Design Document:
Implement a 3-D Scanner Using Triangulation

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

Faculty Mentor: Dr. David Wolff

CSCE 499, Fall 2012

Contents

List of Figures

1 Introduction

Our project is to build a 3D Scanner that will scan a real world object and create a digital representation. This process will use mathematical triangulation, calculating intersections between a laser line plane and the camera's "rays". The math behind it uses primarily linear algebra and geometry techniques, such as matrix operations and vector-plane calculations [1]. In order for these calculations to accurately display a 3D scan, we must account for intrinsic parameters and extrinsic parameters of the camera [2, 3].

The process involves the following:

1. Calibrate a webcam to obtain the intrinsic and extrinsic parameters.
2. Build the scanning environment and integrate hardware pieces.
3. Develop code that will interact with the development board.
4. Develop the application for calibration, scanning, and output.
5. Create a polygonal mesh representation of the object using the point cloud.

For our project to be considered a success, we should be able to produce a point cloud, and hopefully a polygonal mesh representation, of a real-world object.

Please see our requirements document for more details.

2 Research Review

This section gives more detail to some of the research we have done related to implementation.

2.1 Camera Settings

OpenCV does not currently appear to support setting a variety of camera settings that we need automated for our project such as white levels, gain control, etc. In order to achieve this will need to use a C++ library called videoInput. This video library uses Microsoft Windows DirectDraw library (also known as VFW, or Video For Windows) to make it easier to set parameters for the camera. This is documented at [4] and can be downloaded at: [5].

2.2 Math

Here is a description of the mathematical procedure to generate the points for the point cloud of an object:

For each image...

1. **Determine Laser Planes**

   1.1. For each row of pixels in the "back plane" region:

   1.1.1. Find the image coordinates of the red component (where the laser hits the back plane) by obtaining where the red component is near its highest point in the pixel row. Store it in memory.

   1.1.2. Repeat for pixel rows in "ground plane" region.

   1.2. For the stored points in the "back plane" region:

   1.2.1. Convert the obtained image coordinates, which are in a real-life 2D image-camera coordinate system, to the ideal 3D image-camera coordinate system via undistortion (using the distortion parameters of the camera) and the inverse intrinsic matrix K^{-1} followed by converting to homogeneous coordinates. This will give us an ideal 3D camera coordinate of $P_C = \lambda u$, where u is the resulting idealized image coordinate and λ is some scalar.

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

   1.2.2.1. Represent the ray through the image point u found above using parametric form: $R = \{P_C = q_C + \lambda u\}$. Since we are working in the ideal camera system, we can assume q_C is the origin of the camera system, $(0, 0, 0)$.

   1.2.2.2. Pick a point P_{BW} on the back plane, namely $(0, 0, 0)$, and convert it to camera coordinates using the equation $P_C = R_B P_{BW} + T_B$, where R_B and T_B are the extrinsic parameter rotation and translation matrices respectively. The resulting point will be q_P in following steps.

   1.2.2.3. Assuming the back plane is the xz-plane in its coordinate system, the normal vector is $(0, 1, 0)$. Convert this to camera coordinates as well. This will be n in the steps below.

   1.2.2.4. Any line will intersect a non-parallel plane at one point. So using the implicit representation of a plane, the point of intersection p is where $n^T(p-q) = 0$. The point p is in the ray we defined above, so the equation becomes:

   $$n^T(q_C + \lambda u - q_P) = 0.$$

   Then solving for λ:

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

   1.2.3. Use each λ to obtain the associated camera coordinate P_C of the back plane using $P_C = \lambda u$ and store it in memory.

1.2.4. After all points have been processed, find the best fit line of all P_C's and store it in memory. This represents the laser line across the back plane in camera coordinates.

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

1.4. Perform a least squares approximation of the intersection point between the two lines. Store it along with the normal vector (cross product of the vector components of the 2 lines) in memory. The approximate intersection will be used as the point on the plane, along with the normal vector to the plane, for the ray-plane intersection in the object region.

2. **Object/Laser Intersection**

2.1. For each row of pixels in the "object" region:

2.1.1. Find the image coordinates of the red component (where the laser hits the object) by obtaining where the red component is near its highest point in the pixel row.

2.1.2. Convert the obtained image coordinates, which are in a real-life 2D image-camera coordinate system, to the ideal 3D image-camera coordinate system via undistortion (using the distortion parameters of the camera) and the inverse intrinsic matrix K^{-1} followed by converting to homogeneous coordinates. This will give us an ideal 3D camera coordinate of $P_C = \lambda u$, where u is the resulting idealized image coordinate and λ is some scalar.

2.1.3. Solve for λ using a ray-plane intersection.

2.1.3.1. Represent the ray through the image point u found above using parametric form: $R = \{P_C = q_C + \lambda u\}$. Since we are working in the ideal camera system, we can assume q_C is the origin of the camera system, $(0, 0, 0)$.

2.1.3.2. Any line will intersect a non-parallel plane at one point. So using the implicit representation of a plane, the point of intersection p is where $n^T(p-q) = 0$. The point p is in the ray we defined above, so the equation becomes:

$$n^T(q_C + \lambda u - q_P) = 0.$$

Then solving for λ:

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

We let q_C (a point in the laser plane) in this case be the approximate intersection of l_G and l_B found above.

2.1.4. Use λ to obtain the camera coordinate P_C of the object/laser plane intersection using $P_C = \lambda u$.

2.1.5. Convert P_C to either the back plane or ground plane world coordinate system using the respective $P_W = R^{-1}P_C - R^{-1}T = R^T P_C - R^T T$ equation.

2.1.6. Store P_W, along with the number of the current image, in memory.

3 Design Methodology

This section describes part of the design we will use for implementation.

3.1 Software

This section describes the software portion of the project. Due to page constraints, we could not include all of our diagrams, including sequence diagrams for calibration and scanning, as well state diagrams for intrinsic and extrinsic calibration and the interrupt.

3.1.1 State Diagram: Scanning

The state diagram for scanning can be found in Appendix A at Figure 1.

3.1.2 Class Design

The class diagram showing our object-oriented, MVC design can be found in Appendix A at Figure 2 and 3.

3.1.3 GUI Sketches

Updated GUI designs can be found at Figures 4-9.

3.1.4 Testing Plan

Our project will consist of ongoing testing throughout the projects various phases. This will be to ensure that all data and application behaviors are working properly throughout each phase. Because our project is predominantly concerned with the accuracy of data points of a scanned object, we will need to ensure that our data precisely represents the object. In order to test this, we will ensure the following items are tested in their corresponding stages:

- Calibration (Intrinsic and Extrinsic parameters):
    - Accuracy and precision of data
    - Processing images for calibration works as intended
- Scanning Process
    - Scanning Speed to ensure optimal data is collected

    - Accuracy of point data collected (including time-data of each point)
    - Best-Fit Line is determined correctly
    - Correct calculations (using our pinhole camera model and related mathematics)
    - Processing points in the final stages
    - Polygonal Meshing accurately represents object
- User Interactions
    - Correctly produce messages to user, ensuring complete understanding of the scanning process.
    - Making sure the XML files that are loaded contain correct data, and that they are stored correctly.
    - Any user input will be correctly handled
- Miscellaneous Testing
    - Ensure views and models are getting updated correctly

Each of the developers will be responsible for testing and determining if the test results are acceptable. Testing frameworks may be used to correctly assess collected data, where applicable. Once testing has been completed for each segment, we will then proceed on with the next phase of development. For example, we will need to ensure that calibration is working correctly before moving on to the scanning segment of the project.

3.2 Hardware

This section describes the hardware portion of the project.

