Final Document:
3D Scanner: Obtaining 3D Points Through Image Processing

Grady Barrett–BSCS & BSCE
Jeffrey Bishop–BSCS
Tyler Gunter–BSCS

Faculty Mentors: Dr. David Wolff & Dr. Abdullah Kakar

CSCE 499, Spring 2013

Abstract

3D Technology is a growing field, particularly in 3D Printing. Our project does the reverse: it scans a real-world object to create a digital representation. Using a motor, laser line, webcam, and math, we gather and manipulate data to determine 3D points that can be drawn and used to form a mesh. As you will see in this document, the resulting point clouds successfully portray the objects scanned.

Contents

List of Figures

1 Introduction

Three-dimensional technology is at an all-time high. 3D printers have the ability to create actual objects, and 3D scanners are used in applications ranging from the medical industry to architectural preservation [1, 2]. We have achieved success by not only having a working 3D scanner, but being able to explain the functionality and math behind it, both in general and in a presentation setting.

1.1 What is a 3D Scanner?

A 3D Scanner is a device to obtain 3D data from a real-world object that can be used for digital representation of the object. We take images of a laser line crossing an object. This laser line is really a plane of light that we can determine based on the location of the laser line on two perpendicular surfaces.

Every pixel in an image can be thought of as a "camera ray" originating from a distinct point (the camera origin) and passing through an image plane (see Appendix A, Figure 1). We want to find the scalar value that defines the distance to a point in 3D space where our camera ray ends. We can find this scalar, and thus this point, by performing a camera ray-laser plane intersection (Figure 2). These are the object points that our 3D scanner finds and outputs as a point cloud.

1.2 Functional Objectives

The functional objectives in doing this project include the following:

- Construct a scanning environment consisting of a motor, laser line generator, camera, and stage.
- Implement a motor to move a laser line across a scanning scene.
- Successfully calibrate a camera to find its intrinsic parameters and extrinsic parameters.
- Capture image data from a webcam.
- Calculate and generate a set of 3D data points using optical triangulation and image processing.
- Output the data to be used in a 3D modeling application, portraying a good visualization of the real-world object.

1.3 Educational Objectives

There are also educational objectives in doing the project; these are listed below:

- Achieve successful implementation of a medium-scale software application with a hardware component.
- Further develop our software engineering skills, including coding and working in a team environment.
- Learn more about computer graphics, including associated mathematics.

2 Requirements

This section describes various aspects of the project that are necessary to achieve success as well as what the user can expect from the application.

2.1 Performance Requirements

Our application should be able to run on a Windows 7 machine with at least 2 GB of RAM. The entire scan process will take approximately 45-60 seconds, depending on the size of the object. The scanning process will be consistent by regulating the scanning motion and speed using a Servo motor and gearbox.

Results depend on the user's setup and environment, but assuming these are setup correctly, each scan should depict a fairly accurate representation of the object. Any noise should be able to be trimmed off in an application such as MeshLab.

Hardware should be consistent and reliable in order for the scanning process to provide good results.

2.2 User Characteristics

Users of our scanner system should have a basic technical understanding to set up the scanning environment (e.g. camera, laser, stage), load and run code on the Dragon-12 development board, and calibrate the camera and scan objects using our application. Users will likely be hobbyists or general enthusiasts of computer graphics. They should expect a relatively simple procedure with a clean user interface through the entire calibration and scanning process.

2.3 Assumptions

We assume that users of our scanner have the following:

- A basic technical understanding.

- Sufficient memory to accommodate image capture and triangulation processing of image data simultaneously.
- Hardware items:
  - 4.5 V Laser Line generator (SN C005858) or similar
  - Logitech C310 webcam or similar
  - Tripods
  - Continuous rotation Servo motor
  - Dragon 12-+ Development Board
  - A perpendicular scanning stage
  - Computer with Windows installed
- A dark room to scan in.

2.4 Reliability

Given a successful calibration and environment (e.g. lighting), our application should maintain a high level of reliability and provide successful data to the user.

Our scan reliability is limited by what the camera can actually see. Therefore, any surface of the object that is obscured by a surface in front of it will not be able to be picked up in the scan. The user should not expect this missing data to be inferred by the application. Also, reflective objects that distort the laser and dark objects that do not show the laser very well will not produce ideal results.

2.5 Portability

At this time, we only support the Windows platform. Our application will contain Windows-specific components required to run the software. A future adaptation of the application could extend the software to another platform (e.g. Mac, Unix/Linux).

2.6 Required Components

This section describes the hardware and software components that are necessary to implement the project to success.

2.6.1 Hardware

Figure 12 in Appendix A shows a sketch of the hardware setup we will be using.

The servo motor will be connected to the pulse width modulation interface on a Dragon12 microcontroller which will be programmed in C to control the motion of the motor. This motion will rotate the laser across the object being scanned. The Dragon12 will also contains voltage dividers to provide correct voltage to the servo motor and laser. The webcam will be connected via USB to the computer running the application.

2.6.2 Languages, Libraries, and File Formats

Our scanning software is written in C++. It includes the OpenCV library for computer vision and the Qt framework for GUI design. Interacting with the development board is done via the CSerial library (see [3]), and the code for the development board is written in C [3].

Other file formats used in the application include XML files to save and load the intrinsic and extrinsic parameters, as well as VRML files for point clouds.

2.7 Use Cases

We have 2 main uses cases: Calibration and Scanning. Calibration is necessary in order for the calculations to accurately display the 3D points; this includes intrinsic and extrinsic calibration [4, 5].

2.7.1 Calibration: Intrinsic Parameters

Description: The user will need to calibrate the camera they will use in order to get the intrinsic parameters required for scanning. Once calibrated, the output .xml files can be loaded for extrinsic calibration and scanning.

Precondition: Program started with webcam connected.

Normal Postcondition: Calibration successful with intrinsic parameters saved in an .xml file.

Abnormal Postcondition: Calibration unsuccessful and user must exit.

Table 1: Calibration: Intrinsic Parameters

User	Application
1. The user enters the number of horizontal and vertical squares on the chess board and chooses a save directory and then presses “Start” to begin the calibration process.	

Continued on next page

Table 1 – *Continued from previous page*

User	Application
	2. The application stores the number of inner chessboard corners as a variable, which is defined as $(horizontal - 1) \times (vertical - 1)$.
	3. The application turns the camera on and displays the calibration window for the user to position the board for the next picture. A video feed will be displayed in order to make positioning the board easier on the user.
4. The user positions the board for the next calibration picture and presses the “Take Picture” button.	
	5. The application captures the image.
	6. The application processes the image to find the interior corners.
	7a. The corners are found and the application tells the user that the corners were found and increments the number of successes in the interface.
	7b. The corners are not found and the application displays a message that the corners were not found.
8a. The corners were found so the user repositions the board for the next picture and presses the “Take Picture” button.	
8b. The corners were not found so the user repositions the board close to where it was for the picture that did not process successfully. The user then presses the “Take Picture” button.	
	9. Steps 5-8 are repeated until the hard-coded specified number of pictures have been successfully processed.
	10. Once all pictures have been successfully taken and processed, the application performs the necessary remaining calculations, saves the intrinsic data in an .xml file in the user-specified directory, and notifies the user that it was successful.
11. The user presses the “Cancel” button.	
	12. The Calibration window closes.

2.7.2 Calibration: Extrinsic Parameters

Description: The user will need to determine the extrinsic parameters for the camera in order to transform between the world coordinate systems and the camera coordinate system.

Precondition: Camera has been calibrated to determine the intrinsic parameters, which are saved in an .xml file, and the camera and stage are set up in the proper positions.

Normal Postcondition: Extrinsic parameters are successfully determined and saved to an .xml file.

Abnormal Postcondition: Extrinsic parameters were not successfully determined and user must exit.

Table 2: Calibration: Extrinsic Parameters

User	Application
1. The user enters the number of horizontal and vertical squares on the chess board. 2. User clicks "Browse" on each of the "Load Directory" to load the intrinsic data and "Save Directory" to save the extrinsic XML files.	
	3. The XML files that are loaded into the application are parsed. 4a. If the XML is parsed correctly, the application continues. 4b. If the XML is not parsed correctly, the application returns back to step 2 and displays an error message. 5. The application stores the number of inner chessboard corners as a variable, which is defined as $(horizontal - 1) \times (vertical - 1)$. 6. The application turns the camera on and displays the calibration window for the user to position the board on the ground plane for the next picture. A video feed will be displayed in order to make positioning the board easier on the user.

Continued on next page

Table 2 – *Continued from previous page*

User	Application
7. The user positions the board on the back plane of the stage and presses the “Take Picture” button.	
	8. The application captures the image.
	9. The application processes the image to find the interior corners.
	10a. The corners are found, so the application finds the extrinsic parameters and tells the user that the corners were found and increments the number of successes in the interface.
	10b. The corners are not found and the application displays the error processing image message.
11a. The corners were found successfully so the user repositions the board for the next picture and presses the “Take Picture” button.	
11b. The corners were not found so the user repositions the board close to where it was for the picture that did not process successfully. The user then presses the “Take Picture” button.	
	12. Steps 8-10 are repeated with the board on the ground plane of the stage.
	13. Once both pictures have been successfully taken and processed, the application saves the extrinsic parameters in an .xml file in the user-specified director and notifies the user that it was successfuly.
14. The user presses the “Cancel” button.	
	15. The Calibration window closes.

2.7.3 Scanning

Description: The user will scan an object (of an acceptable size) using the laser and the calibrated camera.

Precondition: Camera has been calibrated with calibration data stored in .xml files, and all hardware components are connected properly with the stage in place.

Normal Postcondition: Scanning succeeds and the 3D points are output in a VRML file.

Abnormal Postcondition: Scanning failed.

Table 3: Scanning

User	Application
1. User clicks "Browse" on each of the "Load Directory" to load the calibration data and "Save Directory" to save the point cloud to and enters a filename to save as.	
2. User clicks "Start Scan".	
	3. The XML files that are loaded into the application are parsed.
	4a. If the XML is parsed correctly, the application continues.
	4b. If the XML is not parsed correctly, the application returns back to step 2 and displays an error message asking the user to correct mistakes.
	5. After XML is done parsing, the application will bring up the Region clicking window that will display video of the object being scanned.
6. User will click "Begin Region Clicking".	
	7. The application will take a picture to display for the user to begin clicking the regions.
8. The user will click 4 times: 2 times to define the back plane and 2 times to define the ground plane.	
	9. The application will draw two regions defined by the relative clicks that indicate where the back plane and ground plane are in the image.
10. The user will click 2 more times to define the left and right sides of the object.	

Continued on next page

Table 3 – *Continued from previous page*

User	Application
	11. The application will draw two more regions on either side of the object, leaving the object uncovered to show where the object scan data should be obtained. The “Start Scan” button will be enabled.
12. The user clicks “Start Scan”.	
	13. A dialog box is displayed to tell the user to turn off the lights.
14. The user turns off the lights and hits “OK”.	
	15. The Region clicking window is closed and the Scanning window is opened.
	16. The motor begins to rotate, and the camera captures video until the motor’s rotation time is complete. Scanning process begins. Red channel of images is stored in memory.
	17. Once the motor’s sequence finishes, it rotates in the opposite direction to prepare for another scan. The image processing and algorithms are performed to locate the laser plane intersection with the object, the progress bar is incremented, and status messages appear.
	18. The 3D points are output to the user-specified file, a status message appears for “Exporting data”, and the progress bar increments to completion.
	19. A “Scan Complete” message is shown and the “Done” button is enabled.
20. User clicks “Done”.	
21. Scanning window closes.	

2.8 Budget

The project requires the following items:

- Laser line generator (Provided by CSCE Department)
- 2 Tripods (Already Owned)

- Logitech C310 Webcam or better (Already Owned)
- Parallax continuous rotation Servo motor ($20: Purchased)
- Various electronic items (Provided by CSCE Department)
- Dragon 12 Development board (Provided by CSCE Department)
- Fiber Boards ($10: Purchased)
- Books, etc. (library, online, purchased by Dr. Wolff)
- Visual Studio 2010 (Provided by CSCE Department)
- Code Warrior (Free Special Edition)
- C/C++ (no cost)
- OpenCV (open source)
- MeshLab (no cost)

2.9 Development Documentation

The website for our project can be found at `http://www.cs.plu.edu/~scanners`. All major documents and our final presentation slides and video can be found here.

The code for our project can be found at `http://github.com/scanners/3DScannerProd`.

Each developer of the application has a blog as well. These are linked from our website; the URL for each is also provided below:

- Grady Barrett: `http://barretgm14.wordpress.com/`
- Jeff Bishop: `http://jeff3dscanner.blogspot.com/`
- Tyler Gunter: `http://tylergunter.wordpress.com/`

3 Design

This section describes the research and design of the project that is necessary for efficient and quality implementation.

3.1 Research Review: Math

Here is a description of the mathematical procedures to process images and obtain 3D object points.

Note that $R^{-1} = R^T$, but for coding purposes, we use R^{-1} to maintain the accuracy of data obtained from camera calibration.

