class

This is a wrapper object for the operating system level image object. There are a few pieces of information that need to be tied with the actual image file. The image object keeps track of all the points in the image that have been defined by the user. It also keeps track of the camera type

as well as the camera. The camera type is kept for easy access, as this can easily be pulled from the camera object.

This object includes functions for keeping track of points in the image that have been defined by the user. There are simple checks to figure out which existing point (if any) has been clicked on by a user.

### ReferencePoint class

The reference point is critical in making everything in the program work as the user would like. The primary use of the reference point is to allow the user to mark locations in images that have well defined locations in real world coordinates. These well-defined reference points allow the cameras to figure out where they are located. In addition, these reference points create a stable foundation off of which to base all 3D models.

A single reference point can easily keep up with how many images contain it. When two or more of these images have a known location, the reference point notices the change and automatically finds the most likely location in real world coordinates. Note: this algorithm has not yet been implemented. This is in the agenda for being implemented in the near future.

The reference point object keeps track of the location of the reference point in real world coordinates (if known) and the point's location in each related image. The database is set up in such a way that the reference point can easily recall it's location in an image just by a simple request.

## ModelerDocument class

This is the final data structure that is crucial to the program design at this point in development. The purpose of this object is simply to keep track of all data related to the user's current project. There are quite a few functions attached to this object that allow retrieval of arbitrary objects just by a name (camera) or number (reference point). This object serves as a central hub for all requests that can't be served easily by any other single object. For example: an arbitrary reference point object would not be able to answer a request for a specific reference point, as it only knows about itself. It could not tell anything about a different reference point. The document knows about all objects. The document class also knows about all windows and all objects that are built to control windows and other various user interface elements.

## General classes

There are many other objects defined for this program. However, very few of them have any significance with respect to actual work that is done with algorithms. Many of these additional objects are controller objects or data structures set up to enable useful user interface elements.

There are two classes that are very significant that have not had any mention up to this point: vector and matrix. These are the basic linear algebra data structures that enable the entire program to function. Though they are invaluable, their method of implementation is arbitrary. The only requirement is that certain values be available when requested. These currently sacrifice a large amount of possible optimizations for a highly platform independent state. Being in strict C++, these classes could be easily compiled and used on any processor running any operating system. However, they could also be replaced with highly optimized assembly code that takes advantage of the incredibly powerful vector processing units in today's processors

(e.g. the G4) for a very large speed boost. This has not yet been done, however, as the most complicated algorithm implemented in this program takes a single second to run without these optimizations. It is clear, though, that other future additions to the program will require significant optimizations in the implementations of these classes.

## PRESENTATION OF ALGORITHMS

Several noteworthy algorithms have been devised during the life of this project. One such algorithm is multi-dimensional matrix-based expansion of the Lagrange Interpolation algorithm. The others are all involved in the seamless discovery of camera parameters that would otherwise have taken a significant amount of time to measure.

### Multi Dimensional Lagrange Interpolation

(Implemented in LagrangeInterpolator.mm – code not tested)

The original Lagrange Interpolation algorithm is an algorithm that will find a polynomial function that intersects any number of listed points. Stepping this up to multiple dimensions involves taking multiple variables into account and creating multiple functions. The one-dimensional Lagrange Interpolation algorithm has no limits on the number of input points. However, when taking multiple input variables into account, limits on the number of input points do arise. To generate symmetrical degree functions (functions in which the highest degree of each variable is the same), the number of input points must be one higher than a multiple of the number of variables (equivalent to 1 mod input dimensions).

The basis of the algorithm is to find several polynomial functions to represent a surface in three-dimensional space. We want an equation in the following form:

$$F1(x,y)=Ax^3+Bx^2+Cx+Dy^3+Ey^2+Fy+G$$

$$F2(x,y)=Hx^3+Ix^2+Jx+Ky^3+Ly^2+My+N$$

$$F3(x,y)=Ox^3+Px^2+Cx+Dy^3+Ey^2+Fy+G$$

We want the functions to have symmetrical degrees of the different variables. The problem can be solved with asymmetrical degrees; however, there are multiple solutions to the problem.