3.2.1 Description of Functional Modules

1. Power Supply
    1.1. The external power supply must be capable of delivering between 18V-20V and around 1A of current. This power supply will be used to provide external power for the Stepper Motor via the Terminal Block 4 Vext power connection on the Dragon12 Development Board. The Dragon12 5Voutput will likely be used to provide power to the laser line generator.
2. Stepper Motor
    2.1. The stepper motor being used is an Applied Motion Products HT23-396 motor, which is rated at 7.2V and 1A. It is a 2-phase hybrid style stepper motor wired in a bipolar series configuration. The bipolar nature of the motor means that it requires an H-Bridge to allow for the reversal of current through the motor

coils. The 2-phase construction of the motor also means that 4 Half-H-Bridges are necessary to drive the motor. The 1A current rating as well as the 2-phase, bipolar nature of this stepper motor allows it to be paired nicely with the Quadruple Half-H-Bridge driver on the Dragon12, which is also rated at 1A.

3. Dragon12-Plus Revision F Development Board

   3.1. The Dragon12 is a Freescale HCS12 based development board that provides a host of different microcontroller and other educational functionalities. In this particular project the integrated stepper motor controller functionality is being paired with the onboard SN754410 Quadruple Half-H Driver and the ability to program interrupts in order to drive and control the stepper motor. The ability to connect an external power supply to the Dragon12 is being utilized to overcome the current and voltage limitations of the provided 9V Dragon12 power supply. Finally, one of the two provided serial communications interfaces on the Dragon12 is being employed to provide communication in the form of interrupt signals between the Windows computer, running the scanning/calibration software, and the Dragon12. This communication is necessary for the purposes of implementing interrupt functionality on the Dragon12. This allows the precompiled program on the development board to be, in effect, started and stopped as desired based on signals sent from the scanning/calibration software running on the Windows computer.

4. Desktop Computer running Windows Operating System

   4.1. This is a basic Dell desktop computer. This machine has a serial communications interface port used to compile the stepper motor control program onto the Dragon12 development board. This computer will run the scanning/calibration software that sends the interrupt signal via serial connection thus allowing the stepper motor control program to run.

3.2.2 Theory of Hardware Operation

2-Phase Bipolar Hybrid Stepper Motor

The following information along with more detailed descriptions can be found at the NMB Technology Corporation's Website [6]. A stepper motor is a form of DC motor that relies on magnetism to advance a rotor in increments known as steps. A hybrid stepper motor has been selected for this project. In a hybrid motor, the rotor contains an axially magnetized permanent magnet that is encompassed by two rotor cups, which have teeth notched around their circumference (see Figure 10). The teeth of the two rotor cups are offset to each other by a specified number of degrees that is dependent upon what step resolution the motor has. The motor housing (i.e. stator) that surrounds the rotor contains two phases, each comprised of 4 windings with their own set of teeth for a total of 8 windings between the two phases. Each winding creates a magnetic field according to which direction current is passing through the winding.

For each phase, the 4 windings are placed at 90 degree angles to each other. The windings that are 180 degrees to each other are the same magnetic polarity and the windings that are 90 degrees from each other are opposite magnetic polarities (see Figure 11). When energized, the windings in a particular phase that are of the same polarity will attract the teeth on one end of the axially magnetized rotor while the other set of windings will attract the teeth on the opposite end of the rotor. This in combination with the fact that the two sets of teeth on the rotor are offset from each other causes the rotor to rotate (see Figure 12). In this project, the stepper motor controller functionality built into the Dragon12-Plus development board is used to control the phase sequencing (see Figure 12).

Another aspect of the stepper motor that can be controlled is the choice between a bipolar parallel, bipolar series, or unipolar winding wiring scheme. This choice affects the amount of current required to run the stepper motor, the type of stepper motor controller and driver required, and the torque characteristics of the motor. In this case, a bipolar parallel configuration was chosen because it requires less current than bipolar series. The torque profile of each wiring configuration is not a concern for this project because we are only rotating a very light and small laser line generator.

H-Bridge

The bipolar configuration allows the motor to generate current flow in either direction in each winding. This means that a driver circuit is needed that can switch the direction of the current flowing through the windings. Driving a 2-phase bipolar stepper motor with 4 pairs of windings, where each pair has the same polarity when energized, can be achieved using 2 H-bridge circuits or in the case of this project, 4 half bridge circuits (see Figure 13). Each half bridge circuit is composed of half of a full h-bridge circuit as the name suggests, which means that only two transistor switches on one side of the load are used to reverse the current. The quadruple half bridge integrated circuit (SN754410) that is integrated into the Dragon12 development board serves the purpose of driving each of the pairs of windings.

Dragon12 Stepper Motor Phase Sequence Control

The half bridges are controlled using the stepper motor controller functionality on the Dragon12. The phase sequencing can then be programmed in C and compiled onto the board. An example program for stepper motor control from which the code in Figure 14 was derived can be found on the MicroDigitalEd website [7]. This phase sequence is specific to any given stepper motor. Example phase sequencing code for the stepper motor used in this project can be seen below in the form of an infinite for loop. PORTB in this code is defined to be the M1-M4 voltage output connections on the Dragon12.

Interrupts

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" [8] [9]. The Dragon12 is simply a dumb processor. 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 case, the ISR will contain the code to rotate the stepper motor in the manner necessary to scan the object. After the ISR is finished, the Dragon12 returns to where the original program was halted and continues execution of the original program. 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 are defined in Table 15. In this case, the SCI1 interrupt source is linked with the ISR containing the necessary phase sequencing code to rotate the stepper motor rotor. It is also necessary to enable interrupts on the board as a whole (Clear Interrupt Mask instruction), as well as for the specific interrupt to the used (SCI1).

When an interrupt event occurs, the following takes place:

1. The current instruction execution is completed.
   1.1. In this project, the original program is simply a wait state defined by an infinite for loop.
2. instruction queue is cleared.
3. The return address is calculated.
4. Key registers associated with the current program execution, most notably the Program Counter, are pushed to the stack.
5. The interrupt ('I') bit in the Conditional Code Register (CCR) is set to disable all other interrupts.
6. The interrupt vector (in this case the vector for SCI1) is fetched.
7. Control over the program is transferred to the ISR.
8. The interrupt source is cleared (programmer must do this).
9. The programmer's code in the ISR is executed.
   9.1. In this project, a bit will be sent back to the windows machine when the stepper motor scan routine finishes executing, indicating to the scanning application on the windows machine that the stepper motor has finished the hardware side of the scanning process.
10. The key register values for the original program are popped off the stack.
11. The 'I' bit of the CCR is cleared to re-enable all interrupts.
12. Program control is returned to the original program (RTI instruction).

13. The original program continues execution from where it was interrupted.

    13.1. In this project this is simply a return to the wait state.

3.2.3 Block Diagram

A block diagram of the hardware modules associated with driving the stepper motor, which is composed of the Dragon12 development board, stepper motor, power supply and computer running windows can be seen in Figure 16.

4 Work Completed

This section describes some of the research and tests we have performed.

4.1 Red Plot settings

In order to obtain the laser line from the image, it is necessary to determine the location of the red component in a row of pixels. We have tested this out using our laser line in a dark room for a specific row of pixels (halfway down the image). At first, we were using the intensity value from an HSV image (Hue, Saturation, Value of Intensity). This produced fair results. However, using HSV is unnecessary. The image is first captured as a BGR (Blue Green Red) image, so we can just get the red component from each pixel instead. We did so and plotted the results, which can be found below. This is not on the highest resolution of the camera. Future work will include higher resolution and determining what point to use as the intersection point of the laser and object.