Find difference images

```
1. For each row of pixels between the top of the back plane and the bottom of the ground plane...
   1.1. For each column of pixels... [1]
      1.1.1. For each image...
         1.1.1.1. Check if the red component at this pixel in this image is less than the current row's minimum value. If so, replace the minimum with the current pixel's value.
         1.1.1.2. Check if the red component at this pixel in this image is greater than the current row's maximum value. If so, replace the maximum with the current pixel's value.
   1.2. Find the midpoint red component for the current row by dividing the sum of the minimum and maximum by 2:
        $R_{mid}(y) = \frac{R_{min}(y) + R_{max}(y)}{2}$
   1.3. For each column in the "object" region...
      1.3.1. For each image...
         1.3.1.1. Subtract the current row's midpoint red component from the red component of the current pixel of the current image:
                  $R_{\Delta}(x, y, n) = R(x, y, n) - R_{mid}(y)$
```

Find red points

```
1. For each image...
   1.1. For each row in the "back plane" region...
      1.1.1. For each column in the "object" region...
         1.1.1.1. Check if the delta image crosses zero between this column ($x$) and the next column ($x+1$); i.e. $R_{\Delta}(x, y, n) < 0$ and $R_{\Delta}(x+1, y, n) > 0$.
```

[1]This could be changed to only consider the columns in the object region to cut back on memory and processing; by considering the whole image width, we can get a more objective midpoint red component.

1.1.1.2. If yes, interpolate between x and $x+1$ to find the subpixel value where the difference image is zero. This is the left side of the laser line:

$$x_{sub} = x + \frac{0 - R_{\Delta}(x,y,n)}{R_{\Delta}(x+1,y,n) - R_{\Delta}(x,y,n)}$$

Store this pixel (x_{sub}, y) as a red laser point in the current region of the current image.

1.2. Repeat for each row in the "ground plane" region.

1.3. Repeat for each row in the "object" region.

Find laser planes

1. For each image...

1.1. For the "back plane" region red laser image points...

1.1.1. Undistort the image points using the distortion parameters of the camera and the inverse intrinsic matrix K^{-1}

1.1.2. Insert a 1 in the z-coordinate of each undistorted image coordinate to give us an idealized image coordinate u.

1.1.3. Represent the ray through the image point u using parametric form: $R = \{P_C = q_C + \lambda u\}$, where λ is some scalar with $\lambda \geq 0$. Since we are working in the ideal camera system, we can assume q_C is the origin of the camera system, $(0,0,0)$.

1.1.4. Solve for each λ using a ray-plane intersection.

1.1.4.1. Any line will intersect a non-parallel plane at one point. Using the implicit representation of a plane, the point of intersection p is where $n^T(p-q) = 0$, where n is a normal vector of the plane and q is a point on the plane.

1.1.4.2. Pick a point P_W on the respective stage plane, namely $(0,0,0)$ to be our point on the plane q_p, which is in the world coordinates of the respective stage plane.

1.1.4.3. The stage plane surface is the xy-plane in its coordinate system; thus is has a normal vector $(0,0,1)$. This is our normal vector n of the plane in world coordinates of the respective stage plane.

1.1.4.4. The point p that we are trying to find is in the ray we defined above; thus the point $P_C = q_C + \lambda u$ must be converted from camera coordinates to the respective stage plane's world coordinate system:

$$\begin{aligned} P_W &= R^{-1}(P_C - T) \\ &= R^{-1}(q_C + \lambda u - T) \\ &= R^{-1}q_C + R^{-1}\lambda u - R^{-1}T \end{aligned}$$

This will be our point of intersection p_i

1.1.4.5. The ray-plane intersection equation becomes:

$$\begin{aligned}0 &= n^T(p - q)\\ &= n^T(p_i - q_p)\\ &= n^T(R^{-1}q_C + R^{-1}\lambda u - R^{-1}T - q_p)\end{aligned}$$

Then solving for λ:

$$\begin{aligned}\lambda &= \frac{n^T q_p + n^T R^{-1}T - n^T R^{-1}q_C}{n^T R^{-1}u}\\ &= \frac{n^T(q_p + R^{-1}T - R^{-1}q_C)}{n^T R^{-1}u}\\ &= \frac{n^T(q_p - R^{-1}(q_C - T))}{n^T R^{-1}u} = \frac{n^T(q_p - P_{q_C \to W})}{n^T R^{-1}u}\end{aligned}$$

1.1.5. Use each λ to obtain the associated camera coordinate P_C and store it in memory:

$$P_C = q_C + \lambda u$$

1.1.6. After all points P_C have been found, find the best fit line of all P_C's. This represents the laser line across the respective stage plane in camera coordinates.

1.2. Repeat the process for the stored points in the "ground plane" region.

1.3. Find the approximate intersection point between the back and ground laser lines and store it by minimizing the sum of the square distances to both lines [6, p. 17]. This is our point on the laser plane.

1.4. Find the cross product of the vector components of the back and ground laser lines and store it. This is our normal vector of the laser plane.

Find object points

1. For each image...

1.1. For the "object" region red laser image points:

1.1.1. Undistort the image points using the distortion parameters of the camera and the inverse intrinsic matrix K^{-1}

1.1.2. Insert a 1 in the z-coordinate of each undistorted image coordinate to give us an idealized image coordinate u.

1.1.3. Represent the ray through the image point u using parametric form: $R = \{P_C = q_C + \lambda u\}$, where λ is some scalar with $\lambda \geq 0$. Since we are working in the ideal camera system, we can assume q_C is the origin of the camera system, $(0, 0, 0)$.

1.1.4. Solve for each λ using a ray-plane intersection.

1.1.4.1. Any line will intersect a non-parallel plane at one point. Using the implicit representation of a plane, the point of intersection p is where $n^T(p-q) = 0$, where n is a normal vector of the plane and q is a point on the plane.

1.1.4.2. The approximate intersection point found above is our point on the laser plane q_p in camera coordinates.

1.1.4.3. The cross product of the two laser lines is our normal vector n of the laser plane in camera coordinates.

1.1.4.4. The point p that we are trying to find is in the ray we defined above. This point is defined as $P_C = q_C + \lambda u$. This will be our point of intersection p_i.

1.1.5. The ray-plane intersection equation becomes:

$$\begin{aligned} 0 &= n^T(p - q) \\ &= n^T(p_i - q_p) \\ &= n^T(q_C + \lambda u - q_p) \end{aligned}$$

Then solving for λ:

$$\lambda = \frac{n^T(q_p - q_C)}{n^T u}$$

1.1.6. Use each λ to obtain the associated camera coordinate P_C and store it in memory:

$$P_C = q_C + \lambda u$$

1.1.7. Convert P_C to both back plane and ground plane world coordinates using the respective equation:

$$P_W = R^{-1}(P_C - T)$$

1.1.8. Perform bounds checking. Don't include points that...

1.1.8.1. Have a z-coordinate magnitude within 0.10 of the respective stage plane (approximately on the plane).

1.1.8.2. Have a z-coordinate behind the respective stage plane (we don't need points behind a stage plane).

1.1.8.3. Are outside the sphere whose radius is the distance from the camera origin to the back plane origin:

$$\begin{aligned} P_C &= RP_{BW} + T \\ &= R(0,0,0) + T \\ &= T \text{ (the back world origin in camera coordinates)} \end{aligned}$$

$$\sqrt{{x_{pt}}^2 + {y_{pt}}^2 + {z_{pt}}^2} > \sqrt{{x_T}^2 + {y_T}^2 + {z_T}^2}$$

1.1.9. Store P_W in back plane world coordinates in memory to later write to a file.

3.2 Software

This section describes the high level software design for the scanning portion of the project. Calibration is relatively straight forward and our specific design is not included in this document. For more detail into our specific design, please see the source code at `http://github.com/scanners/3DScannerProd`.

3.2.1 State Diagram: Scanning

The state diagram for scanning can be found in Appendix A at Figure 3. This shows the high level transitions to achieve object points from the scan.

3.2.2 Class Design

The class diagram showing our object-oriented, MVC design can be found in Appendix A at Figure 4. The MVC for calibration included an abstract CalibrationController, with subclasses for each calibration type, along with a single CalibrationModel; and the MVC for scanning included a ScanController and ScanModel.

3.2.3 Sequence Diagram: Scanning

The high level sequence diagram for scanning can be found in Appendix A at Figure 5. This essentially flushes out the state diagram into more detail using the class names.

3.2.4 GUI Design

Our GUI interface is implemented using the Qt UI framework. This is a C/C++ framework that simplified GUI implementation and makes it nearly seamless to provide the user with an intuitive user interface to interact with while using our application. There are many reasons for using Qt, some of which are the following:

- High extensibility
- Well Documented
- Qt Stylesheets
- OpenGL Engine [2]

Our GUI itself is a tabbed window. The user can navigate through the GUI by simply clicking on each tab corresponding to the specific phases of the scanning pipeline. The tabs are in the following order:

1. Home
2. Intrinsic Calibration
3. Extrinsic Calibration
4. Scan

The "Home" tab specifies basic instructions for the user. The first stage to scanning an object is running the intrinsic calibration phase. To do this, the user must first click on the "Intrinsic Calibration" tab, and specify the number of internal corners on the chessboard that will be used during the calibration process. Then, they must specify the output directory for the XML calibration file. A snapshot is shown in Appendix A, Figure 6.

If no errors occur, the calibration view will now be displayed for intrinsic calibration. The user must take a set of 20 pictures of the chessboard at differing angles to capture the intrinsic data (focal length, lens distortions, etc). A snapshot is shown in Appendix A, Figure 7.

Once the intrinsic calibration is complete, the user can then proceed with extrinsic calibration. Please note: the intrinsic calibration must be completed first before this step can begin. In this window the user again specifies the number of horizontal and vertical internal squares on the chessboard. The user then specifies the load directory for the intrinsic calibration and then the extrinsic output file location. A snapshot is shown in Appendix A, Figure 8.

Similarly, the calibration view will open if no errors resulted. This time, the user simply takes two pictures: the first is of the chessboard against the back plane; the second, against the ground plane.

Finally, the user can begin the scan. First, the user loads in the intrinsic and extrinsic XML file directories that were specified from calibration. Then an output filename is given. A snapshot is shown in Appendix A, Figure 9.

Assuming that everything is completed correctly, and the user clicks Start, the overlay view will open up. Before a scan can be completed the user must first indicate the back and ground planes and the left and right regions of the object in the frame displayed in the GUI. A snapshot is shown in Appendix A, Figure 10.

[2]While we didnt implement anything with openGL in our project, it can be used to draw vertices, create a polygonal mesh, and interpolate colors in a future project.

Once the regions are selected, the user must then turn off the lights and turn on the laser. Upon the start of the scan, the scanning view appears, shown in Appendix A, Figure 11. The progress bar is updated when processing the data and a message is shown indicating what the application is processing.

After the scan is completed, the application will then output a VRML file in the specified directory.

3.2.5 Serial Communication

A serial communications library found on CodeProject was used in the software application for interfacing with the serial communications functionality of the desktop computer [3]. We chose to use this serial library over implementing straight in the WIN32 API because the library wraps around the WIN32 API, providing cleaner, easier to understand functions for serial communications setup, reading and writing.

3.2.6 Threading

Part of the scanning software application involves sending data to a microcontroller to initiate rotation of a laser line generator. During this hardware scan, the software application is collecting data and will continue to do so until told to stop by the data sent back to the application from the microcontroller. The fact that the hardware scan needs to happen concurrently with the software application's data collection is a problem because waiting for the stop signal from the microcontroller requires using a blocking call to wait for a serial communications event, indicating data has been received. To solve this problem we used Qt threading to move the method containing the blocking call and the code that handled reading the data from the serial line into a separate thread [7, 8]. This serial communications read method sets a boolean flag before exiting. This flag is then used by the data collection method, running in the main thread, to check to see if the hardware scan has completed. If it has completed, then the application stops collecting data and continues on to the processing stage. A Qt mutex is used on the boolean flag to prevent concurrent accessing/modification problems.

3.2.7 Testing Plan

Our project consists of ongoing testing throughout the projects various phases. This helps ensure that the data and application behaves properly throughout each phase. Testing our project includes the following items:

- Calibration (Intrinsic and Extrinsic parameters)
    - Processing images for calibration works as intended.
- Scanning Process

    - Scanning speed is sufficient to ensure optimal data is collected.
    - Output data points display the scanned object with relative accuracy.
- User Interactions
    - The application correctly produces messages to the user.
    - The application makes sure that the XML files that are loaded contain correct data, and that they are parsed correctly.
    - Any user input is correctly handled.
- Miscellaneous Testing
    - The views and models are getting updated correctly.

3.3 Hardware

This section describes the hardware portion of the project.