With the assumption that the functions must be of the same degree, we can place restrictions on the input to make this a very simple decision.

The first step to setting up the algorithm is to define in matrix format what the equations should look like. We will later adjust the resulting equation to create a well-defined algorithm. The  $x_m$ 's,  $y_m$ 's, and  $f(x_m, y_m)$ 's all represent known values. We want to find the coefficients,  $Q_m$ .

$$A = \begin{bmatrix} x_1^n & x_1^{(n-1)} & x_1^{(n-2)} & \dots & x_1^2 & x_1 & \dots & y_1^n & y_1^{(n-1)} & y_1^{(n-2)} & \dots & y_1^2 & y_1 & 1 \\ x_2^n & x_2^{(n-1)} & x_2^{(n-2)} & \dots & x_2^2 & x_2 & \dots & y_2^n & y_2^{(n-1)} & y_2^{(n-2)} & \dots & y_2^2 & y_2 & 1 \\ x_3^n & x_3^{(n-1)} & x_3^{(n-2)} & \dots & x_3^2 & x_3 & \dots & y_3^n & y_3^{(n-1)} & y_3^{(n-2)} & \dots & y_3^2 & y_3 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ x_n^n & x_n^{(n-1)} & x_n^{(n-2)} & \dots & x_n^2 & x_n & \dots & y_n^n & y_n^{(n-1)} & y_n^{(n-2)} & \dots & y_n^2 & y_n & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ \dots \\ Q_n \end{bmatrix}$$

$$C = \begin{bmatrix} f(x_1, y_1) \\ f(x_2, y_2) \\ f(x_3, y_3) \\ \dots \\ f(x_n, y_n) \end{bmatrix}$$

These matrix components combine to form:

$$A*B = C$$

Now that we have the function set up in matrix format, we can very easily see how we would create a solution:

$$B = A^{(-1)}*C$$

This simply involves computing the inverse of the variable matrix. It is important to note at this point that this only represents a single component of the necessary overall output. When

restricted to just the above, we are mapping from the x-y plane into the real line. This will technically give us a surface. However, this surface is highly restricted in that it is defined for all x-y and can only have a single z value for a given x-y pair. What we want is a surface that potentially can be any surface imaginable. This can be accomplished by creating a parametric function by using three of these functions to create the three coordinates in three-dimensional space. To create a finite surface, the inputs in the x-y space should be limited.

As currently defined, the multi-dimensional Lagrange Interpolation algorithm will clearly map from a subset of the x-y plane onto any arbitrary continuous surface, given enough input points of reference. To increase the dimension of the output, simply create as many functions as dimensions desired.

## Camera Location Discovery

(Implemented in Camera.mm: -discoverLocationFromCube:)

The human brain is capable of easily figuring out where it is located relative to objects it is viewing. If only one view (eye) is available, the brain will still come up with an answer as to where it is located; it may be a bit confusing (optical illusions), but it does have an answer.

This algorithm requires the user to specify certain points in the image that have a well defined location in real world coordinates. The purpose of this is to set up a real world coordinate system with respect to which we will define the location of all objects in the images. If we have a well-defined object in an image, we can find the direction of the camera based on the object's orientation in the image. Once we have this, we can find the distance to the object based on its size. The direction combined with the distance gives us the location of the camera.

As implemented, though not necessary for the algorithm, the user is required to specify a corner of an object that creates three right angles. The program will figure out which of the four points specified is the corner, based on angles created with each of the other points, and then it will figure out which to use as  $i$ ,  $j$ , and  $k$  vectors, ensuring that the right hand rule is still true ( $i$  cross  $j$  is  $k$ ). Once these are known, the program will rotate around the origin (the corner) calculating what angles would be created in the image from that direction. The program keeps track of which direction is the best match so far. When the search is completed, the best direction stored defines a line in real world coordinates on which the camera is located.