4.2 Undistort

One of our questions when walking through the math is what the z-component for an image coordinate obtained from the actual image would be. The undistort method from OpenCV handles converting a distorted image coordinate to the ideal image coordinate. To determine how the undistort method would look for an obtained image point, we tested it out using the calibration code provided by OpenCV. We calibrated provided chessboard images to obtain the intrinsic and distortion parameters and then picked one image that was fairly straight on. We found its corners and converted them from the obtained image coordinates to the ideal coordinates and plotted them in a scatter plot as well as mapped the points directly on the board. The mapped points on the board were undetectable from each other and the scatter plot showed a good correlation between distorted and undistorted as well.

4.3 Motor/Circuitry

We have successfully implemented a spinning motor; interrupts are in the prototype stage. We hope to have the interrupts done in January.

5 Future Work

Plans for our future work include:

- Determining red intersection point technique (Jeff)
- GUI interaction between C++ and OpenCV (Tyler)
- Constructing the scanning stage (Jeff, Tyler, Grady)
- Build C program to control motor/laser (Grady)
- Develop Basic Scanning Functionality (Jeff, Tyler, Grady)
- Testing major deliverables (Calibration, Scanning, etc) (Tyler)
- Final document and presentation (Jeff, Tyler, Grady)

6 Updated Timetable

We have updated our timetable, which can be found in Appendix A at Figure 17; our previous version was quite ambitious. While we may not actually get to merging point clouds, we have left it in the diagram in case we are able to successfully get that far.

Glossary

extrinsic parameters The translation and rotation vectors relating the stage's "world coordinates" to "camera coordinates".

H-bridge A circuit that provides functionality to reverse DC current direction through a load using switches.

intrinsic parameters A model of the camera's geometry and a distortion model of the camera's lens [2, 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] D. Lanman and G. Taubin, “Build your own 3d scanner: Optical triangulation for beginners.” `http://mesh.brown.edu/byo3d/index.html`, 2012.

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

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

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

[5] “videoInput - a free windows video capture library.” `http://www.muonics.net/school/spring05/videoInput/`, 2009.

[6] N. T. Corporation, “The permanent magnet in motors construction and operating theory: Hybrid motor.” `http://www.nmbtc.com/step-motors/engineering/construction-and-operating-theory.html`.

[7] M. Mazidi, “Stepper motor control using the h-bridge on dragon12.” `http://www.microdigitaled.com/HCS12/Hardware/Dragon12/CodeWarrior/Stepper_Motor_Control_H_Bridge_Cprog.txt`.

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

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

[10] Orientalmotor, “Basics of motion control.” `http://www.orientalmotor.com/technology/articles/2phase-v-5phase.html`.

[11] “C++ documentation.” `http://www.cplusplus.com`, 2012.

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

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

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

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

[16] “3d digital corp applications: Architecture.” `http://www.3ddigitalcorp.com/applications/architecture`, 2012.

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

A Appendix A

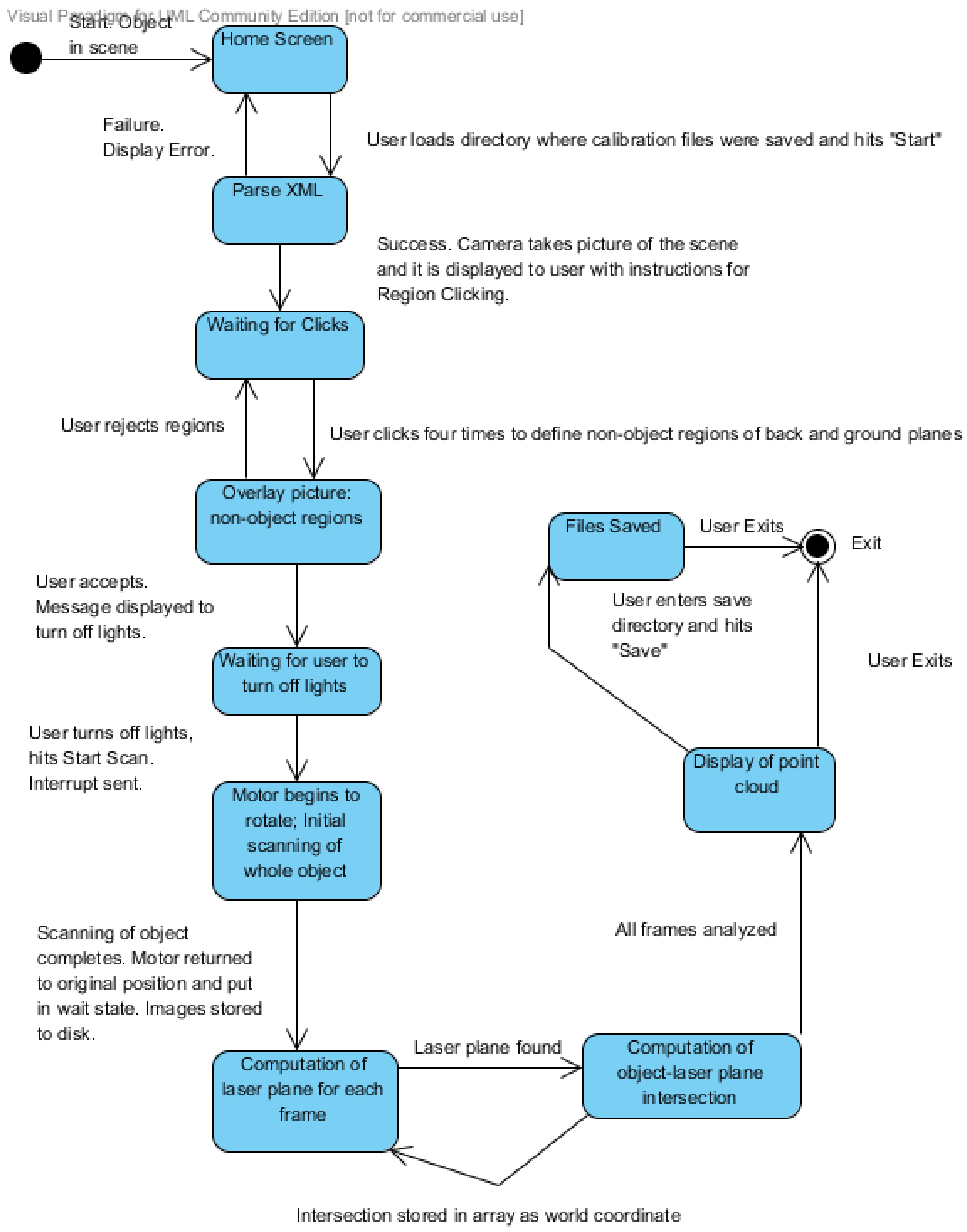

Visual Paradigm for UML Community Edition [not for commercial use]
Start: Object in scene
Home Screen
Failure. Display Error.
User loads directory where calibration files were saved and hits "Start"
Parse XML
Success. Camera takes picture of the scene and it is displayed to user with instructions for Region Clicking.
Waiting for Clicks
User rejects regions
User clicks four times to define non-object regions of back and ground planes
Overlay picture: non-object regions
User accepts. Message displayed to turn off lights.
Waiting for user to turn off lights
User turns off lights, hits Start Scan. Interrupt sent.
Motor begins to rotate; Initial scanning of whole object
Scanning of object completes. Motor returned to original position and put in wait state. Images stored to disk.
Computation of laser plane for each frame
Laser plane found
Computation of object-laser plane intersection
Intersection stored in array as world coordinate
All frames analyzed
Display of point cloud
User enters save directory and hits "Save"
Files Saved
User Exits
Exit
User Exits

Figure 1: This shows the state transitions we will use for the scanning phase.