3.3.1 List of Descriptions of Functional Modules

1. Parallax Continuous Rotation Servo
    1.1. For the final implementation of the hardware a continuous rotation servo was chosen in combination with a speed reducing gear train to rotate the laser line generator. Continuous rotation servos operate a bit differently than traditional servos in that their rotational range is unbounded by removing the potentiometer that acts as the position sensor for the servo. This means that the ability to specify what angle to move the motor to is sacrificed for the ability to rotate continuously. In this project this is a necessity because the gearbox slows the rotation down far enough that multiple rotations of the servo are required to rotate the laser the desired angle (e.g. 180°).
2. Dragon12-Plus Revision F Development Board
    2.1. The Dragon12 is a Freescale HCS12 based development board that provides a host of different microcontroller and other educational functionalities. [9] In this particular project the integrated pulse width modulation functionality is being paired with the onboard 5 Volt power supply, as well as the ability to program interrupts, in order to drive and control the servo motor. One of the two provided serial communications interfaces on the Dragon12 (SCI1) is being employed to provide communication in the form of characters sent between the Windows computer, running the scanning/calibration software, and the Dragon12. This communication is necessary for the purposes of implementing the interrupt functionality on the Dragon12. This allows the precompiled program on the development board to be started and stopped as desired based on signals sent from the

scanning/calibration software. The other serial communications port is restricted to debugging and loading code onto the microcontroller and is, therefore, used for those purposes only.

3. Laser Line Generator

   3.1. The laser line generator is a special type of laser that instead of projecting a single point, as with a laser pointer, projects a laser line. The width of this line is adjustable and the chosen laser color in this case was red.

4. Voltage Divider

   4.1. Conveniently there is a bread board attached to the Dragon12 microcontroller allowing for the construction of basic circuits. In this case the breadboard is utilized to construct a basic voltage divider circuit to provide the correct voltage drop across the laser line generator. This is essentially just a resistor in parallel with the load (the laser). This is necessary because the laser only requires around 3 - 3.5 volts out of the full 5 volts provided by the Dragon12.

5. Desktop Computer running Windows Operating System

   5.1. This is a basic Dell desktop computer running Windows 7. This machine has a serial communications interface port used to compile the stepper motor control program and load it onto the Dragon12 development board. This computer will run the scanning/calibration software application that sends data to the microcontroller via serial connection in order to trigger the interrupt and allow the servo motor control code to run.

3.3.2 Theory of Hardware Operation

Continuous rotation servo use and control

The motor style chosen for the final hardware design in this project ended up being a Parallax continuous rotation servo. In order to understand how the servo is controlled it is important to understand how a servo functions.

Normally the hallmark of a servo is a feedback based design composed of a control chip, motor, gear train and an output position sensor, all connected in a closed loop as seen in Figure 13. This combination of components allows the user to move the servo to specific angles. It is important to note that the average servo has a limited rotational range (e.g. 180°). As stated earlier, however, the motor used in this instance is a continuous rotation servo. This posses a problem is that modifying the servo to make it continuous rotation requires that we remove the potentiometer that acts as the position sensor. This tricks the control chip into continually driving the motor. This means that we can no longer move the motor to specific angles but we can now rotate our motor continuously. In the case of this project, being able to specify precise angular movements is not critical.

Controlling a servo motor requires a technique called Pulse Width Modulation. This is essentially where the servo is sent a series of voltage high pulses of a specified duration with a certain amount of voltage low time between each of these pulses. Traditionally, varying the width of the pulses specifies which angle that the servo will move to. For the Parallax continuous rotation servo (#900-00008), since the position sensor has been removed, varying the pulse width results in a change in rotation speed. Sending the motor a 1.5ms pulse will result in the motor not moving as seen in Figure 14. This can be regarded as the "center" position from which to vary the pulse widths to achieve different speeds and rotation directions. If a shorter pulse width is sent to the servo, say 1.3ms as is Figure 15, then clockwise rotation is achieved and the farther away from the "center" 1.5ms pulse the width gets, the faster the motor will spin. Likewise, if a longer pulse is sent, for example 1.7ms as in Figure 16, then counter clockwise rotation is achieved. The same principle applies that the farther the pulse width gets from the 1.5ms, the faster the motor will spin. In the design for this project, the Dragon12 was used to provide the pulse width modulated voltage control signal to the servo. Pulse widths of 1.6ms for scanning from right to left and 1.3ms for returning the laser to the starting position were used.

Note: In the Parallax #900-00008 continuous rotation servo, the output position sensor potentiometer has been replaced with a trimmer potentiometer to allow the user to calibrate the servo to not move when a pulse width of 1.5ms is sent to the motor [10]. Before using the servo, it must be calibrated so that a 1.5ms signal does actually specify the motor stopped state. This is done by sending a 1.5ms pulse to the servo and adjusting the trimmer potentiometer so that the servo does not move.

Dragon12 Pulse Width Modulation

The Dragon12 microcontroller contains a pulse width modulation hardware interface and corresponding registers to control pulse width settings. When connecting the servo to the Dragon12 it is important to read the manufacturers specifications on which servo wires correspond to the control signal, power and ground. In the case of the Parallax continuous rotation servo, the white wire is the wire to which the pulse width control signal is sent, the red wire corresponds to power and the black to ground. This means that when connecting the Parallax servo to the PP4 output pins on the Dragon12 Rev. F, the white wire will be connected to the left-most pin of PP4 (see Figure 21), as this pin supplies the modulated pulse signal.

In order to send the right pulse width signals to the servo, the correct register values must be selected/calculated. The set up code for the pulse width modulation interface on the Dragon12 can be seen in Figure 18. The important thing to notice here is that on the Dragon12 a "period" and a "duty-cycle" are specified. The period is the total length of the signal that will be repeated. For the servo used in this project this is 20ms. The duty cycle specifies the percentage of the period for which the signal will be voltage high. For example, to obtain a series of 1.5ms pulses separated by roughly 20ms, a period of 50Hz (period is given in terms of frequency on the Dragon12), and a duty cycle of 7.5% are provided to the PWMPER4 and PWMDTY4 registers, respectively. One thing to notice is that this actually means that there is only a 18.5ms delay between each 1.5ms pulse. This was not found to cause any issues with the Parallax servo used here but if it did cause issues, the period could simply be extended to compensate for part of the period being taken up by the voltage high pulse. Another thing to notice is that the period frequency specified in the Dragon12 code is derived via an amplified signal from the microcontroller's master clock. For the Dragon12, this signal is 24MHz, which is then divided down to 50Hz using a series of other registers as seen in the code snippet in Figure 18.

The channel 4 (PP4) PWM signal is enabled by loading the PWM Enable register (PWME) with the hex value corresponding to the channel 4 bit in the register (0x01). This causes the motor to begin spinning according to the PWM settings specified. A delay method can be written to allow for a specified period of time to pass before disabling or changing the PWM signal, thereby stopping or changing the rotation of the motor. This delay method, as is the case in this project, can be used in combination with changes made to the duty cycle to specify how far and fast the laser line generator scans across the object. It can also be used to return the laser line generator to its starting position to automate the process of performing multiple scans.

Interrupt Events, Interrupt Service Routines and Serial Communication on the Dragon12

Information in this section as well as more detailed information on interrupts and serial communication can be found in "Microcontroller Theory and Applications: HC12 & S12" and "Designing with Microcontrollers - The 68HCS12" [11, 12]. The Dragon12 is simply a stripped down computer. This means that its default mode of operation is to start running the code that has been compiled onto it as soon as compilation finishes. In order to be able to control the Dragon12's code execution, there must be a connection between the PC and the board. A serial communications interface on the Dragon12 provides for a serial connection between the PC and the development board. This connection provides several asynchronous interrupt events. For this project, the Receiver Interrupt Enable (RIE) interrupt will serve as the backbone for controlling the execution of code on the Dragon12.

When an interrupt event is triggered on the Dragon12, execution of the current program is halted and a programmer specified interrupt service routine (ISR) is run. In this project, the ISR only contains code to clear the interrupt RIE interrupt flag (done by reading from the SCI1SR1 register), read any data on the serial line sent from the software application (done by reading from the serial data register SCI1DRL), and set a flag variable indicating that the hardware scan should occur. The flag variable described in the last step of the ISR is checked in a conditional statement in the infinite loop residing in the main method. This is the "wait state" that the Dragon12 sits in until the interrupt is triggered and the hardware scan flag is set. The conditional statement in this loop contains the code necessary to rotate the servo motor, which may take anywhere from 10 seconds to a minute or more depending on how slow the laser line generator spins. This is important to realize when designing the ISR code because if this servo rotation code were moved into the ISR, thereby extending the ISR's length from milliseconds to seconds, the Dragon12 might behave unexpectedly. This is due to the fact that ISR's are suppose to be as short as possible [13].

In addition, in order for interrupts to work properly, the memory locations for the interrupt vectors must be linked with the address of the ISR that the programmer wishes to be called when that particular interrupt occurs. The memory locations of the various available interrupts for the Dragon12 are defined in Figure 19. In this case, the SCI1 interrupt source is linked with the ISR described earlier. This will allow the specified ISR to run when the receiver interrupt is triggered (i.e. when we receive data from the software application over serial).

The last important element here is that when the interrupt is triggered and the servo rotation code finishes, a signal must be sent back to the software application running on the PC, indicating to the application that the stepper motor has finished the hardware side of the scanning process. This allows the software application to stop collecting data and start processing it.

In this project, a bit will be sent back to the windows machine when the servo motor scan routine finishes executing, indicating to the scanning application that the microcontroller has finished the hardware side of the scanning process.

Reducing Gear Train

In order to get the laser line generator to spin extremely slowly (preferably below 1 RPM) a reducing gear train was designed. An example of this can be seen in Figure 20. The design used in this program made use of four different gear sizes. One with 8 teeth, one with 12 teeth, one with 36 teeth and one with 40 teeth. The 12 toothed gear served as the drive gear, which was attached to the servo and the 40 toothed gear served as the final gear and platform for the laser line generator. The 12 and 36 toothed gears were connected together as shown in Figure 20 to reduce the overall speed. The gearing ratio represented by this configuration is calculated by multiplying the successive ratio of drive gears divided by the driven gears. In this case this looks like the following:

$$\frac{20}{36} \times \frac{12}{36} \times \frac{12}{36} \times \frac{12}{36} \times \frac{36}{40} = \frac{1}{54} \tag{1}$$

This yields a 54:1 slow down ratio. If the servo is rotating at 10 RPM, then this will give us a .19 RPM rotation rate for the final gear, on which the laser is mounted. This is a much more useful RPM magnitude for the laser. This will ensure that an adequate amount of data is collected so that the final point cloud has a good density of points.

Voltage Divider

A simple voltage divider is employed to reduce the voltage drop across the laser line generator. This is necessary because the laser only needs 3.5 volts but the Dragon12 provides 5 volts. The voltage divider is simply a 56Ω resister in series with the 5 volt voltage source, the combination of which is in parallel with the laser line generator. This gives us a voltage drop of around 3.5 volts across the laser line generator.

3.3.3 Block Diagram

A block diagram of the hardware modules associated with driving the stepper motor, which is composed of the Dragon12 development board, servo motor, laser line generator and computer running windows can be seen in Figure 21..

4 Implementation

In order for our goals to be realized and for this project to be a success we had to coordinate with the three members of our group accordingly, adhere to our software development process, and often times work on the same code base while writing code. This section describes some of these details.

4.1 Team Communication and Collaboration

Communication was largely executed via emails, GitHub commit notes, and keeping each other informed via SMS text. If a group member was unable to attend a meeting, it was disseminated to the other group members so they were informed and updated. On a few occasions video conferencing via Skype was used to meet with the group remotely if necessary to collaborate ideas, problems, etc. to the absent member. This was largely beneficial because it allowed for work to be completed even if group members were not at the scheduled meeting time. Also, when designing our presentations, we were able to utilize the collaborative structure of the cloud-based Google docs, allowing us to work on the same presentation from multiple computers.

Our group meetings during the implementation phase were kept relatively constant throughout the semester. This helps to get the group in a good pattern. Here is what a typical week of meeting times looked like:

- Tuesday: 3:45-6:30pm (met with Dr. Wolff at 3:45)
- Wednesday: 3:45-6:30pm
- Thursday: 3:45-6:30pm
- Saturday: occasional, as needed

4.2 Software Development Process

We followed a somewhat parallel phased development cycle throughout this project. While one group member worked on math and algorithmic processes, another worked on GUI design, memory management and architecture, and the third worked on hardware planning and implementation. Overall there were three main phases to the project:

- Calibration
  - Intrinsic
  - Extrinsic
- Scanning
- Hardware Integration

Testing occurred throughout the process to ensure that the results that we were obtaining were accurate and correct.

To facilitate this parallel development and to keep a record of changes that were made, we utilized GitHub for our version control system. This allowed the person working on the hardware integration to branch off of the master branch that the other two group members were working on. It also allowed us to work simultaneously on the same project and merge any conflicts that occurred.

4.3 Complications

This section describes problems that we ran into during our development and how we resolved those problems. The problems are as follows:

4.3.1 Camera Settings in OpenCV

At the onset of our project we wanted to automate camera settings in the application so that the user wouldnt have to be worried about explicitly configuring their camera to achieve the best result throughout the scanning process. However, this is a problem that was unable to be resolved. There are simply too many differing cameras to support these settings, and unfortunately, this is the part of our project that we were unable to resolve.

4.3.2 Chessboard Orientation

On the Saturday before we presented our project, we discovered that the orientation of the chessboard does matter. It turns out that OpenCV looks for a specific color combination at which to begin finding the corners. Based on our implementation, the user must hold the chessboard with a black square in the bottom-right hand corner for intrinsic and back plane extrinsic calibrations and in the upper-left hand corner for ground plane extrinsic calibration. This location is where OpenCV will begin its corner-finding algorithm.