To calculate how the angles would appear, a temporary camera location is defined on the unit sphere, representing the direction to the camera. A pseudo look-at direction, up direction, and right direction are created. The look-at direction is simply the camera looking back at the origin. The other two pseudo axes are arbitrary. For the sake of simplicity, we use the  $k$  axis as the up direction. The only instance in which we would not be able to do this would be if the camera were located on the  $k$  axis. In this case, we would know the direction of the camera (though the program does not allow defining reference points in such a manner that this would be possible). The up direction is adjusted to make it perpendicular to the look-at vector by creating the right direction.

Once these pseudo axes are created, we will use the linear algebra concept of conversion between bases. We will define each of the user-created axes with respect to the pseudo basis. By simply flattening this space along the third axis (the look-at axis), we will create the angles that would be created in the image taken from this direction. This is done by simply ignoring the third coordinate and using the dot products of the resulting vectors to find the angles between the in-image user-defined axes.

At this point, we actually have two possible directions for the location of the camera, as we have found a line on which the camera is located. Two possible directions exist based on the simple comparison listed. To find whether or not the defined best direction is the correct direction, we will simply compare the order (clockwise, counterclockwise) that the points appear in the image with how they would appear if the camera actually were in the direction found.

Once we have a well-defined direction for the camera, we can quite easily find the distance. We know the angle between two points with respect to the camera location based on their locations in the image. We also know the angle between the camera and each axis with respect to the origin based on the camera direction. Knowing these two angles in the triangle created by the camera location, the origin, and a single axis tells us the angle opposite the side represented between the camera location and the origin. Using the law of sines, we now know the camera distance from the origin. We will calculate this for each of the three axes and average the resulting values for a best location.

### Look-At Vector Discovery

(Implemented in Camera.mm: -discoverLookAt:)

The purpose of this algorithm is to find the direction that the camera is looking, assuming that the location of the camera is known. This algorithm requires three reference points in the image with known real world coordinates.

This is a very simple inverse matrix algorithm. We know the location in the image of the center of the image. We can figure out quite easily the angle between the look-at vector and another point in the image. Given three points in the image, we want to find the one vector that

will create the three in-image vectors. The necessary equations are easily set up using the angle definition of the dot product.

## Up Vector Discovery

(Implemented in Camera.mm: -discoverUp:)

This algorithm is similar to the look-at vector discovery algorithm, though it is a bit more complex. Using conversion between bases, we will be able to find the up vector in the real world coordinate system.

Inside of the image, the up vector is quite obvious. It's the vector pointing from the center of the screen to the top of the screen. This is quite easy to describe, but it is more difficult to translate. Using two reference points in the image, we will create a basis. The up vector is defined with respect to this basis.

Separately, we must convert these two points from real world coordinates to camera coordinates. We do this by converting to the same basis defined in the camera location discovery algorithm. Similarly, we will strip the third coordinate off the resulting vector. This time, however, we will also convert from this basis back to the real world coordinate system. In doing this, we have created a virtual screen in front of the camera that is in the real world coordinate system.

Lastly, the two-dimensional vector created two paragraphs above is used in conjunction with the real world screen vectors above to recreate the screen's up vector in real world coordinates.

## USER MANUAL

### Main Document Window

This is the window that represents the document. There are four tabs across the top of the window. Only two of these currently serve any purpose: Images and Cameras. A single document may hold references to many images. It will also keep up with many reference points, cameras, and eventually modeled objects.

#### Images Tab:

This tab contains three important objects. The image array shows all images that have been opened within the document. The slider on the bottom left controls the number of images placed in a single row in the array. The reference point counter on the bottom right shows the total number of reference points that have been created within the entire document.

To add images to the document, choose “Add Image From File...” in the “Images” menu. In the window that pops up, you can choose to add a single file or multiple files simply by selecting more than one file within a single folder while holding down the command key.

To edit attributes of a single image, such as the specific camera that is tied to that image, double click the image in the image array. A new window will open. This window is described later.

#### Cameras Tab:

This tab consists of an outline view (hierarchical list of items) and a button in the lower right corner labeled “New Camera.” (Remove camera does nothing yet.) First, a camera must be created. Click the “New Camera” button. A sheet will pop up with four fields. These fields must be filled in from top to bottom. The pop up menus will list all records created of the

specific type with respect to the above records. The “Identifier” field needs to be unique across the entire document. This is for ease of use when choosing which to attach to a specific image.