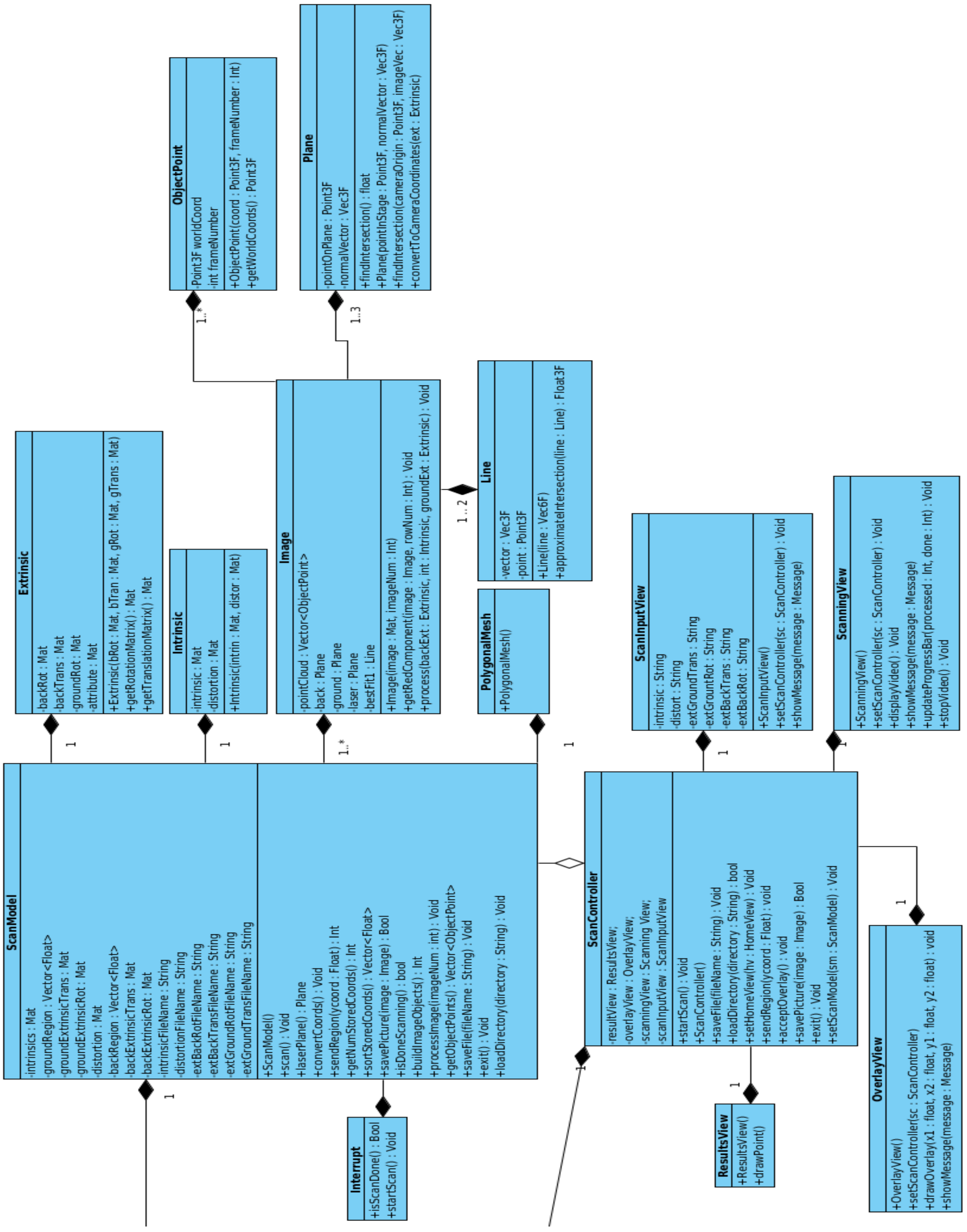

ScanModel
-intrinsics : Mat
-groundRegion : Vector<Float>
-groundExtrinsicTrans : Mat
-groundExtrinsicRot : Mat
-distortion : Mat
-backRegion : Vector<Float>
-backExtrinsicTrans : Mat
-backExtrinsicRot : Mat
-intrinsicFileName : String
-distortionFileName : String
-extBackRotFileName : String
-extBackTransFileName : String
-extGroundRotFileName : String
-extGroundTransFileName : String
+ScanModel()
+scan() : Void
+laserPlane() : Plane
+convertCoords() : Void
+sendRegion(ycoord : Float) : Int
+getNumStoredCoords() : Int
+sortStoredCoords() : Vector<Float>
+savePicture(image : Image) : Bool
+isDoneScanning() : bool
+buildImageObjects() : Int
+processImage(imageNum : int) : Void
+getObjectPoints() : Vector<ObjectPoint>
+saveFile(fileName : String) : Void
+exit() : Void
+loadDirectory(directory : String) : Void
Extrinsic
-backRot : Mat
-backTrans : Mat
-groundRot : Mat
-attribute : Mat
+Extrinsic(bRot : Mat, bTran : Mat, gRot : Mat, gTrans : Mat)
+getRotationMatrix() : Mat
+getTranslationMatrix() : Mat
Intrinsic
-intrinsic : Mat
-distortion : Mat
+Intrinsic(intrin : Mat, distor : Mat)
Image
-pointCloud : Vector<ObjectPoint>
-back : Plane
-ground : Plane
-laser : Plane
-bestFit1 : Line
+Image(image : Mat, imageNum : Int)
+getRedComponent(image : Image, rowNum : Int) : Void
+process(backExt : Extrinsic, int : Intrinsic, groundExt : Extrinsic) : Void
ObjectPoint
-Point3F worldCoord
-int frameNumber
+ObjectPoint(coord : Point3F, frameNumber : Int)
+getWorldCoords() : Point3F
Plane
-pointOnPlane : Point3F
-normalVector : Vec3F
+findIntersection() : float
+Plane(pointInStage : Point3F, normalVector : Vec3F)
+findIntersection(cameraOrigin : Point3F, imageVec : Vec3F)
+convertToCameraCoordinates(ext : Extrinsic)
Line
-vector : Vec3F
-point : Point3F
+Line(line : Vec6F)
+approximateIntersection(line : Line) : Float3F
PolygonalMesh
+PolygonalMesh()
Interrupt
+isScanDone() : Bool
+startScan() : Void
ScanController
-resultView : ResultsView;
-overlayView : OverlayView;
-scanningView : Scanning View;
-scanInputView : ScanInputView
+startScan() : Void
+ScanController()
+saveFile(fileName : String) : Void
+loadDirectory(directory : String) : bool
+setHomeView(hv : HomeView) : Void
+sendRegion(ycoord : Float) : void
+acceptOverlay() : void
+savePicture(image : Image) : Bool
+exit() : Void
+setScanModel(sm : ScanModel) : Void
ScanInputView
-intrinsic : String
-distort : String
-extGroundTrans : String
-extGrountRot : String
-extBackTrans : String
-extBackRot : String
+ScanInputView()
+setScanController(sc : ScanController) : Void
+showMessage(message : Message)
ScanningView
+ScanningView()
+setScanController(sc : ScanController) : Void
+displayVideo() : Void
+showMessage(message : Message)
+updateProgressBar(processed : Int, done : Int) : Void
+stopVideo() : Void
ResultsView
+ResultsView()
+drawPoint()
OverlayView
+OverlayView()
+setScanController(sc : ScanController)
+drawOverlay(x1 : float, x2 : float, y1 : float, y2 : float) : void
+showMessage(message : Message)
1
1
1
1..*
1
1..*
1..3
1 .. 2
1
1
1
1
1
1

Figure 2: The scanning portion of our class design.