4.3.3 Vertex Data

There were many hours spent trying to figure out why the scan output wasn't working. It turns out that there was a math error. When finding the lambda scalars to find the laser planes, we were converting a normal vector based on the world-to-camera transformation equations. However, these equations are for points, not vectors. So we switched to doing the calculations in world coordinates. However, there was another error in that we were converting λu rather than the full camera point $P_C = q_C + \lambda u$. Once we changed that and re-solved the ray-plane equation for λ (see Section 3.1), we were able to obtain results.

4.3.4 Hardware Roadblocks

Interrupt Vector Table

The first issue we ran across related to implementing the interrupt service routine and connecting it to the Receive Interrupt Event interrupt. It was difficult to find information about syntax for performing this task in C specific to the Dragon12. This resulted in quite of bit of trial and error until we finally pieced together enough information to figure out how this is done. Part of the problem in debugging this issue was that we were also having issues with the software application side of the serial communications so we were never quite sure whether the problem was that we weren't sending data successfully to the microcontroller, or if the interrupt just wasn't connected correctly. We finally resolved this debugging problem by using PuTTY to send characters to the microcontroller over serial. This allowed us to focus on problems with the microcontroller. Once we had solved the vector interrupt table issues and knew the interrupt was triggering properly, this freed us up to successfully debug the software application serial communications.

Serial Handshaking

In terms of the software application, all of the problems with not being able to send information to the microcontroller were related to handshaking. Handshaking is an essential process in serial communications because it is the interaction that establishes the connection between the hardware devices and sets the type of negotiation for when devices are ready to send and/or receive data. The primary ways of preforming handshaking are hardware handshaking and software handshaking. Hardware handshaking uses a dedicated wire to handle this process, and as such requires that the serial cable and both devices support it. Software handshaking shares the receive and transmit wires in a serial cable and uses these to send data signals back to perform handshaking tasks. Originally we had specified the type of handshaking to be hardware handshaking when initializing serial communications in the software application. This was done because is was assumed that the microcontroller and serial cable supported hardware handshaking. The problem that we encountered was that the serial code in the software application would run but the interrupt would not trigger. We gathered that this was likely a problem with initialization so we changed the handshake type to software and everything worked perfectly.

Motor Choice

The last major problem that we encountered with the hardware was related to the initial choice of motor. Originally we chose a stepper motor. This became a problem later on because we needed to spin the motor very slowly and smoothly. This is an issue because a stepper motor moves in discrete degree steps (in our case 1.8°). When the motor is slowed down these steps result in a very choppy rotation of the motor. This could be resolved by implementing microstepping to divide each step into multiple smaller steps. Unfortunately the Dragon12 does not have the necessary hardware connections for this. As a result we started looking for alternative motor types. There were some old basic robots that had continuous rotation servos on them. After playing around with these servos and pulse width modulation we determined that this was the direction that we needed to go but that the servo we had could not spin slow enough. We ended up getting a Parallax continuous rotation servo because of the fact that it had a linear response to ramping from 1-50 RPM. This allowed us to run the servo very slowly and smoothly. Unfortunately not even this servo rotated slowly enough so we ended up implementing the reducing gear train described earlier, which significantly reduced the final rotation speed of the laser line generator.

5 Future Work

There were a few complications we ran into throughout the development process, as is expected. The following are parts of our project that we did not get to completely implement. All of the listed items were unachievable due to time constraints. We were focused on getting the basic project features implemented first and wanted to ensure that it was working solidly before moving onto more complicated portions.

5.1 3D Mesh Algorithm

This would allow the user to create a mesh around the object in which was scanned. To do this, we would have liked to implement an algorithm that would take standard openGL concepts and drawn triangles and interpolated a mesh throughout each triangle.

5.2 Color Interpolation

Color interpolation was something we were largely excited to achieve. We would have needed to take a picture of the object before prompting the user to turn off the lights. Once we had all the vertex data, we could then apply the pixel color at that location to the object to create a colored representation of the object.

5.3 Merging Point Clouds

Ultimately, this was our end-goal for the project. We wanted to take multiple scans of the object at various angles, and then reconstruct the scans into one consolidated 3D model of the object.

6 Conclusion

We believe that we have achieved successful implementation in our project. If a later group or student wanted to continue or extend this project, we hope these details are sufficient for them to be successful. We simply did not have enough time to achieve all possible features. Knowing as much about 3D scanning as we do now, we believe we would be able to get much closer to implementing more. Like any project, there is a certain learning curve to understanding the concepts. If we were to start this project over, we would have a stronger knowledge and understanding of calibration, scanning, and hardware. In the future, we hope this document serves as a guide to interested students.

Glossary

extrinsic parameters	The translation and rotation vectors relating the stage's "world coordinates" to "camera coordinates".
intrinsic parameters	A model of the camera's geometry and a distortion model of the camera's lens [4, p. 371].
stage	Two perpendicular boards in which the object to be scanned is placed.
triangulation	A mathematical process of determining an intersection point using geometry.

References

[1] T. Tong, "Medical applications in 3d scanning." `http://blog.3d3solutions.com/bid/78455/Medical-Applications-in-3D-Scanning`, 2011.

This blog provides information from the 3D3 Solutions company, which makes professional-grade 3D scanning products. The blog provides examples of how 3D scanning is used in the medical industry.

[2] "3d digital corp applications: Architecture." `http://www.3ddigitalcorp.com/applications/architecture`, 2012.

This webpage provides architectural applications of the scanners from 3D Digital Corporation. Their website also lists other applications, including manufacturing and automotive.

[3] R. de Klein, "Serial library for c++." `http://www.codeproject.com/Articles/992/Serial-library-for-C`, November 2003.

This is the serial library we used in the software application for communicating with the microcontroller.

[4] G. Bradski and A. Kaehler, *Learning OpenCV*. Sebastopol, CA: O'Reilly Media, Inc., 2008.

This book provides a learning tool for OpenCV, as well as examples. We are using OpenCV to program our application; the book provides direct assistance for calibrating a camera for use in OpenCV and we will extend the calibration parameters to use in the scanning process.

[5] J.-Y. Bouget, "Camera calibration toolbox for matlab." `http://www.vision.caltech.edu/bouguetj/calib_doc/index.html`, 2010.

This website contains the toolbox for camera calibration in Matlab. It also contains information and examples for using it.

[6] D. Lanman and G. Taubin, "Build your own 3d scanner: Optical triangulation for beginners." `http://mesh.brown.edu/byo3d/index.html`, 2012.

This website provides a major resource for how to build a 3D scanner. It includes critical mathematics necessary for triangulation as well as helpful information regarding the scanner system itself.

[7] "Qthread class reference." `http://qt-project.org/doc/qt-4.8/qthread.html`.

This is the class reference for the QThread Qt object used in our project to implement threading.

[8] "Threading basics." `http://qt-project.org/doc/qt-4.8/thread-basics.html`.

This article describes the basics of implemeting threading in Qt.

[9] *Dragon12-Plus Trainer: For Freescale HCS12 microcontroller family: User's Manual for Rev. F board Revision 1.06.*

This is the user manual for the version of the Dragon12-Plus microcontroller used in this project. It contains definitions for pins on the board as well as some general information about features and specifications of the microcontroller.

[10] Jan, "Continuous-rotation servos and multi-turn servos." `http://www.pololu.com/blog/24/continuous-rotation-servos-and-multi-turn-servos`, 2011.

This article contains a useful description of how a servo is designed and how it is modified to make it continuous rotation.

[11] D. J. Pack and S. F. Barrett, *Microcontroller Theory and Applications HC12 & S12.* Upper Saddle River, NJ: Pearson Education Inc., 2008.

This book provides a wealth of information relating to how certain technologies are available on an HC12 and S12 microcontrollers. The information used in this project pertained primarily to how serial communications are done on an S12 microcontroller as well as how that ties into interrupts.

[12] T. Almy, *Designing with Microcontrollers - The 68HCS12.* Lulu.com, 2006.

This book provides information about S12 microcontrollers similar to the book by Daniel Pack and Steven Barrett. Sections used most included those on how serial communiation occurs in general as well as the interrupt process and associated registers on S12 microcontrollers.

[13] P. Kaur, "Interrupts short and simple: Part 1 - good programming practices." `http://www.embedded.com/design/programming-languages-and-tools/4397803/Interrupts-short-and-simple--Part-1---Good-programming-practices`, October 2012.

This is an article outlineing some good programming practices when implementing interrupts in embedded systems. It is in this resource that I first encountered a clear description of how interrupts being too long can cause strange behaviour and how this problem can be addressed. This ultimately let to the successful debugging of this problem as encountered in this project.

[14] P. Inc, *Parallax Continuous Rotation Servo (#900-00008).*

This document provides the specifications for the Parallax 900-00008 continuous rotation servo motor. It also provides information on the pulse widths for running the motor and some generic code samples for programming a microcontroller. Unfortunately, the model for pulse width modulation used in this code is not applicable to the Dragon12.

[15] M. Mazidi, "Controlling servo motor with pwm (pulse width modulation) on dragon12 plus trainer board." `http://www.microdigitaled.com/HCS12/Hardware/Dragon12/CodeWarrior/PWM_Servo_motor_Cprog.txt`.

This is some example code for pulse width modulation using the dragon12 on which our code was based for driving the stepper motor.

[16] "Gears and drivetrains." `http://myweb.cwpost.liu.edu/magot/classes/csc/RobotFundamentals/downloads/gears_drivetrain.htm`.

This is a short article about gearing mechanisms that contains a generic example of a reducing gear train used as the theoretical basis for the gear train built in this project.

[17] "C++ documentation." `http://www.cplusplus.com`, 2012.

This provides a helpful reference for C++ functionality.

[18] G. Taubin, "ENGN 2502 3d photography." `http://mesh.brown.edu/3DP/index.html`, 2012.

This website provides additional information about 3D processing and photography from Brown University. It includes some of the professor's lecture slides, which can serve as a resource.

[19] G. Taubin, "3d photography and image processing." `http://mesh.brown.edu/3DPGP-2009/index.html`, 2009.

This website provides additional 3D information and mathematics from Brown University. It also gives a resource for post-processing information.

[20] T. Shifrin and M. R. Adams, *Linear Algebra: A Geometric Approach.* New York, NY: W.H. Freeman and Co., 2 ed., 2011.

This book serves as a math resource for learning about some of the mathematics needed for matrix calculations and vector.

[21] "How to use opencv to capture and display images from a camera." `http://opencv.willowgarage.com/wiki/CameraCapture`, 2011.

This website provided us a start on getting images from a webcam.

[22] Utkarsh, "Calibrating and undistorting with OpenCV in C++ (oh yeah)." `http://www.aishack.in/2010/07/calibrating-undistorting-with-opencv-in-c-oh-yeah/`, 2010.

This person's description of calibrating a camera provided us with a C++ implementation, which was a helpful resource for our own implementation.

[23] "OpenCV camera_calibration.cpp." `docs.opencv.org/_downloads/camera_calibration.cpp`.

This was actual code for camera calibration that helped us to figure out how to successfully implement calibration in C++.

[24] "OpenCV C++ documentation." `http://opencv.willowgarage.com/documentation/cpp/`.

The documentation provided us with what methods and variable types, such as Matrices and vectors, to use for implementation.

[25] “Qt 4.8 documentation.” `http://qt-project.org/doc/qt-4.8/index.html`.

This provides the documentation needed to develop our GUI using the Qt framework.

[26] “How can i handle events in the titlebar and change its color etc ?.” `http://qt-project.org/faq/answer/how_can_i_handle_events_in_the_titlebar_and_change_its_color_etc`.

This provided the code for changing the natural Windows look to a custom titlebar look-and-feel.

A Appendix A

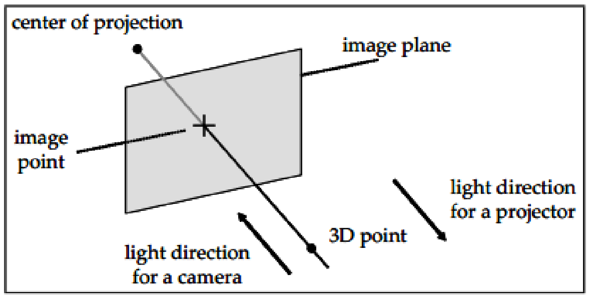


Figure 1: The pinhole camera model showing a camera ray passing through the image plane [6].