To expand an item in the outline view, click the corresponding triangle to the left of the item. The numbers are editable. To edit a value, double click on it. All blocks will appear to be editable, but only the numbers are.

## Images Window

This window is where a large portion of the setup work is done. The majority of the window is used to display the specific image being worked on. In this image, all reference points relating to the specific image are displayed in image. Small circles drawn in the image represent these.

To create a reference point, click on a point in the image and then click the button labeled “New Point.” The button will change to “Set Point.” When a reference point in the image is selected, the button will show “Set Point.” When a point in the image has been specified that is not a reference point, the button will change to “New Point.”

To specify (or change) the location in real world coordinates of the reference point, enter the x, y, and z values in the respective data input fields immediately above the button and click “Set Point.” To change the in-image coordinates, select a reference point, change the horizontal and vertical values, and click “Set Point.” The controls to increment and decrement by one have not yet been implemented.

To specify a reference point in an image that has already been defined in another image, specify the location in the image by clicking, enter the number of the existing reference point in the “Reference Point” field, and click the “Set Point” button. This specifies that these images have this point in common.

To enable the “Camera Information” portion of the window, select an identifier from the “Camera Identifier” pop-up menu. This pop up menu lists only the identifiers of any cameras that were created in the “Cameras” tab of the document window. Once a camera has been selected, the location, look-at, and up vectors will be available to be edited. To set these by hand, enter values and click the button labeled “Set Manually.” To have these values figured out for you, define four reference points in the image with defined real world coordinates. After creating four reference points, the “Discover” button will be enabled. For the algorithm to function properly, the four reference points must create a corner—all defined lines from this corner to the other reference points must be perpendicular to each other. It is not necessary to enter the points in a specific order or define which is which, as the program will find this automatically.

## SYSTEM REQUIREMENTS

Macintosh Running Mac OS X 10.2 or later  
500MHz G3 or faster recommended  
256MB RAM min recommended  
512MB RAM recommended  
Digital Camera HIGHLY recommended

## TEST SYSTEMS

G4 (Digital Audio)  
Dual 533MHz G4  
1GB RAM  
OS X Server 10.2.2

iBook (Dual USB)  
500MHz G3  
640MB RAM  
OS X 10.2.2

Fuji A101 Digital Camera

## PLANS FOR LATER REVISIONS

- The program is incapable of saving the document.
- Camera calibration has not been implemented.
- Triangulating reference point location has been designed but not implemented.
- The reference points should be connected in the image to allow for certain simple modeling algorithms.
- In the main document window exists a tab labeled reference points. This is intended to be a hierarchical view of the reference points similar to the view of the camera types and cameras.
- The user interface needs to be improved to be more user friendly.
- The pop-up menu in the image window listing camera identifiers should be converted into a hierarchical pop-up menu starting with the brand, to allow for easy selection from a very large number of different cameras, including ones not created by hand.
- The camera creation sheet does not disappear when the user is done; it is just hidden behind the document window. The reason for this is unknown.
- The vector tools should be expanded to a full-fledged linear algebra calculator.
- The 3D renderer has been implemented, but is currently unused. The camera is only capable of moving—not rotating. This needs to be updated.
- The camera location discovery algorithm requires the user to specify a corner. As the algorithm is defined, this is optimal, but not necessary.
- The magnification option in the image window should be changed to show the error in the values given by the camera location discovery algorithm.
- The program assumes a  $60^\circ$  native field of view angle for the camera. Enabling the user to modify this would require an extensive rewrite of several objects and thus has not been implemented for this version of the program. This will introduce error in the location of the camera. This needs to be user editable, but will be automatically set by camera calibration.

APPENDIX A  
Research Done  
On CD

APPENDIX B  
Screen Shots  
On CD

APPENDIX C  
Code  
On CD

APPENDIX D  
Functioning Binary  
On CD

APPENDIX E  
Test Images Used  
On CD