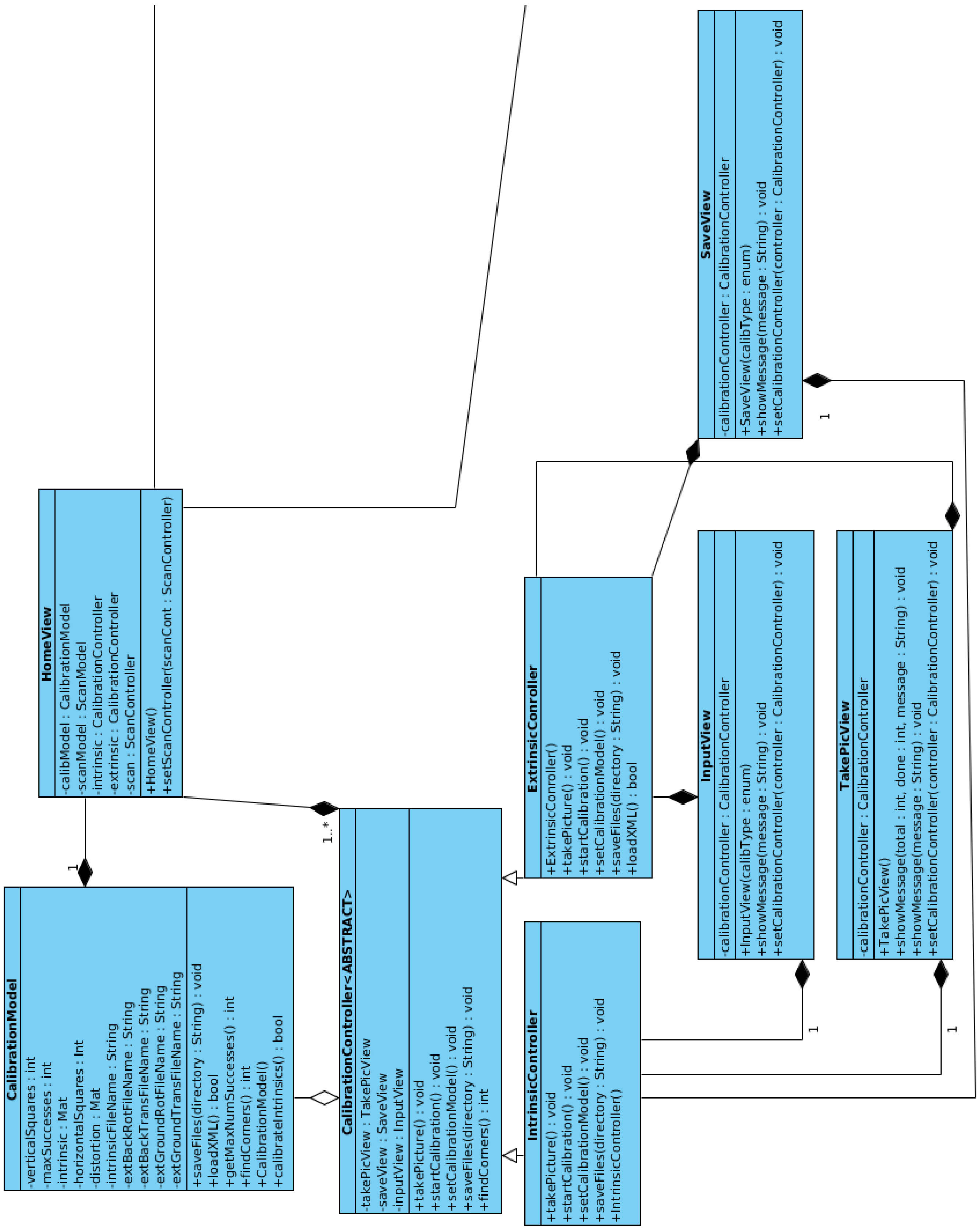

CalibrationModel
-verticalSquares : int
-maxSuccesses : int
-intrinsic : Mat
-horizontalSquares : Int
-distortion : Mat
-intrinsicFileName : String
-extBackRotFileName : String
-extBackTransFileName : String
-extGroundRotFileName : String
-extGroundTransFileName : String
+saveFiles(directory : String) : void
+loadXML() : bool
+getMaxNumSuccesses() : int
+findCorners() : int
+CalibrationModel()
+calibrateIntrinsics() : bool
HomeView
-calibModel : CalibrationModel
-scanModel : ScanModel
-intrinsic : CalibrationController
-extrinsic : CalibrationController
-scan : ScanController
+HomeView()
+setScanController(scanCont : ScanController)
CalibrationController<ABSTRACT>
-takePicView : TakePicView
-saveView : SaveView
-inputView : InputView
+takePicture() : void
+startCalibration() : void
+setCalibrationModel() : void
+saveFiles(directory : String) : void
+findCorners() : int
ExtrinsicConroller
+ExtrinsicConroller()
+takePicture() : void
+startCalibration() : void
+setCalibrationModel() : void
+saveFiles(directory : String) : void
+loadXML() : bool
IntrinsicController
+takePicture() : void
+startCalibration() : void
+setCalibrationModel() : void
+saveFiles(directory : String) : void
+IntrinsicController()
InputView
-calibrationController : CalibrationController
+InputView(calibType : enum)
+showMessage(message : String) : void
+setCalibrationController(controller : CalibrationController) : void
TakePicView
-calibrationController : CalibrationController
+TakePicView()
+showMessage(total : int, done : int, message : String) : void
+showMessage(message : String) : void
+setCalibrationController(controller : CalibrationController) : void
SaveView
-calibrationController : CalibrationController
+SaveView(calibType : enum)
+showMessage(message : String) : void
+setCalibrationController(controller : CalibrationController) : void
1
1..*
1
1
1

Figure 3: The calibration portion of our class design.

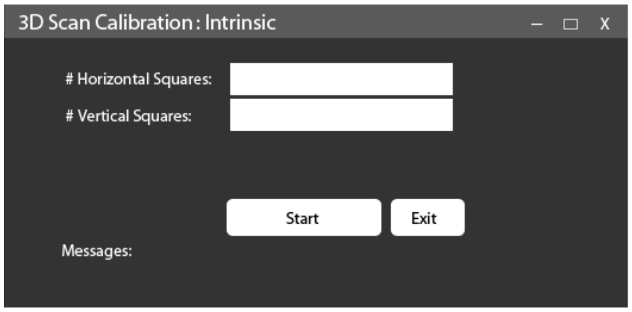


Figure 4: Main calibration screen for intrinsic calibration.

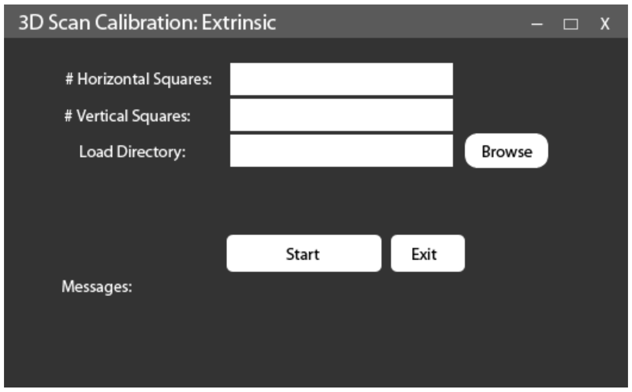


Figure 5: Main calibration screen for extrinsic parameters.