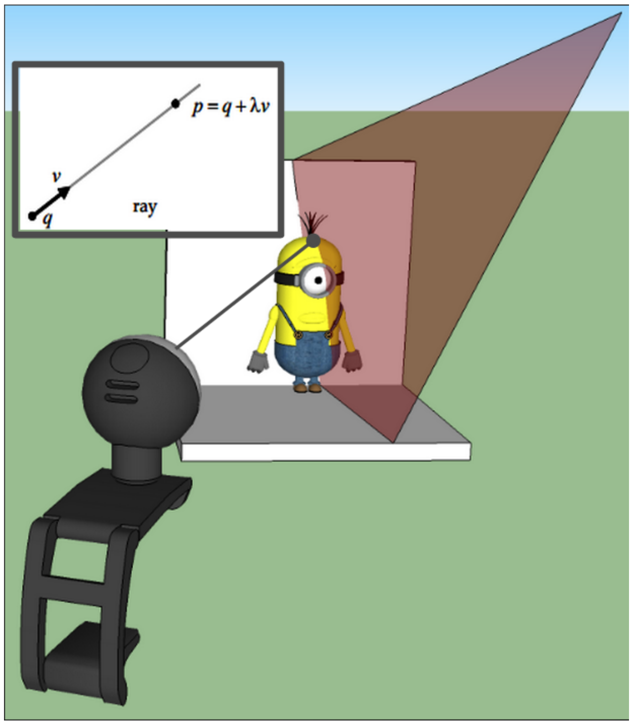
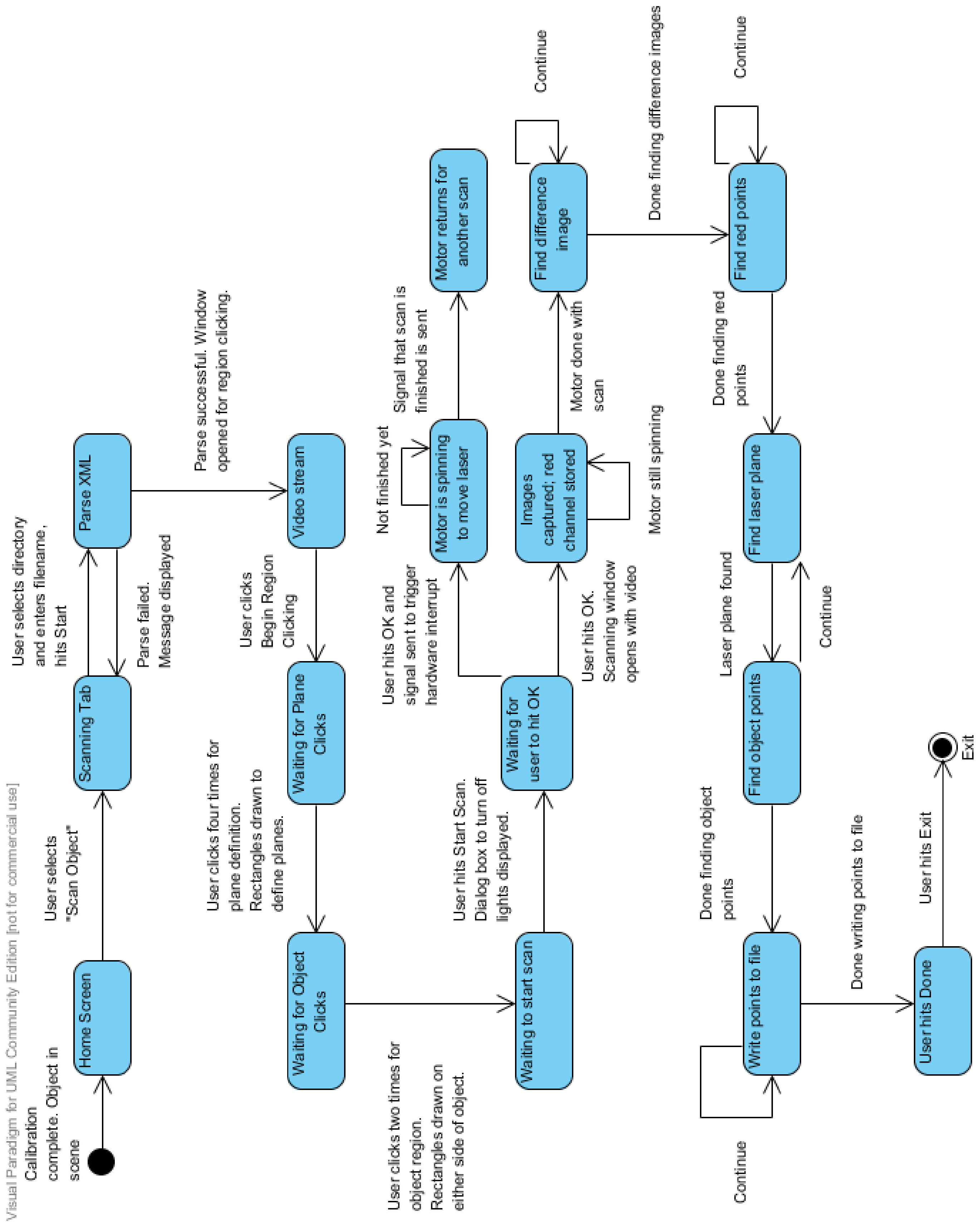


Figure 2: This diagram shows a camera ray from the camera to the point where it intersects the laser plane.



Visual Paradigm for UML Community Edition [not for commercial use]
Calibration complete. Object in scene
Home Screen
User selects "Scan Object"
Scanning Tab
User selects directory and enters filename, hits Start
Parse failed. Message displayed
Parse XML
Parse successful. Window opened for region clicking.
Video stream
User clicks Begin Region Clicking
Waiting for Plane Clicks
User clicks four times for plane definition. Rectangles drawn to define planes.
Waiting for Object Clicks
User clicks two times for object region. Rectangles drawn on either side of object.
Waiting to start scan
User hits Start Scan. Dialog box to turn off lights displayed.
Waiting for user to hit OK
User hits OK and signal sent to trigger hardware interrupt
Motor is spinning to move laser
Not finished yet
Signal that scan is finished is sent
Motor returns for another scan
User hits OK. Scanning window opens with video
Images captured; red channel stored
Motor still spinning
Motor done with scan
Find difference image
Continue
Done finding difference images
Find red points
Continue
Done finding red points
Find laser plane
Continue
Laser plane found
Find object points
Done finding object points
Write points to file
Continue
Done writing points to file
User hits Done
User hits Exit
Exit

Figure 3: This shows the state transitions we will use for the scanning phase.

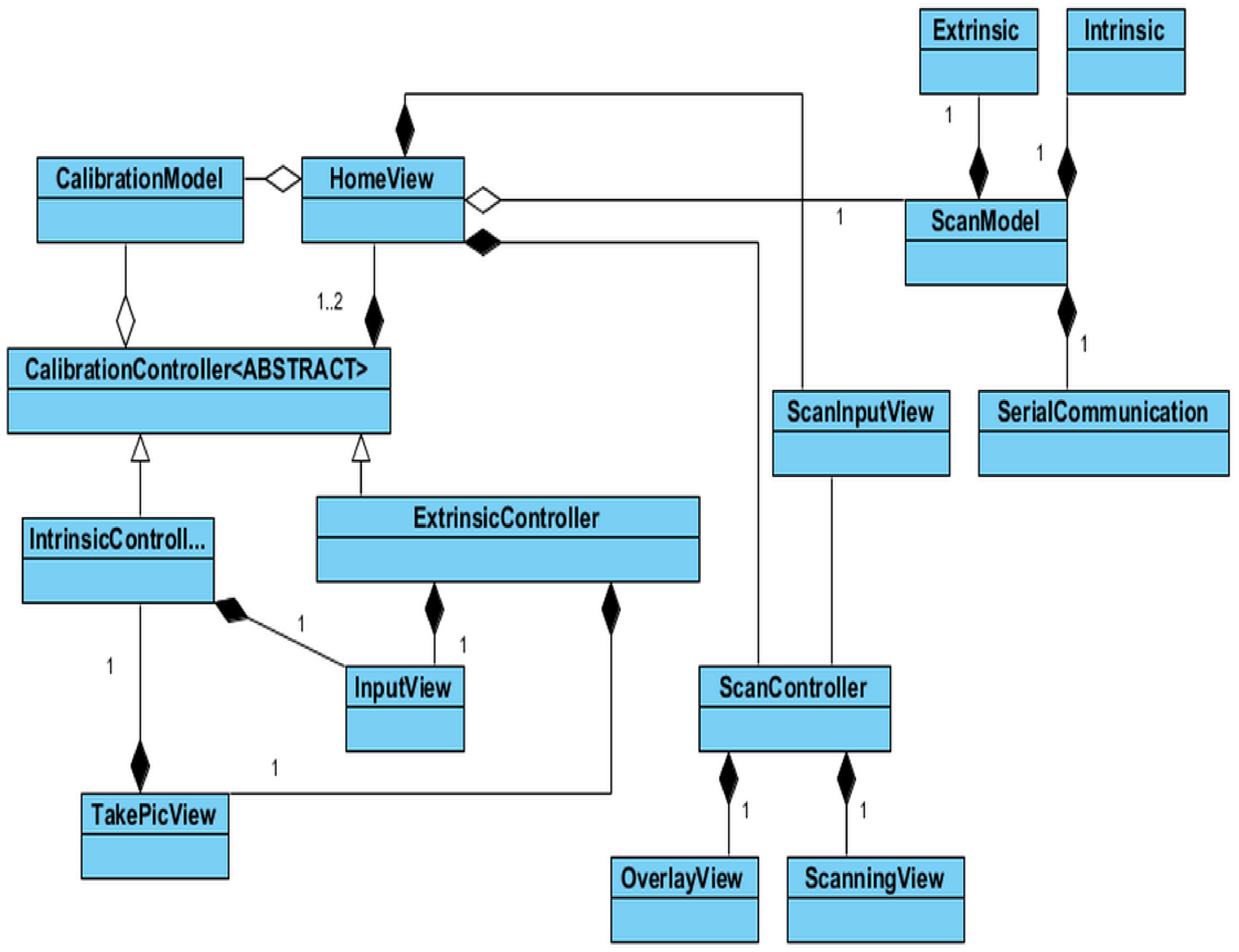
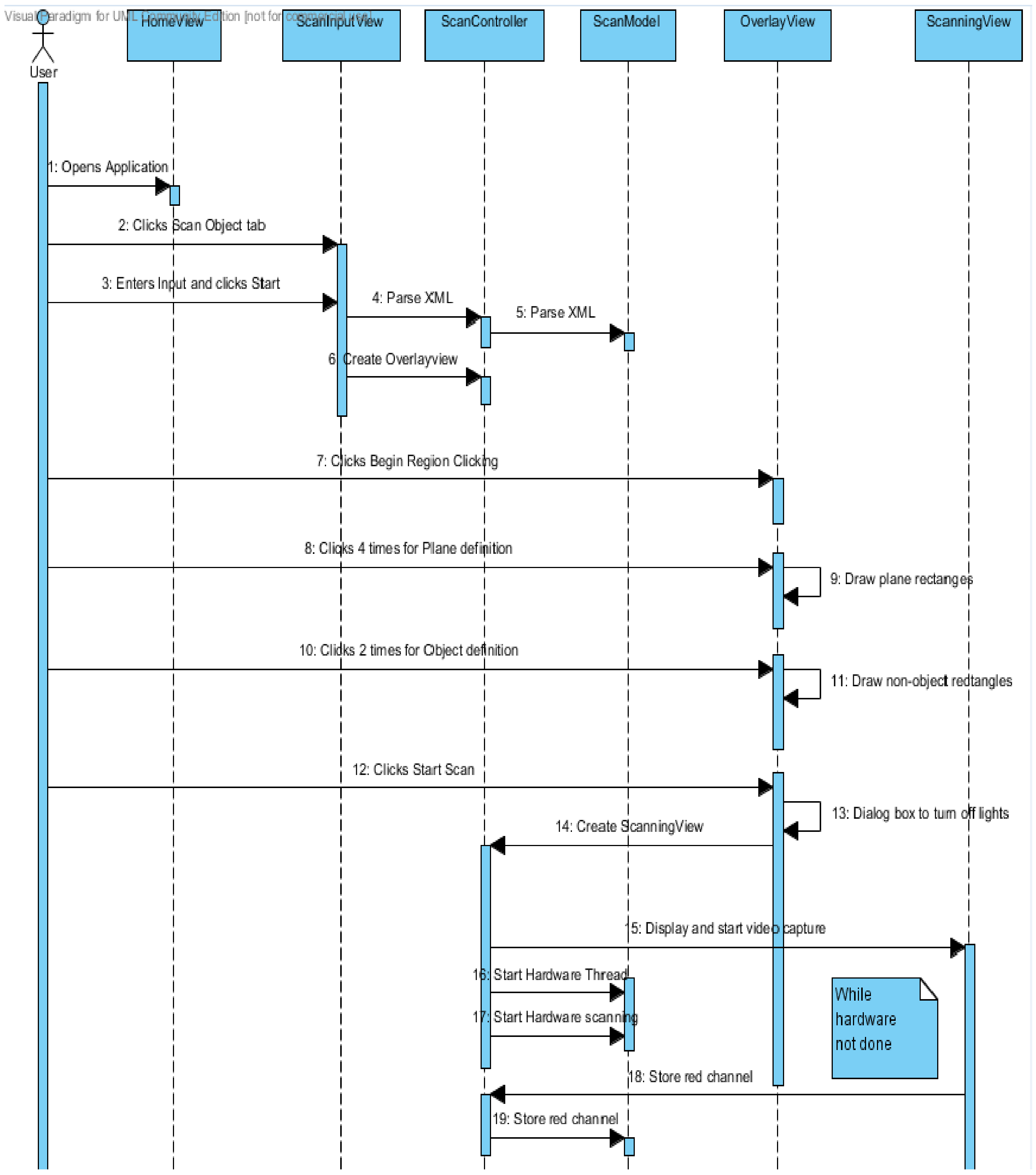


Figure 4: The high level class diagram.



Visual Paradigm for UML Community Edition [not for commercial use]
User
HomeView
ScanInputView
ScanController
ScanModel
OverlayView
ScanningView
1: Opens Application
2: Clicks Scan Object tab
3: Enters Input and clicks Start
4: Parse XML
5: Parse XML
6: Create Overlayview
7: Clicks Begin Region Clicking
8: Clicks 4 times for Plane definition
9: Draw plane rectanges
10: Clicks 2 times for Object definition
11: Draw non-object rectangles
12: Clicks Start Scan
13: Dialog box to turn off lights
14: Create ScanningView
15: Display and start video capture
16: Start Hardware Thread
17: Start Hardware scanning
While hardware not done
18: Store red channel
19: Store red channel

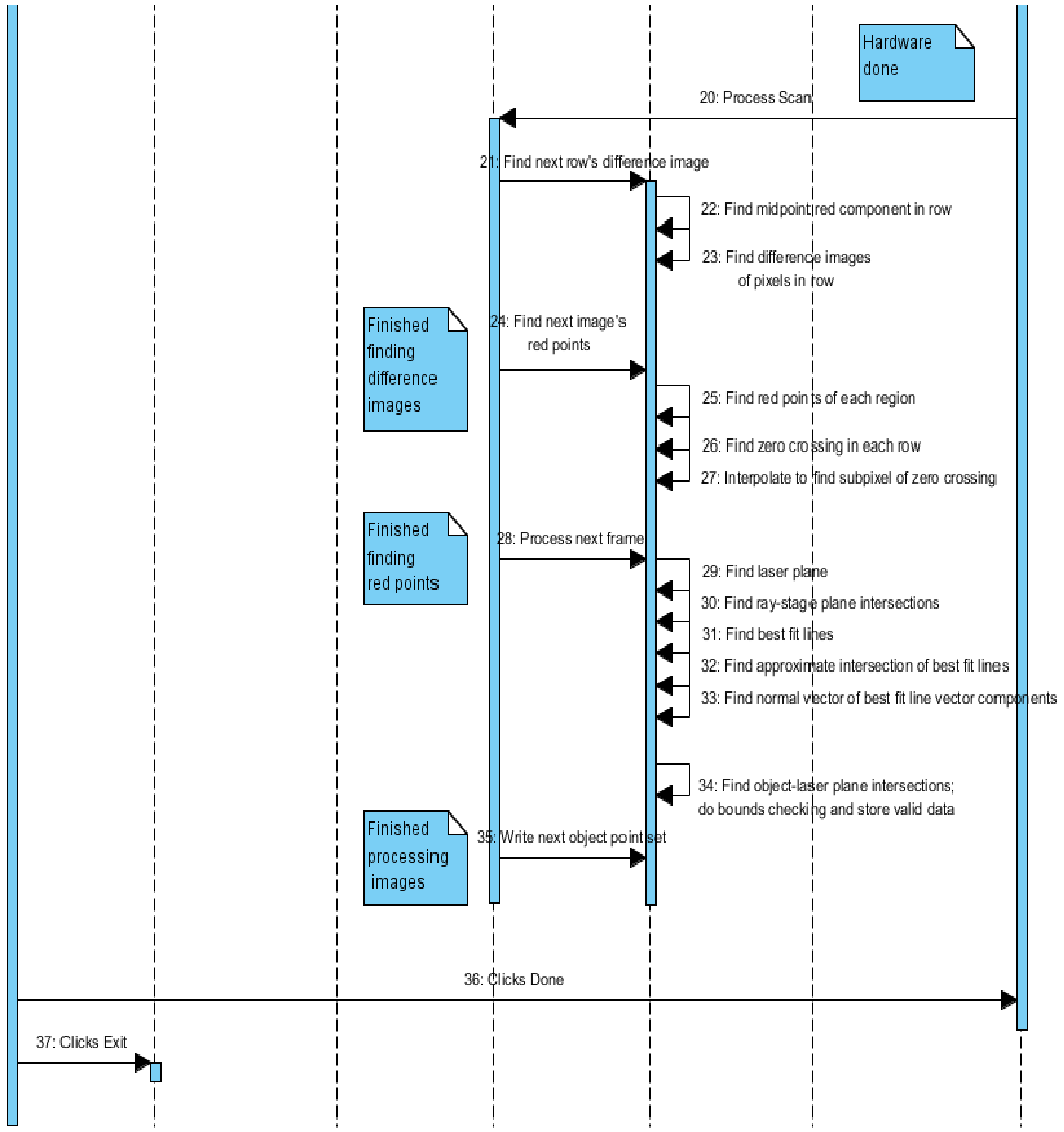
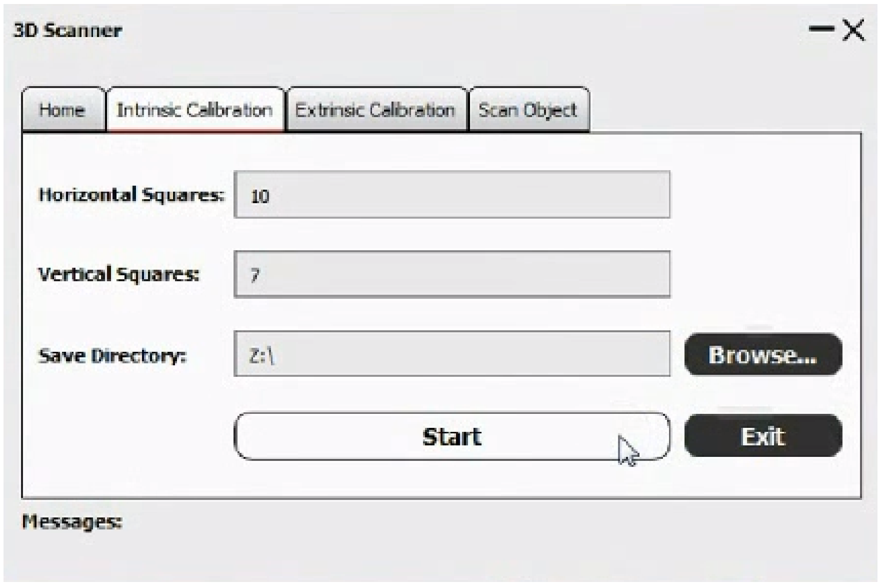

Hardware done
20: Process Scan
21: Find next row's difference image
22: Find midpoint red component in row
23: Find difference images of pixels in row
Finished finding difference images
24: Find next image's red points
25: Find red points of each region
26: Find zero crossing in each row
27: Interpolate to find subpixel of zero crossing
Finished finding red points
28: Process next frame
29: Find laser plane
30: Find ray-stage plane intersections
31: Find best fit lines
32: Find approximate intersection of best fit lines
33: Find normal vector of best fit line vector components
34: Find object-laser plane intersections; do bounds checking and store valid data
Finished processing images
35: Write next object point set
36: Clicks Done
37: Clicks Exit

Figure 5: The high level sequence diagram for scanning.


3D Scanner

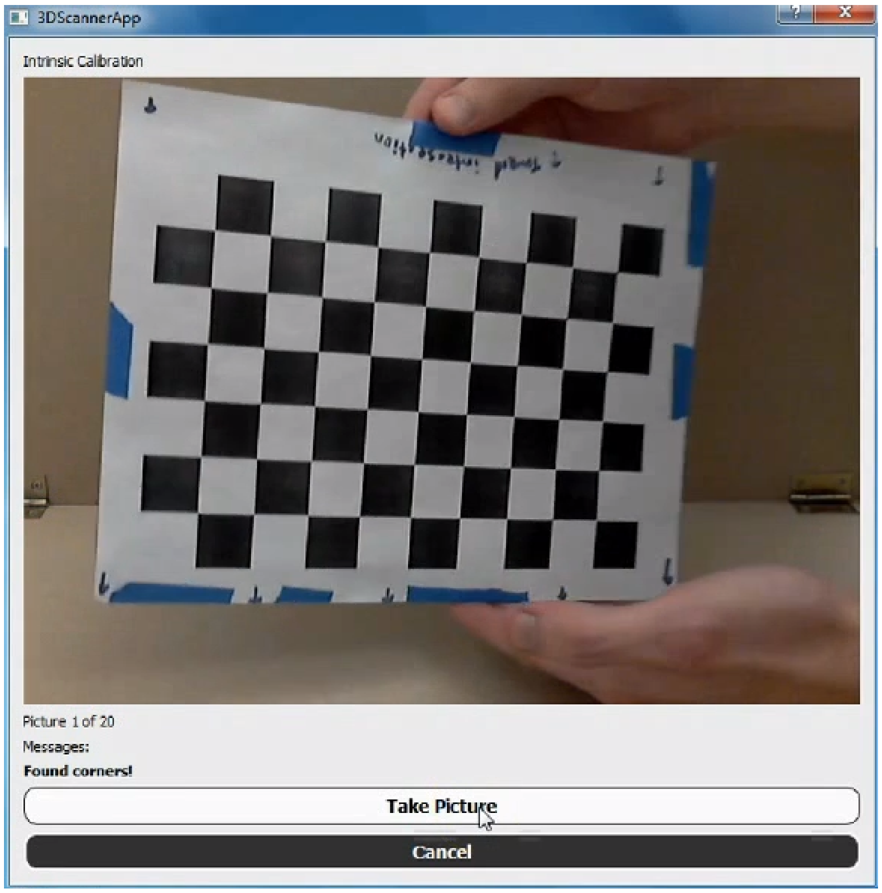
Horizontal Squares: 10

Vertical Squares: 7

Save Directory: Z:\

Messages:

Figure 6: Main calibration screen for intrinsic calibration input.


3DScannerApp

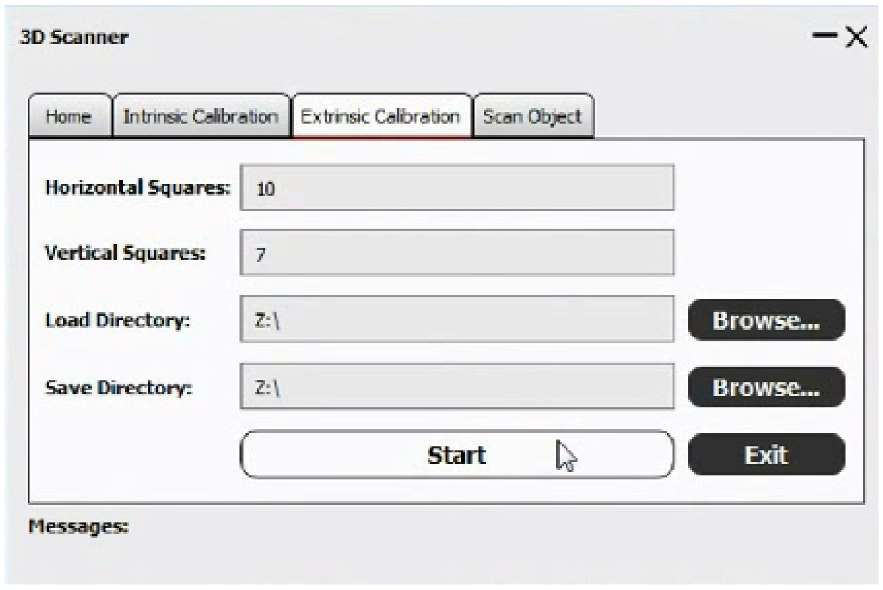
Intrinsic Calibration

Picture 1 of 20

Messages:

Found corners!

Figure 7: This is the screen the user will use to take a picture of the board for calibration.



Horizontal Squares: 10

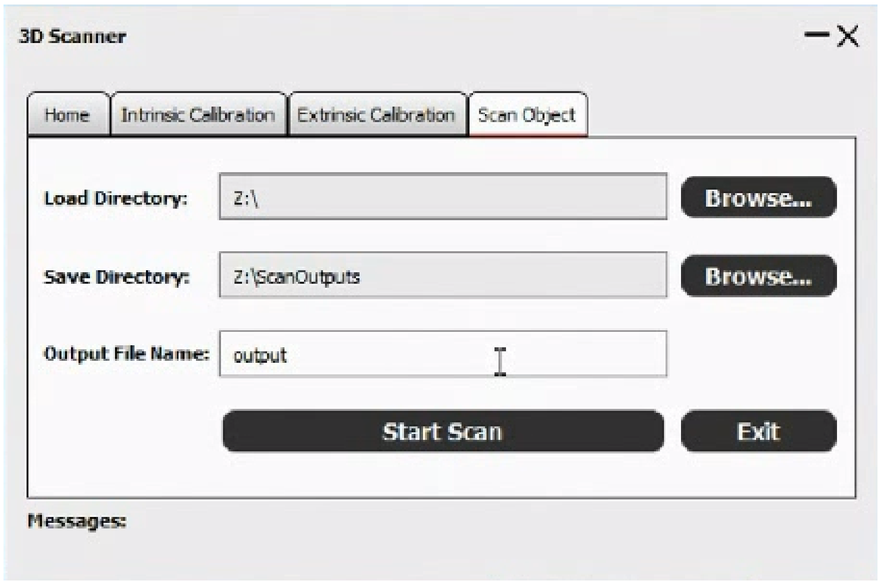
Vertical Squares: 7

Load Directory: Z:\

Save Directory: Z:\

Messages:

Figure 8: Main calibration screen for extrinsic calibration input.