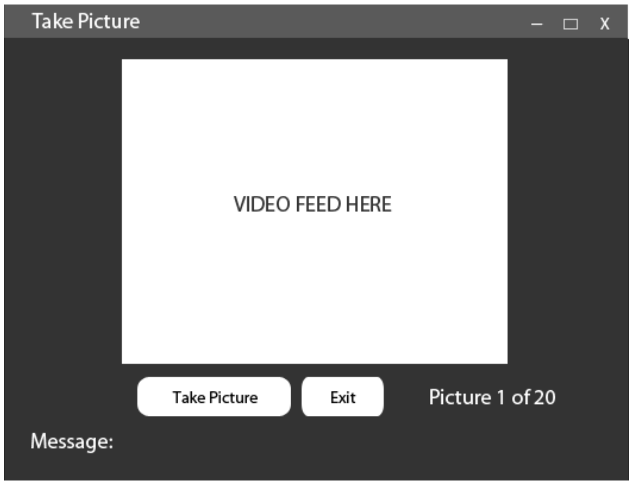


Figure 6: This is the screen the user will use to take a picture of the board.

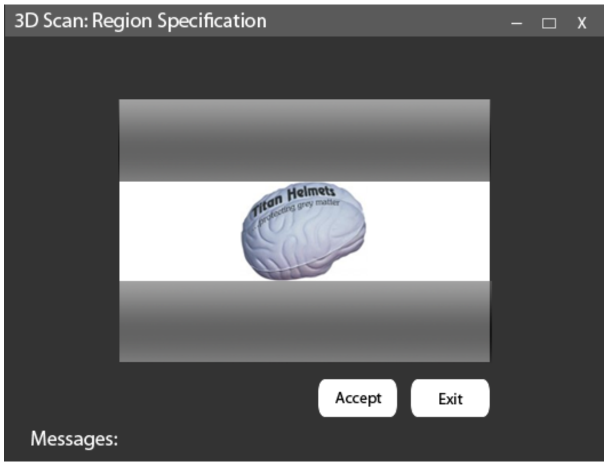


Figure 7: Screen for the user to define the non-object regions.

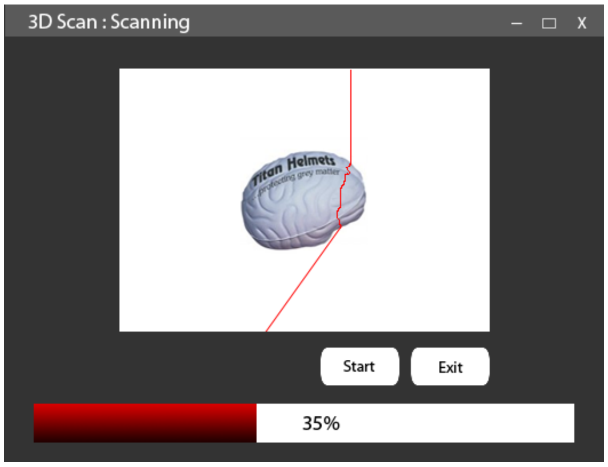


Figure 8: Screen showing scan-in-progress.

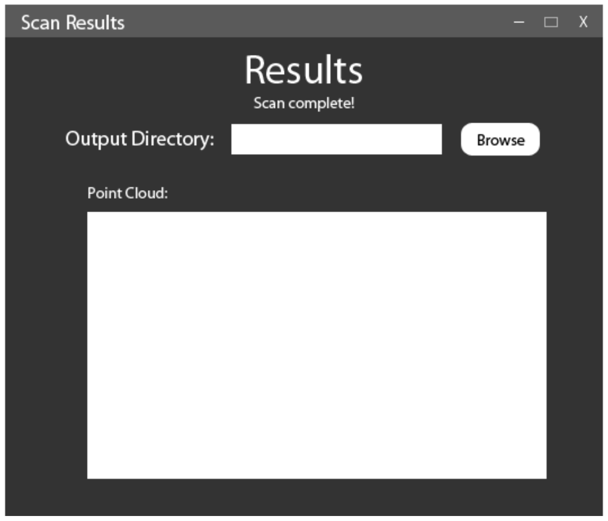


Figure 9: Screen showing the resulting point cloud.

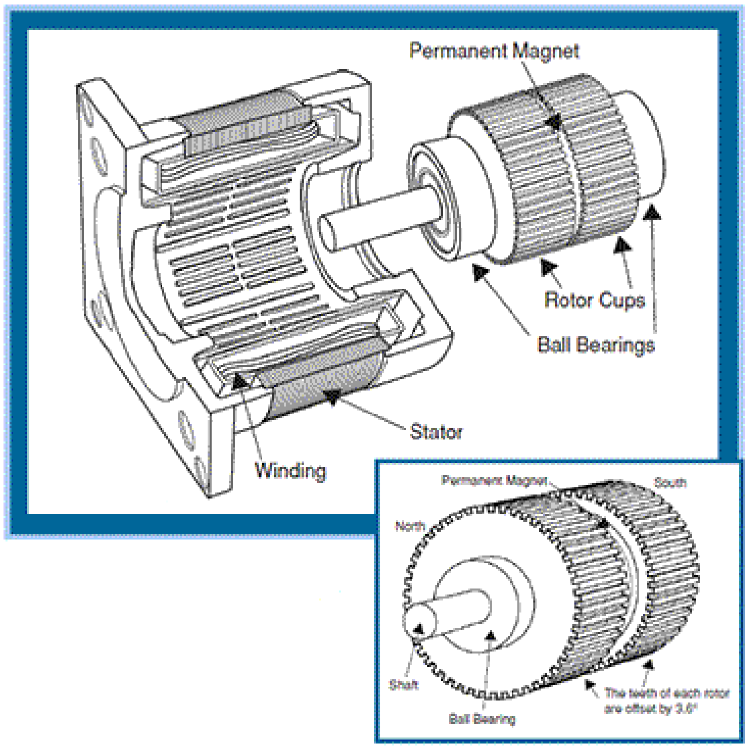


Figure 10: The Construction of a 2-phase Hybrid stepper motor. Image taken from NMB Technologies Corporation Website [6].

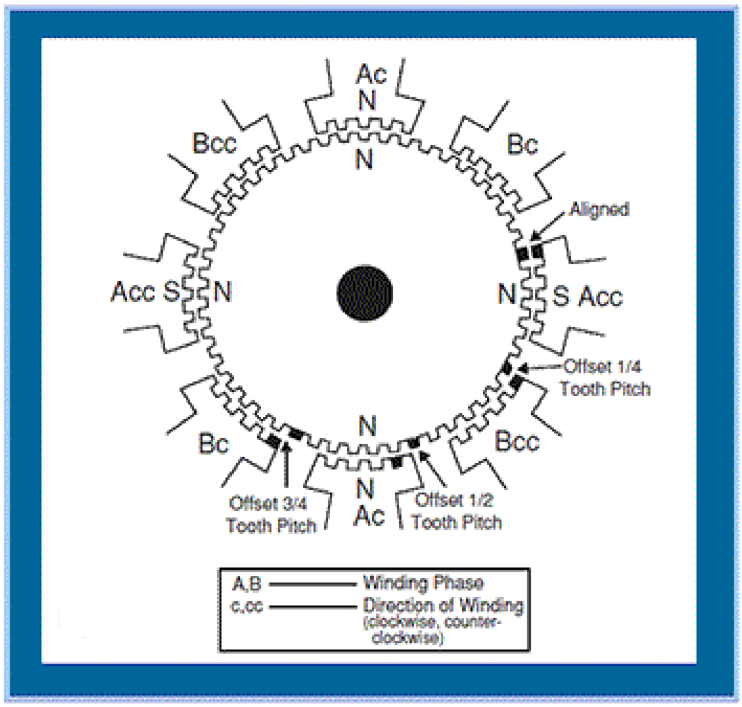


Figure 11: The 8 windings that make up the 2 phases (A and B) in a 2-phase hybrid stepper motor. Image taken from NMB Technologies Corporation Website [6].