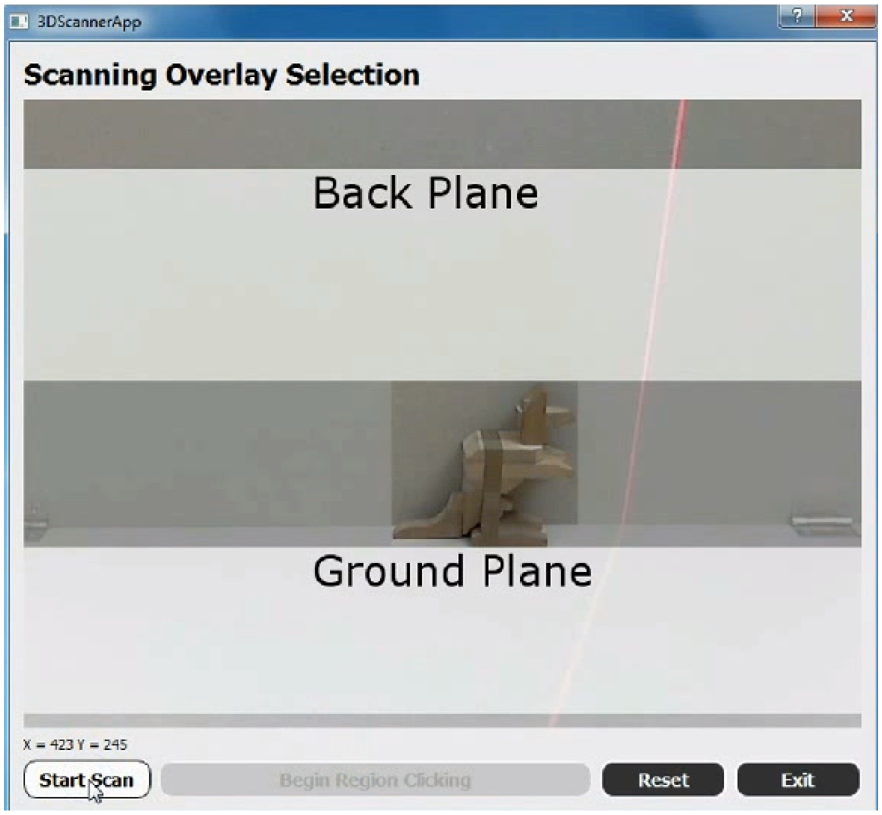
Load Directory: Z:\

Save Directory: Z:\ScanOutputs

Output File Name: output

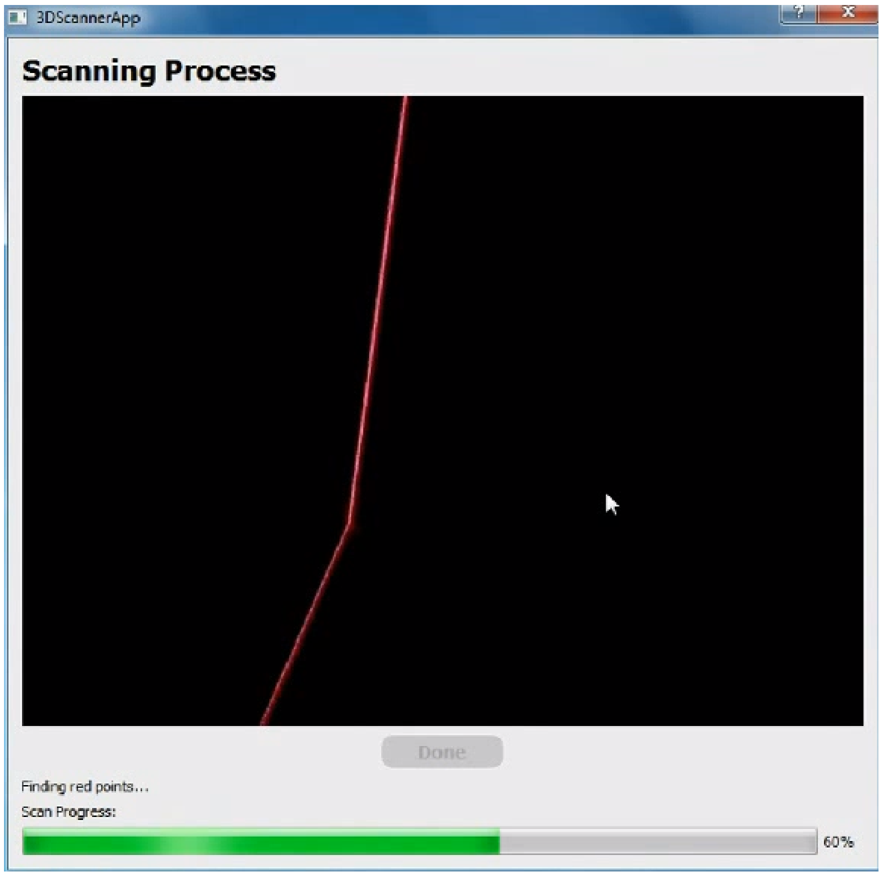
Messages:

Figure 9: Main screen for scan input.



X = 423 Y = 245

Figure 10: Screen for the user to define the various regions.



Finding red points...

Scan Progress:

60%

Figure 11: Screen showing scan.

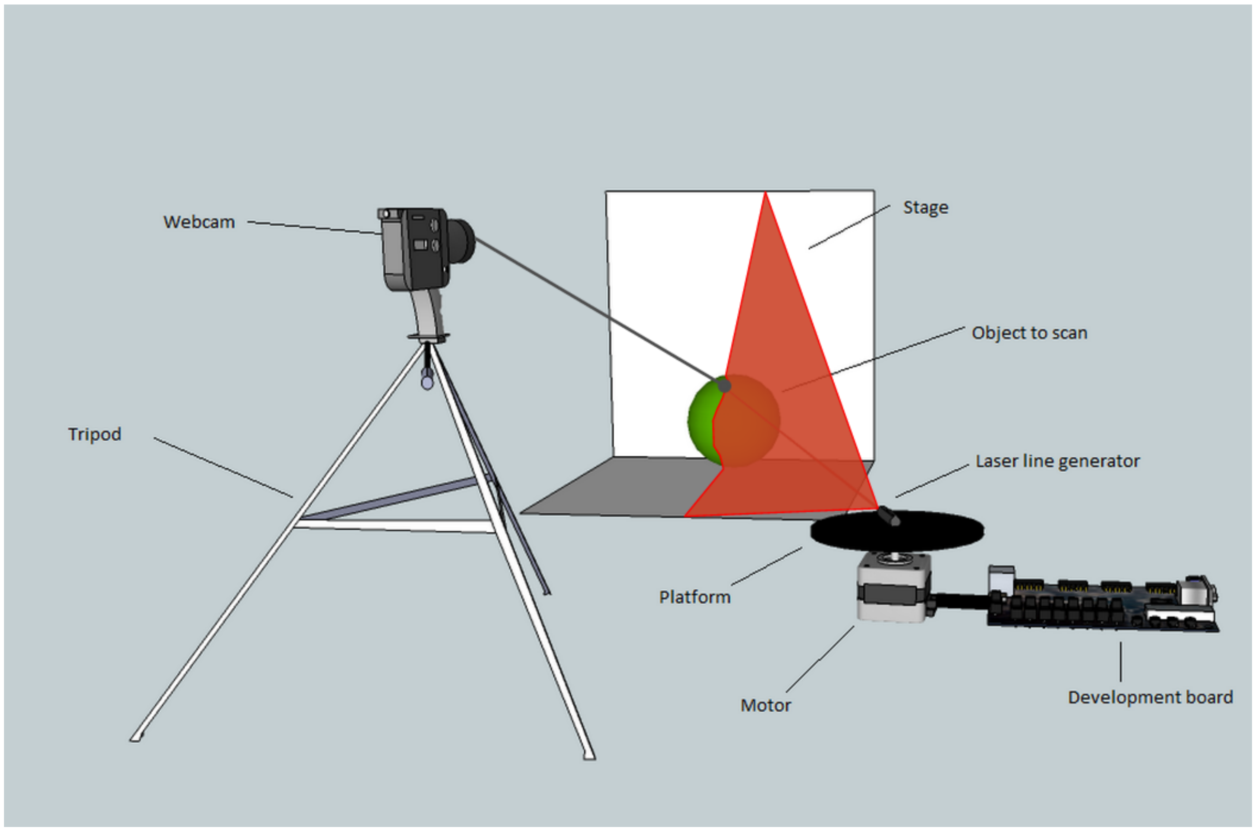


Figure 12: Sketch of what our setup will be.

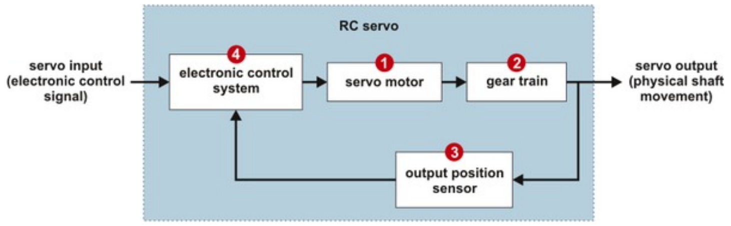

RC servo
servo input (electronic control signal)
4 electronic control system
1 servo motor
2 gear train
servo output (physical shaft movement)
3 output position sensor

Figure 13: Block diagram for servo motor design. Image taken from Pololu Robotics and Electronics Website [10].

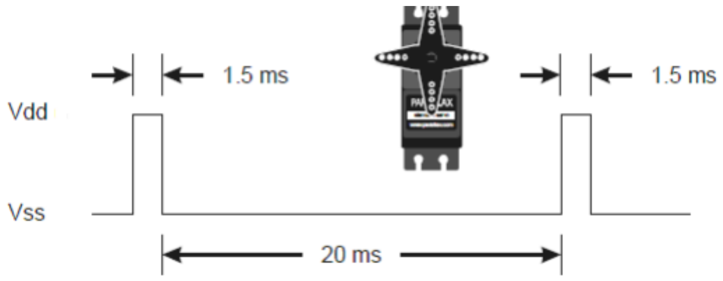


Figure 14: Pulse Width Modulation signal for stopped/”centered” servo. Image taken from Parallax manual for their continuous rotation servo [14].

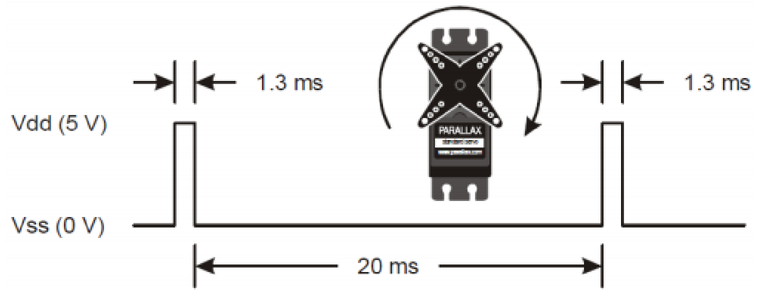


Figure 15: Pulse Width Modulation signal for clockwise rotating servo. Image taken from Parallax manual for their continuous rotation servo [14].

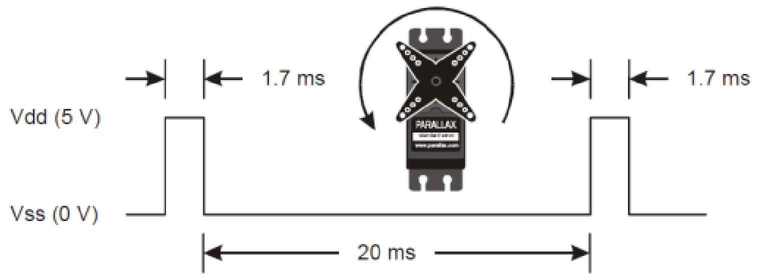


Figure 16: Pulse Width Modulation signal for counter-clockwise rotating servo. Image taken from Parallax manual for their continuous rotation servo [14].