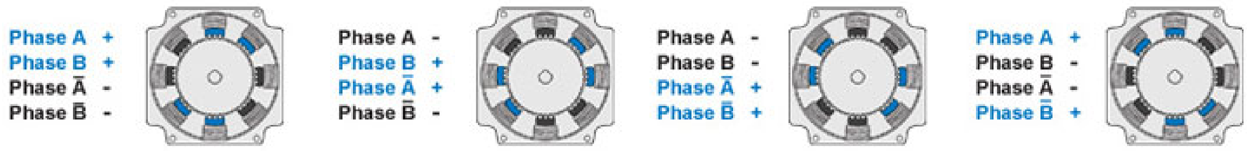


Figure 12: Full-step phase winding polarity progression to turn the stepper motor. Image taken from Orientalmotor website [10].

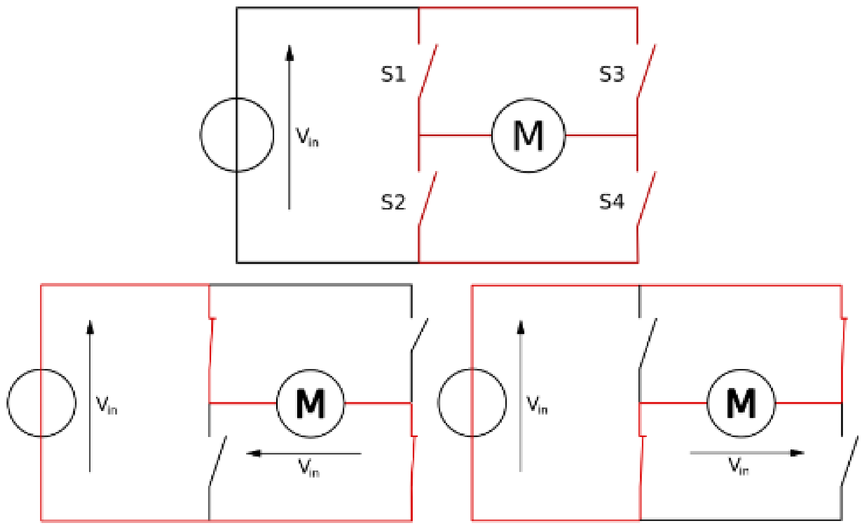
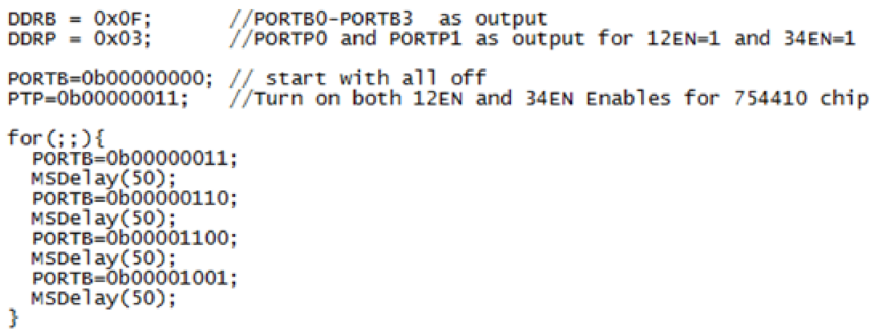


Figure 13: *above*: A full h-bridge circuit where switches S1-S4 are generally transistor switches and M is the load. *below*: The two valid operating configurations of a full h-bridge circuit.

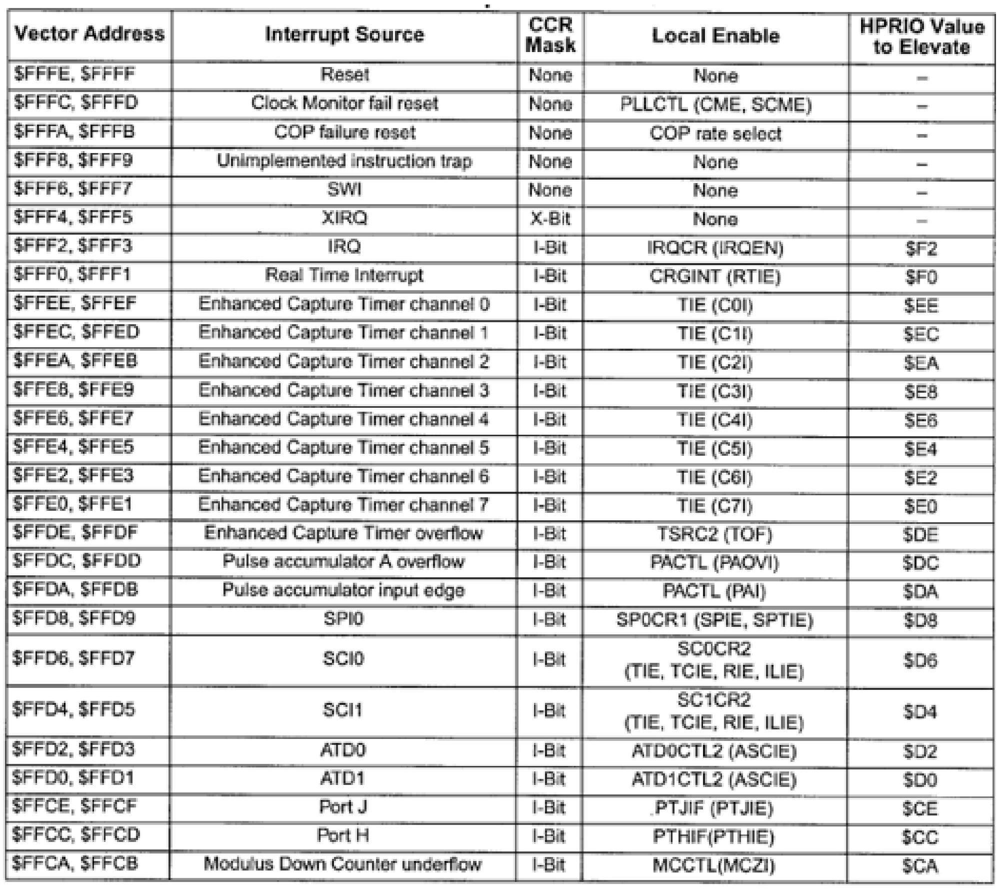


```
DDRB = 0x0F;       //PORTB0-PORTB3  as output
DDRP = 0x03;       //PORTP0 and PORTP1 as output for 12EN=1 and 34EN=1

PORTB=0b00000000; // start with all off
PTP=0b00000011;    //Turn on both 12EN and 34EN Enables for 754410 chip

for(;;){
  PORTB=0b00000011;
  MSDelay(50);
  PORTB=0b00000110;
  MSDelay(50);
  PORTB=0b00001100;
  MSDelay(50);
  PORTB=0b00001001;
  MSDelay(50);
}
```

Figure 14: Phase sequencing code for infinite clock-wise rotation. The mapping for PORTB to M1-M4 is as follows: PORTB = 0 b 0 0 0 0 M4 M3 M2 M1, where M1, M2, M3 and M4 are 1 when voltage is desired on that output connection and 0 when no voltage is desired. Code adapted from example stepper motor control code at MicroDigitalEd website [7].



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 15: Interrupt Vector Locations for the Dragon12 (HCS12) Development Board. Table taken from "Microcontroller Theory and Applications" [8].