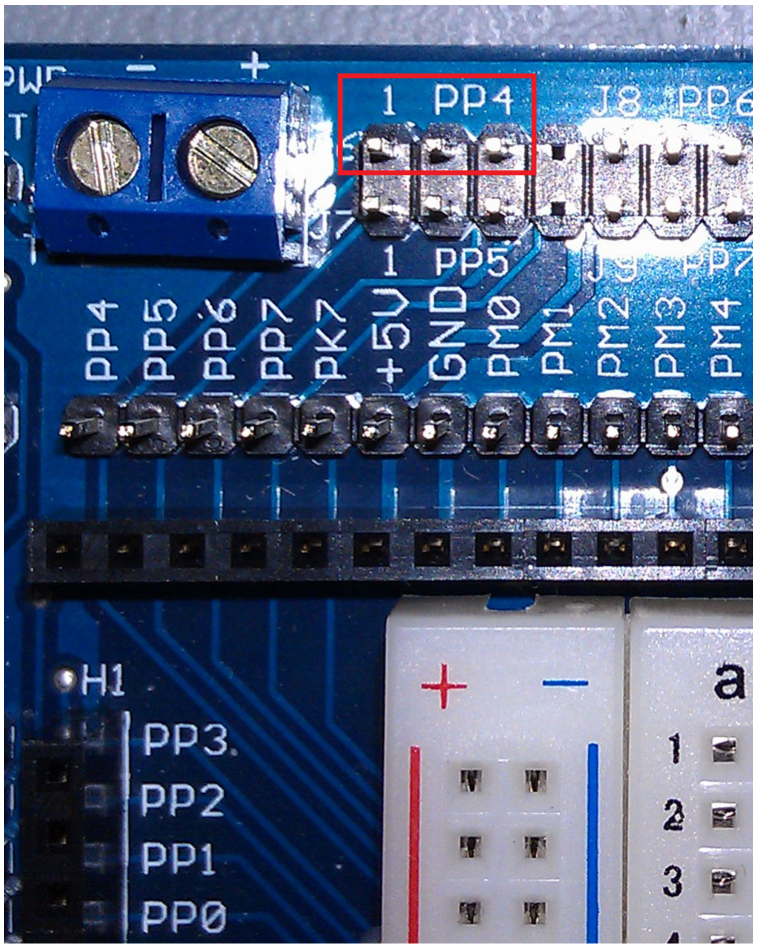
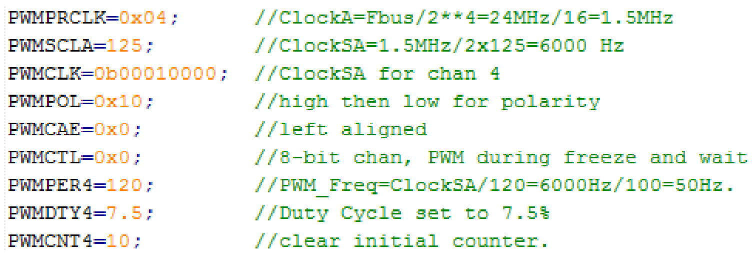
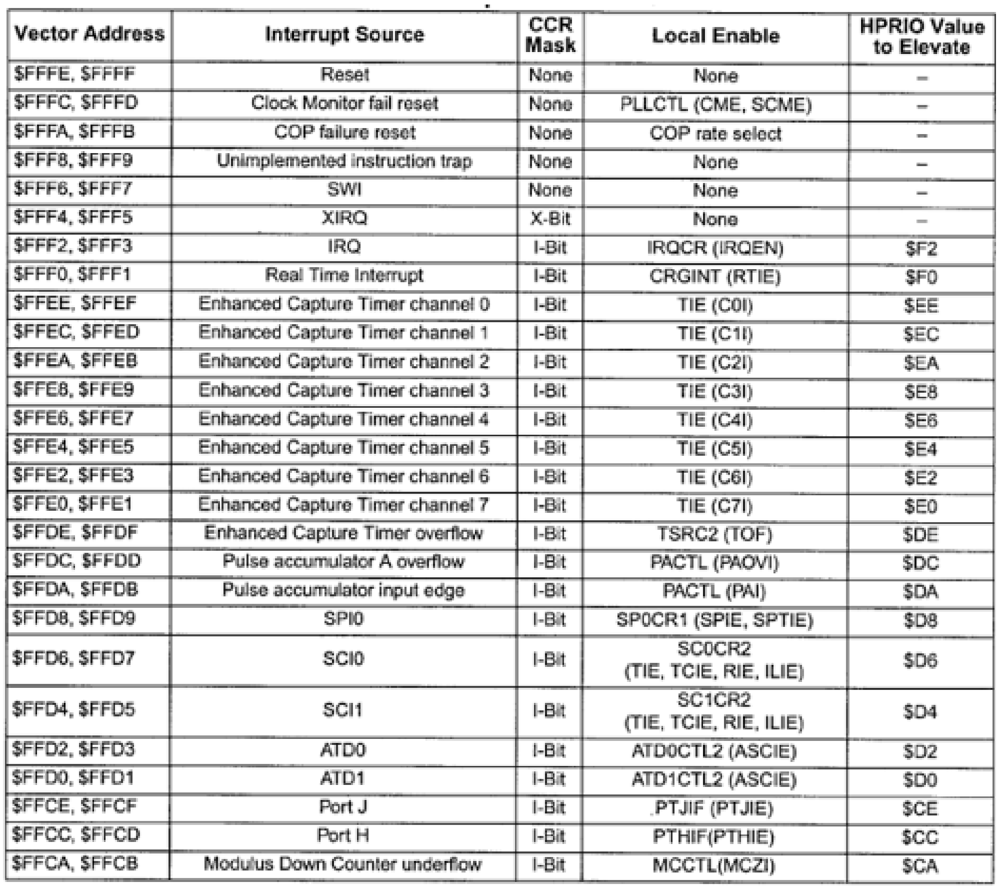


Figure 17: PP4 Pulse Width Modulation Hardware Interface on the Dragon12 Rev. F Microcontroller. The pulse width modulated control signal is the pin on the left, power supply voltage is the center pin and ground is the right side pin.



```
PWMPRCLK=0x04;       //ClockA=Fbus/2**4=24MHz/16=1.5MHz
PWMSCLA=125;         //ClockSA=1.5MHz/2x125=6000 Hz
PWMCLK=0b00010000;   //ClockSA for chan 4
PWMPOL=0x10;         //high then low for polarity
PWMCAE=0x0;          //left aligned
PWMCTL=0x0;          //8-bit chan, PWM during freeze and wait
PWMPER4=120;         //PWM_Freq=ClockSA/120=6000Hz/100=50Hz.
PWMDTY4=7.5;         //Duty Cycle set to 7.5%
PWMCNT4=10;          //clear initial counter.
```

Figure 18: Dragon12 registers and associated values used to set up a modulated pulse signal to send to the Parallax continuous rotation servo. This code was adapted to this application from code found on the MicroDigitalEd website [15].



Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
$FFFE, $FFFF	Reset	None	None	–
$FFFC, $FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
$FFFA, $FFFB	COP failure reset	None	COP rate select	–
$FFF8, $FFF9	Unimplemented instruction trap	None	None	–
$FFF6, $FFF7	SWI	None	None	–
$FFF4, $FFF5	XIRQ	X-Bit	None	–
$FFF2, $FFF3	IRQ	I-Bit	IRQCR (IRQEN)	$F2
$FFF0, $FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	$F0
$FFEE, $FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	$EE
$FFEC, $FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	$EC
$FFEA, $FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	$EA
$FFE8, $FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	$E8
$FFE6, $FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	$E6
$FFE4, $FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	$E4
$FFE2, $FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	$E2
$FFE0, $FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	$E0
$FFDE, $FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	$DE
$FFDC, $FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	$DC
$FFDA, $FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	$DA
$FFD8, $FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	$D8
$FFD6, $FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	$D6
$FFD4, $FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	$D4
$FFD2, $FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	$D2
$FFD0, $FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	$D0
$FFCE, $FFCF	Port J	I-Bit	PTJIF (PTJIE)	$CE
$FFCC, $FFCD	Port H	I-Bit	PTHIF(PTHIE)	$CC
$FFCA, $FFCB	Modulus Down Counter underflow	I-Bit	MCCTL(MCZI)	$CA

Figure 19: Interrupt Vector Locations for the Dragon12 (HCS12) Development Board. Table taken from "Microcontroller Theory and Applications" [11].

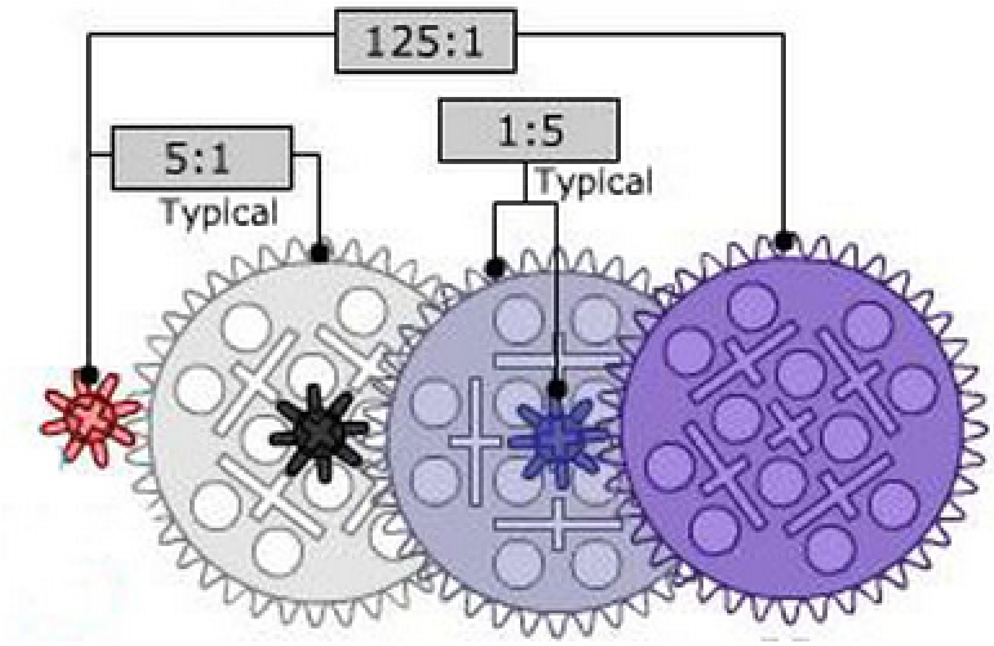


Figure 20: This figure shows an example of the construction of a reducing gear train [16].

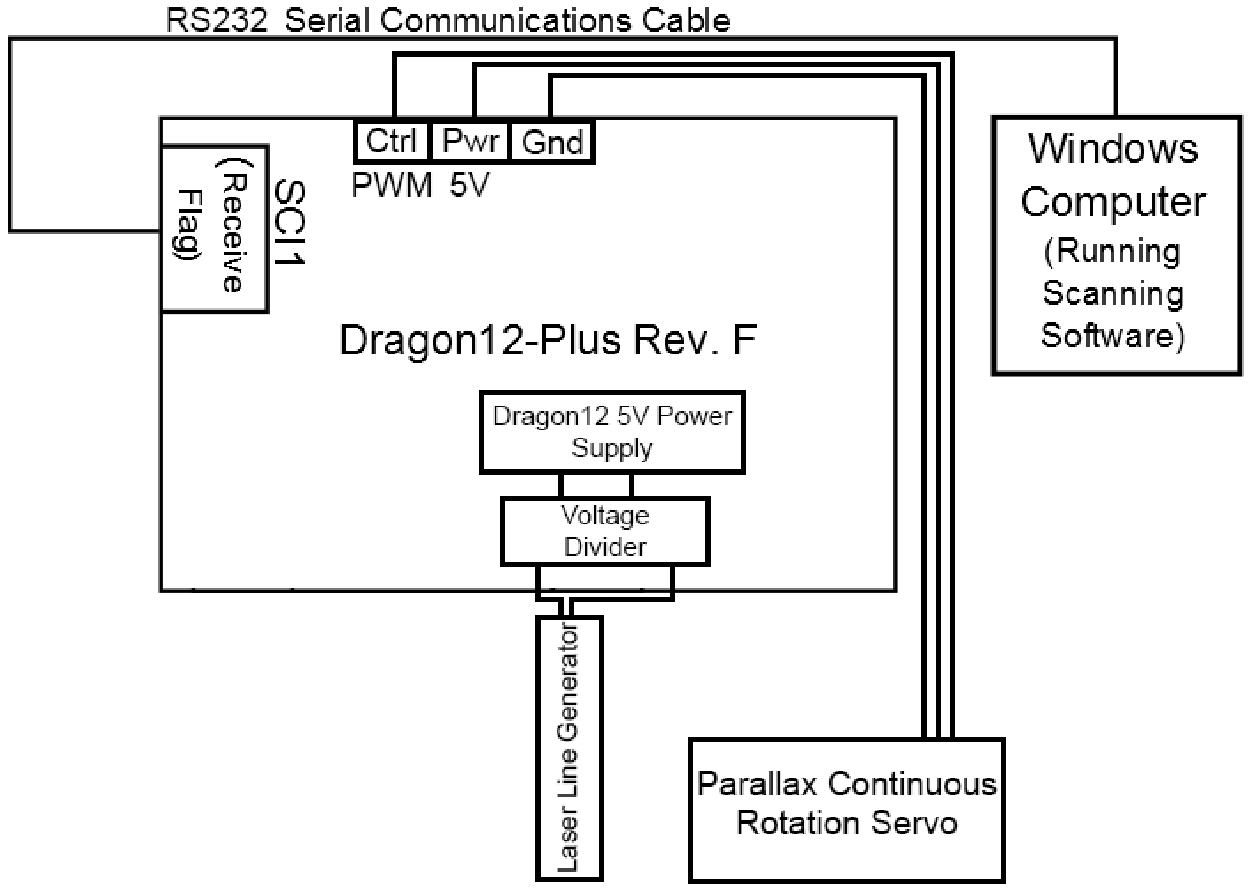


Figure 21: The block diagram of the hardware systems associated with the Dragon12 for the purpose of controlling the stepper motor.