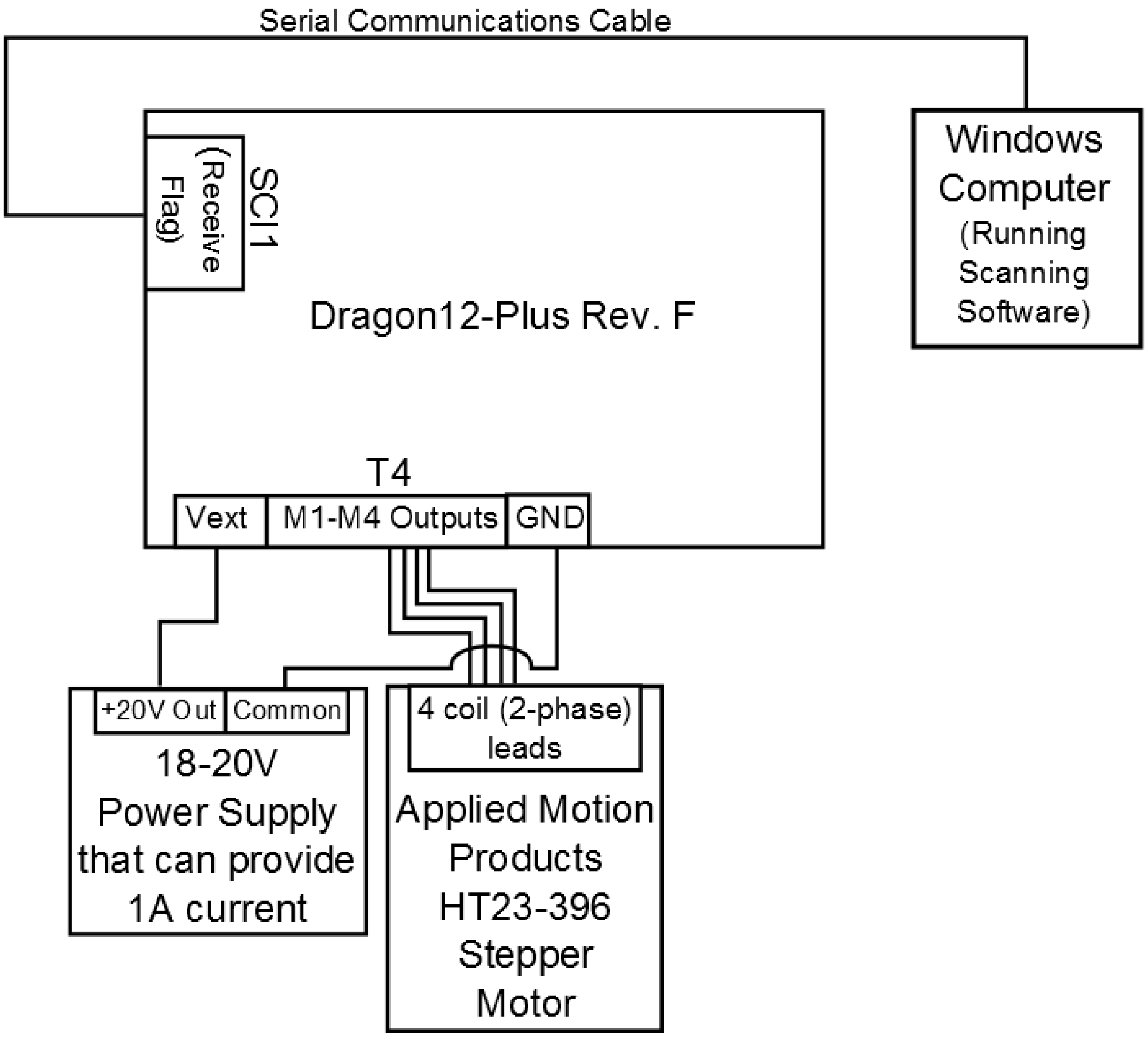

Serial Communications Cable
SCI1 (Receive Flag)
Dragon12-Plus Rev. F
Windows Computer (Running Scanning Software)
T4
Vext
M1-M4 Outputs
GND
+20V Out
Common
18-20V Power Supply that can provide 1A current
4 coil (2-phase) leads
Applied Motion Products HT23-396 Stepper Motor

Figure 16: The block diagram of the hardware systems associated with the Dragon12 for the purpose of controlling the stepper motor.

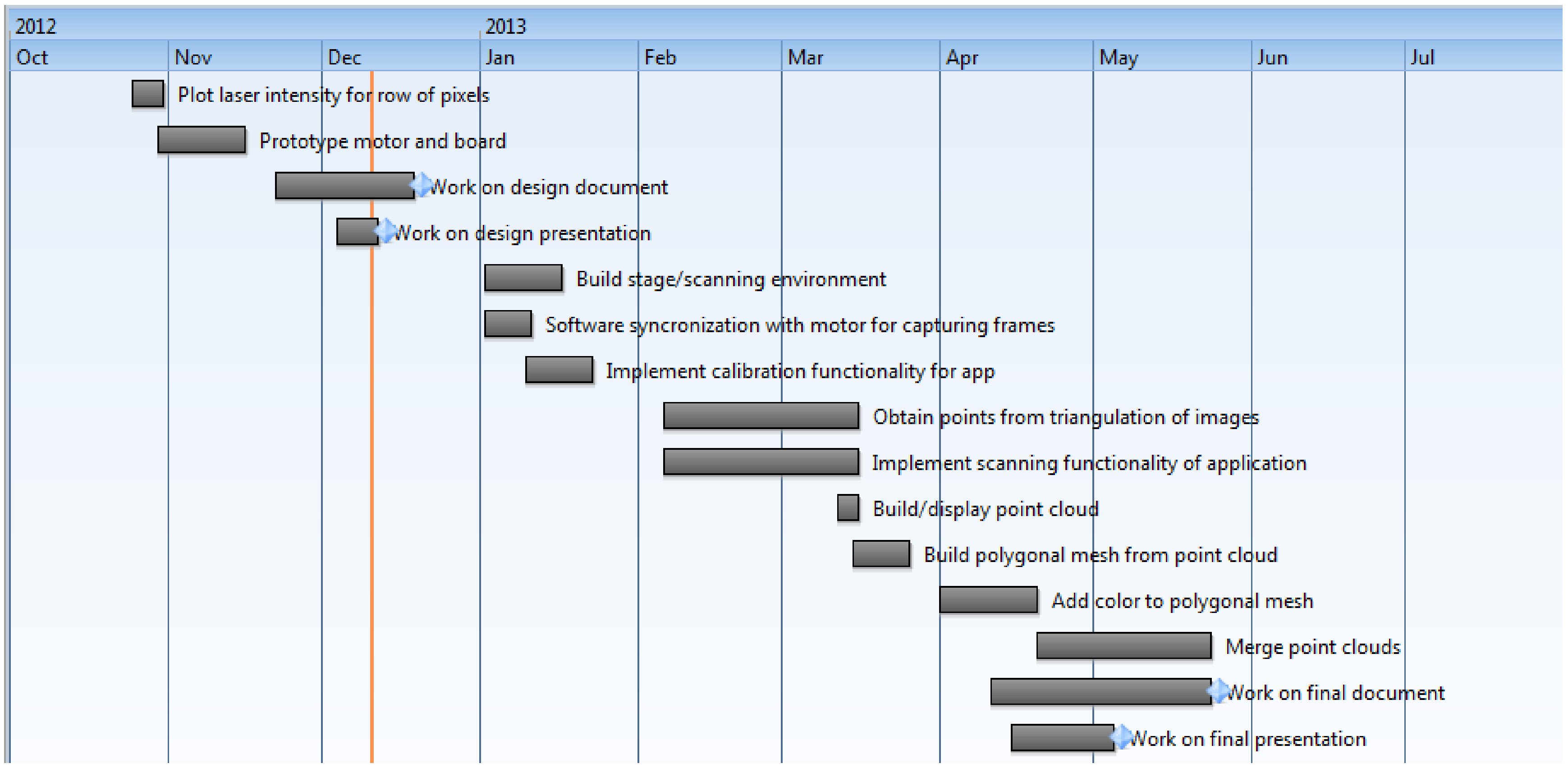


Figure 17: This is an updated version of our timetable.